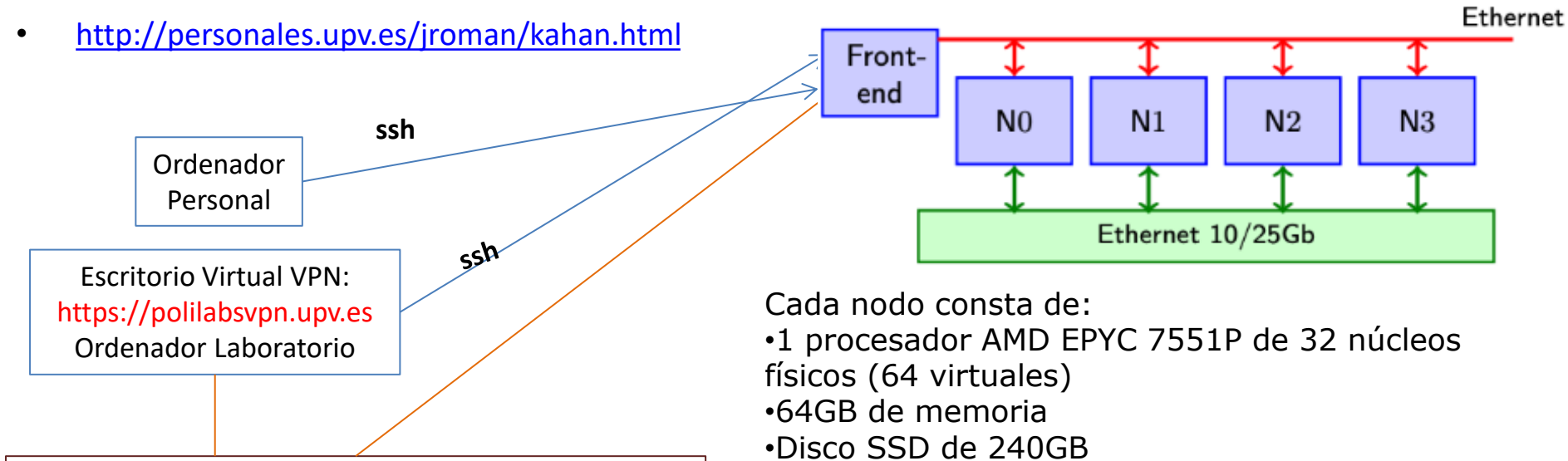
Práctica 3

Sesión 1

Cluster de prácticas Kahan

- http://personales.upv.es/jroman/kahan.html



Cada nodo consta de:
- 1 procesador AMD EPYC 7551P de 32 núcleos físicos (64 virtuales)
- 64GB de memoria
- Disco SSD de 240GB
- Ethernet 10/25Gb 2-port 622FLR -SFP28

DiscoW

- En este disco se copian los ficheros de las prácticas procedentes de Tareas de Poliformat
- Se pueden editar los códigos, compilar y realizar ejecuciones poco costosas para ver su correcto funcionamiento
- Los nombres de los directorios en DiscoW no pueden contener espacios en blanco

- En el **front-end** de **Kahan** (modo terminal):
  - Los códigos ejecutables deben estar almacenados en un directorio del **front-end**
  - Se pueden hacer cosas muy básicas: edición, compilación y ejecuciones poco costosas
- Las ejecuciones sobre el cluster se realizan desde el **front-end** mediante el sistema de colas **SLURM**, usando pruebas con un elevado coste computacional

En las tres siguientes sesiones tendréis que trabajar desde el Escritorio Virtual VPN o desde un ordenador personal vuestro

Contenido

- La práctica 2 consta de 5 sesiones:
  - Cálculo de Pi
  - Fractales de Newton
  - Producto matriz-vector
  - Sistemas de ecuaciones lineales
  - Examen individual de tipo test (1.5 sobre 10)
- Copia en una carpeta de la unidad W los ficheros anteriores necesarios para hacer la práctica, los cuales encontrarás en **Tareas** de Poliformat. Esta carpeta las puedes denominar como:

