ikd baliabideak 12 (2016)

# Programación paralela: MPI

## Cuaderno del Estudiante



**Olatz Arbelaitz Gallego**

**Olatz Arregi Uriarte**

**Agustin Arruabarrena Frutos**

**José Ignacio Martín Aramburu**

**Javier Muguerza Rivero**

**ÍNDICE**

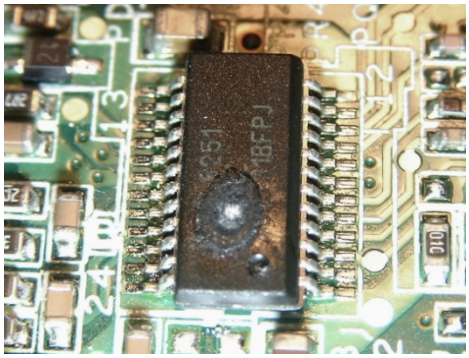
# 1. CONTEXTO DEL PROYECTO

## 1.1 INTRODUCCIÓN

El proyecto a realizar se centra en la segunda parte de la asignatura Sistemas de Cómputo Paralelo (3er curso del Grado en Ingeniería en Informática, especialidad en Ingeniería de Computadores): Programación paralela en MPI. El proyecto abarcará un 60% del total de la asignatura, esto es, 3,6 créditos ECTS. Esta dedicación supone un total de 36 horas presenciales y de 54 horas no presenciales. Dado que se formarán grupos de 3 estudiantes, dicha carga supone una dedicación de 270 horas por grupo.

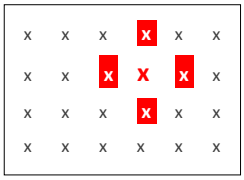
## 1.2 PREGUNTA MOTRIZ

¡Cuidado, que quema!



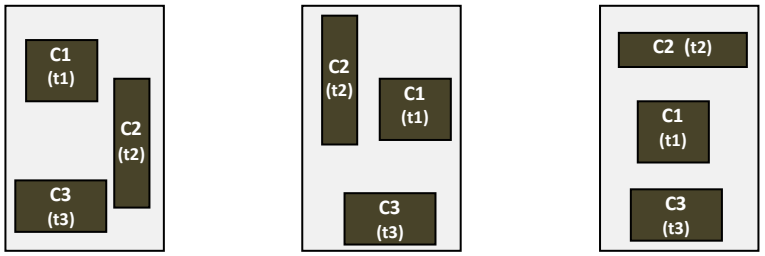
## 1.3 ESCENARIO

La empresa TXIPSA se dedica a la fabricación de placas de circuitos impresos para diversos usos. Estas placas contienen varios chips que se calientan a diferentes temperaturas en su uso normal, por lo que deben ser colocados de forma estratégica en la placa para reducir la temperatura global del sistema y evitar que la placa se queme y deje de funcionar.



Antes de fabricar el circuito final, se simula un algoritmo de difusión del calor usando diferentes localizaciones de los chips en la tarjeta, para elegir aquella configuración que minimiza la temperatura media global de la tarjeta. Para aplicar el algoritmo de difusión del calor, se divide la tarjeta en una rejilla bidimensional de puntos, y se va modificando la temperatura de cada punto teniendo en cuenta la **temperatura de los puntos vecinos**, hasta que, iteración tras iteración, el sistema converge a una determinada temperatura media, que depende de la posición de las fuentes de calor, los chips de la tarjeta.

La simulación se hace en un equipo monoprocesador analizando el comportamiento de diferentes configuraciones, hasta obtener la mejor solución. La siguiente figura muestra tres posibles configuraciones de una tarjeta de circuito impreso con tres chips, en las que, tras el proceso de difusión del calor, se alcanzan diferentes temperaturas medias en la tarjeta.




Config. 1 — $T_m1$ Config. 2 — $T_m2$ Config. 3 — $T_m3$

Recientemente TXIPSA ha adquirido un *cluster* de 33 procesadores de **memoria distribuida**, conectados con una red Gigabit Ethernet, y quiere paralelizar el programa de difusión del calor y cálculo de la temperatura media. Así, **cada procesador se encargará de simular el comportamiento térmico de un trozo de la tarjeta**, calculando entre todos los procesadores la temperatura media final. De esa manera, se pretende realizar más simulaciones en menos tiempo, lo que redundará en una reducción del tiempo de fabricación, y, en su caso, en un aumento de los beneficios.

La empresa te ha encargado que paralelices de manera eficiente la fase de simulación térmica de diferentes configuraciones de los chips en una tarjeta.

### 1.4 TEMARIO A CUBRIR Y RESULTADOS DE APRENDIZAJE

El proyecto desarrolla el tema 3 de la asignatura, Programación paralela: MPI. Se abordará el análisis de los problemas que surgen al desarrollar aplicaciones eficientes para ser ejecutadas en sistemas paralelos de memoria distribuida: la comunicación entre procesadores, la definición adecuada de tipos de datos que minimicen el coste de la comunicación, y diversos modelos de planificación de carga (estáticas y dinámicas, basadas éstas en el modelo *manager*/*worker*). Para ello, se trabajará con MPI, la herramienta estándar para la programación de sistemas paralelos de memoria distribuida.

Al término de este proyecto deberás ser capaz de:

- Programar de forma eficiente pequeñas aplicaciones en computadores de memoria distribuida, utilizando el estándar MPI.
- Analizar el rendimiento que se obtiene en un sistema de cómputo paralelo y de las aplicaciones que en él se ejecuten.

Así mismo, se trabajan otras competencias generales y transversales (ver página web del Grado en Ingeniería Informática, apartado “Plan de estudios”).

### 1.5 LISTA DE ENTREGABLES

**E1** Acta de constitución del grupo y documento de compromisos de los componentes del grupo (E1.1), junto con las actas de reuniones realizadas para llevar a cabo el proyecto (E1.2).

**E2** Póster realizado en el análisis del escenario.

**E3** Recopilación del trabajo realizado por cada persona en el puzle: (E3.1) breve informe que incluye el trabajo teórico realizado y la resolución/análisis de los ejercicios planteados; (E3.2) presentación del puzle. Se incorporará el material teórico utilizado para el estudio de las tareas (siempre que se trate de material novedoso no entregado por el profesorado).

**E4** Actas de las evaluaciones por pares. Por una parte, acta de la evaluación de la presentación del puzle (E4.1) y, por otra, acta de la evaluación de la presentación de la aplicación desarrollada (E4.2).

**E5** Examen de control de conocimientos mínimos adquiridos en el desarrollo del proyecto.

**E6** Fin de la primera fase de la implementación de la aplicación: breve informe (1 hoja) con los principales resultados