  W/cpa/prac2

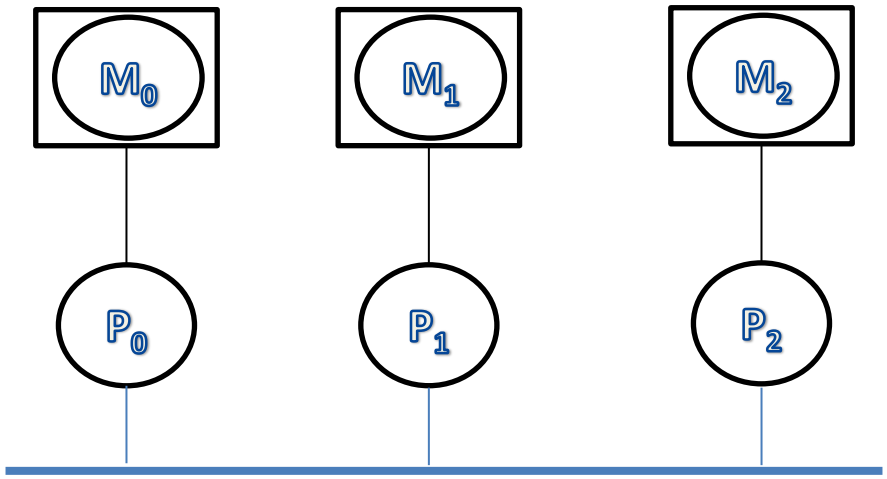
- Posteriormente, puedes almacenar en una carpeta de kahan estos ficheros y los que vayas creando.

Contenido primera sesión

- Primeros pasos con MPI:
  - Primer programa: hello.c
  - Modificación de hello.c, para mostrar el identificador de proceso y el número de procesos
- Comunicaciones punto a punto (MPI_Send, MPI_Recv):
  - Cálculo del número Pi
  - Determinación del modelo de tiempo de comunicaciones: programa ping-pong
- En las máquinas del laboratorio no está instalado OpenMPI, por lo que las pruebas se deben realizar en Kahan (pruebas pocos costosas en el front-end y más costosas lanzar al sistema de colas el trabajo)

Recordatorio

- Arquitectura de memoria distribuida



P_i: procesador (proceso) i-ésimo
M_i: memoria local del procesador P_i

- Todos los procesos tienen la misma copia del programa que se va a ejecutar
- Todos los procesos tienen las mismas variables, pero éstas son locales
- La única manera de intercambiar datos es el envío/recepción de mensajes
- Pueden ejecutar diferentes instrucciones dependiendo de sus identificadores
- Para que tenga éxito una comunicación es necesario que uno de los dos procesos implicados haga la petición del envío (**send**) y el otro la petición de recepción (**rec**):

P_0 : Send(…,1,…) ⟶ P_1 : Recv(…,0,…)

1.1 Primeros pasos con MPI

- Programa hello, compilación y ejecución:

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  printf("Hello\n”);
  MPI_Finalize();
  return 0;
 }
```

  - **Compilación**: mpicc –Wall -o hello hello.c
  - Con “mpicc –show” (se muestran las opciones usadas por el script)
  - **Ejecución**: mpiexec ./hello
  - Si se ejecuta en la línea de comandos, se puede indicar el número de procesos que se usan:

    mpiexec –n 4 hello (se ejecuta en el front-end con 4 procesos)

    de esta forma, todos los procesos se lanzan sobre el mismo nodo
  - Esa forma de ejecutar puede servir solo para pruebas rápidas

1.1 Primeros pasos con MPI

Script para lanzar el programa en el sistema de colas de Kahan el programa hello:

```
#!/bin/sh
#SBATCH --nodes=2
#SBATCH --ntasks=4
#SBATCH --time=5:00
#SBATCH --partition=cpa
scontrol show hostnames $SLURM_JOB_NODELIST

mpiexec ./hello
```

Se reservan 2 nodos de los 4 nodos de Kahan

Número de procesos MPI lanzados

Muestra los nodos usados de Kahan

Nota: como --ntasks=4 y –nodes=2, entonces se lanzarán de 4/2=2 procesos por nodo

1.1 Primeros pasos con MPI

Realizar estos ejercicios posteriormente

- Ejercicio 1: Realiza diversas ejecuciones del programa usando el sistema de colas, con diferente número de procesos y nodos.
- Ejercicio 2: Modifica el programa para obtener el identificador de proceso y el número de procesos, y mostrar esa información en el saludo. Recuerda que para ello debes usar las funciones **MPI_Comm_rank** y **MPI_Comm_size**

```
1  #include<stdio.h>
2  #include <mpi.h>
3  int main(int argc, char* argv[]) {
4      int id; /* identificador del proceso */
5      int p; /* número de procesos */
6      MPI_Init(&argc, &argv);
7      MPI_Comm_rank(MPI_COMM_WORLD, &id);
8      MPI_Comm_size(MPI_COMM_WORLD, &p);
9      printf("Hola mundo, soy el proceso %d de %d\n", id, p);
10     MPI_Finalize();
11     return 0;
12 }
```

1.2 Calculo de Pi

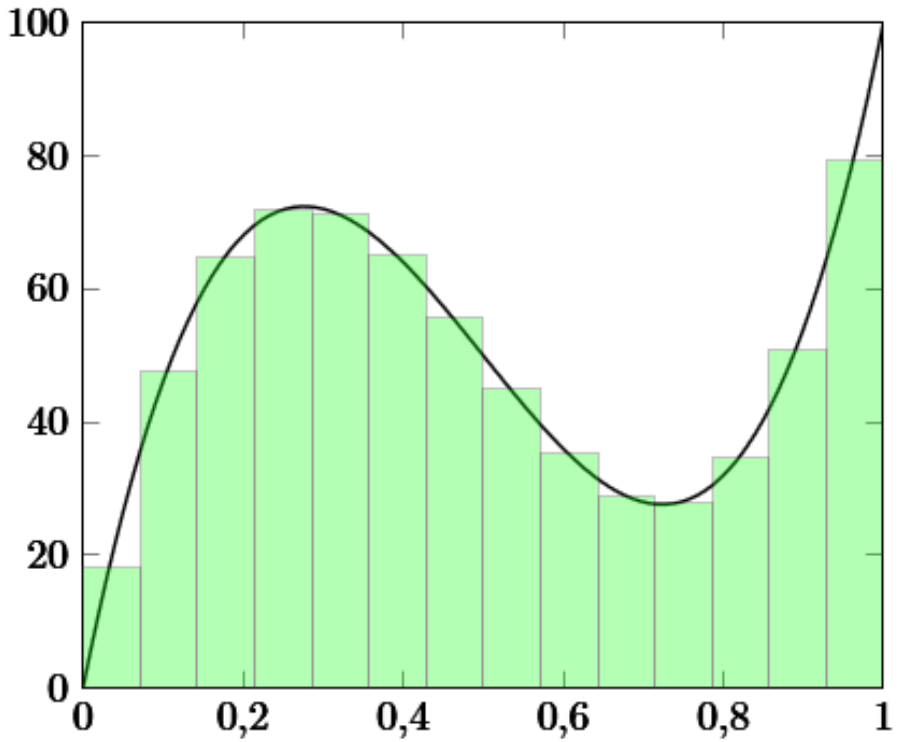
- Cálculo del número pi/4 en MPI (suponiendo que hay 3 procesos)

$$\int_0^1 \frac{1}{1+x^2}\,dx = \frac{\pi}{4} \cong h\sum_{i=1}^{n} f(x_i) =$$

Puntos medios:

$$x_i = h(i-0.5), i = 1, 2, \cdots n$$

$$h\sum_{i(P_0)} f(x_i) + h\sum_{i(P_1)} f(x_i) + h\sum_{i(P_2)} f(x_i)$$



h = tamaño del intervalo

x_i = puntos medios

$$f(x) = \frac{1}{1+x^2}$$

1.2. Cálculo de Pi (programa mpi_pi.c)

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

int main(int argc,char *argv[])
{
  int    n, myid, numprocs, i;
  double mypi, pi, h, sum, x;

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

  if (argc==2) n = atoi(argv[1]);
  else n = 100;
  if (n<=0) MPI_Abort(MPI_COMM_WORLD,MPI_ERR_ARG);

  /* Cálculo de PI. Cada proceso acumula la suma parcial de un subintervalo */

  h   = 1.0 / (double) n;
  sum = 0.0;
  for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += (4.0 / (1.0 + x*x));
  }
  mypi = h * sum;

  /* Reducción: el proceso 0 obtiene la suma de todos los resultados */
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

  if (myid==0) {
    printf("Cálculo de PI con %d procesos\n",numprocs);
    printf("Con %d intervalos, PI es aproximadamente %.16f (error = %.16f)\n",n,pi,fabs(pi-M_PI));
  }

  MPI_Finalize();
  return 0;
}
```

Reparto cíclico del cálculo de las sumas parciales

$$\text{mypi}(P_0) = h \sum_{i(P_0)} f(x_i)$$

$$\text{mypi}(P_1) = h \sum_{i(P_1)} f(x_i)$$

$$\text{mypi}(P_2) = h \sum_{i(P_2)} f(x_i)$$

$$\text{pi}(P_0) = \text{mypi}(P_0) + \text{mypi}(P_1) + \text{mypi}(P_2)$$

1.2. Cálculo de Pi

Ejercicio 3: Modifica el programa anterior, sustituyendo la llamada a la función **MPI_Reduce** por un fragmento de código equivalente, pero utilizando solo comunicaciones punto a punto (pruébalo solo en **polilabs**)

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

- mypi=variable local que contiene el dato aportado por cada proceso
- pi= valor local del proceso que contendrá el resultado, en este caso la suma de todos los valores mypi
- Los valores 1 y MPI_DOUBLE indican que las variables mypi y pi son datos simples de double precisión
- MPI_SUM indica que la operación de reducción es una suma
- El valor 0 indica que el proceso P_0 recibirá en su variable pi el resultado
- MPI_COMM_WORLD=comunicador universal

	mipi
P_0	mypi(P_0)
P_1	mypi(P_1)
P_2	mypi(P_2)

P_0	pi(P_0)=mypi(P_0)+mypi(P_1)+mypi(P_2)
P_1	
P_2	

1.2. Cálculo de Pi

	mipi
P_0	mypi(P_0)
P_1	mypi(P_1)
P_2	mypi(P_2)

P_0	pi(P_0)=mypi(P_0)+mypi(P_1)+mpi(P_2)
P_1	
P_2	

Código a implementar mediante comunicaciones punto a punto:

Si soy P0

```
pi=mypi;
Para i=1, 2,…,numprocs-1{
        Recv(mypi, Pi) /*P0 recibe el valor mypi del proceso Pi*/
         /*mejor: recibir de cualquiera→ Recv(mypi, any)*/
         pi += mypi;
 }
```

Escribir el resultado obtenido (pi) y el error cometido (tal como está en el código)

En caso contrario

Send(mypi,P_0)

Envio/recepción estándar

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

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

senbuf/recbuf puntero al dato que se va a envíar/recibir

- Dato simple lleva delante &; array no tiene delante &

count: número de datos a enviar/recibir

datatype : tipo de dato (MPI_DOUBLE, MPI_INT, MPI_CHAR, etc.)

- dato simple: 1, array: número de elementos del array

dest/source: Identificador del proceso al que se va a envíar el mensajeo del proceso del cual queremos recibir el mensaje

tag: etiqueta del mensaje (entero positivo)

comm: comunicador, normalmente se usa el comunicador universal **MPI_COMM_WORLD**

status: información del mensaje:

- status.MPI_SOURCE: proceso que ha realizado el envío
- status.MPI_TAG: etiqueta del mensaje recibido

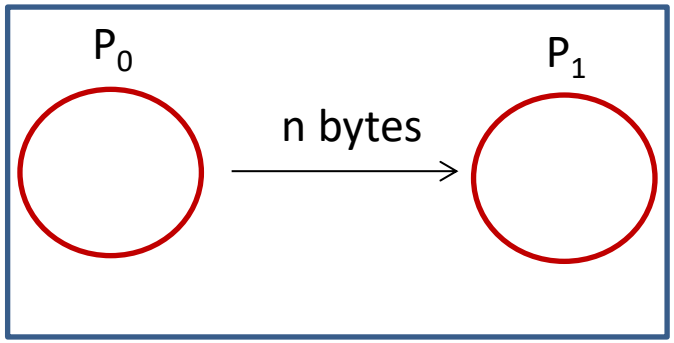
En MPI_Recv se pueden usar las constantes:

MPI_ANY_SOURCE (en source), MPI_ANY_TAG(tag), MPI_STATUS_IGNORE(en *status)

1.3 El programa Ping-Pong

Ejercicio 4: Se quiere calcular aproximadamente el tiempo de envío de un mensaje con n bytes. Para eso, completa el programa ping-pong.c, teniendo en cuenta:

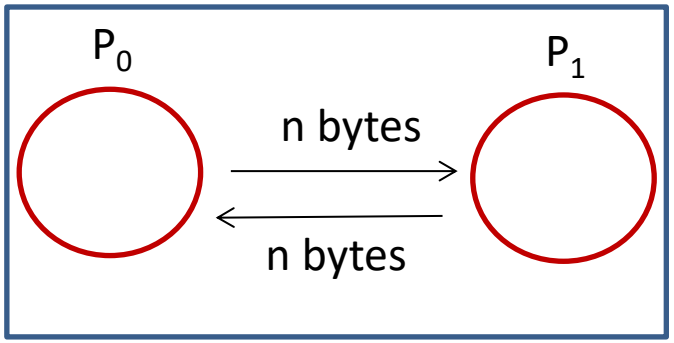
- El programa tiene como argumento el tamaño de mensaje n.
- Usar `MPI_Wtime)` para medir tiempos.
- Para que los tiempos medidos sean significativos, el programa debe repetir la operación `NREPS` veces y mostrar el tiempo medio.
- Las operaciones de envío y recepción se realizan con `MPI_Send` y `MPI_Recv` con `MPI_BYTE` como tipo de dato de envío/recepción.



1.3 El programa Ping-Pong

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define NMAX 1000000
#define NREPS 100
char buf[NMAX];
int main(int argc,char *argv[])
{
  int n, myid, numprocs;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  /* The program takes 1 argument: message size (n), with a default size of 100
     bytes and a maximum size of NMAX bytes*/
  if (argc==2) n = atoi(argv[1]);
  else n = 100;
  if (n<0 || n>NMAX) n=NMAX;

  /* COMPLETE: Get current time, using MPI_Wtime() */

  /* COMPLETE: loop of NREPS iterations.
     In each iteration, P0 sends a message of n bytes to P1, and P1 sends the same
     message back to P0. The data sent is taken from array buf and received into
     the same array. */


  /* COMPLETE: Get current time, using MPI_Wtime() */


  /* COMPLETE: Only in process 0.
     Compute the time of transmission of a single message (in milliseconds) and print it.
     Take into account there have been NREPS repetitions, and each repetition involves 2
     messages. */

  MPI_Finalize();
  return 0;
}
```

Repetir NREPS veces:



P0 calcula el tiempo total

P0 calcula e imprime el tiempo medio

Usar el tipo MPI_BYTE para enviarlos datos

Observa que el tiempo que se requiere para calcular aproximadamente el tiempo de envío de un mensaje con n bytes se puede calcular aproximadamente dividiendo el tiempo calculado en el programa entre 2* NREPS

1.3 El programa Ping-Pong

Lanzar el trabajo al cluster Kahan

- Conéctate a Kahan y copia los ficheros a un directorio del Front-end.
- Compila el programa ping-pong y crea un script para enviarlo al sistema de colas.
- Lanza el trabajo al sistema de colas mediante **sbatch**

Script de ejemplo:

```
#!/bin/sh
#SBATCH --output=ping-pong.txt
#SBATCH --nodes=2
#SBATCH --ntasks=2
#SBATCH --time=5:00
#SBATCH --partition=cpa
scontrol show hostnames $SLURM_JOB_NODELIST

mpiexec  ./pingpong
```

1.3 El programa Ping-Pong

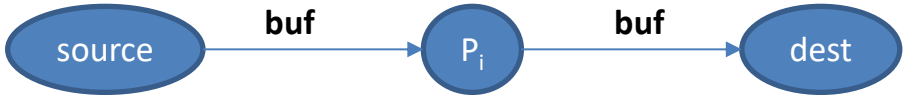
- Ejercicio 5: ¿Por qué se envían dos mensajes en cada iteración del bucle? ¿se podría eliminar el mensaje de respuesta de *P*1 a *P*0?
- Ejercicio 6: En cada iteración, el proceso *P*0 tiene que hacer un envío y una recepción. ¿Podría utilizar para ello la función **MPI_Sendrecv_replace**? ¿y el proceso *P*1?

```
int MPI_Sendrecv_replace(void *buf, int
    count, MPI_Datatype datatype, int
    dest, int sendtag, int source, int
    recvtag, MPI_Comm comm, MPI_Status
    *status)
```

- Esta operación combina en una sola llamada el envío de un mensaje a un destino y la recepción de otro mensaje procedente de otro proceso, utilizando la misma variable.

Operaciones combinadas en MPI

MPI_Sendrecv_replace(void ***buf**, int count, MPI_Datatype datatype, int **dest**, int sendtag, int **source**, int recvtag, MPI Comm_comm, MPI_Status *status)



- P_i es el proceso que invoca a la función
- Ambas operaciones combinan en una sola llamada el envío de un mensaje a un destino y la recepción de otro mensaje, procedente de otro proceso. Los dos (origen y destino) pueden ser el mismo.
- Esta operación es útil para ejecutar una operación de desplazamiento a través de una cadena de procesos.
- La diferencia está en que en MPI_Sendrecv se utilizan diferentes variables para el envío y recepción y en MPI_Sendrecv_replace la misma.

¿Es posible hacer esto y que funcione con MPI_Sendrecv_replace:?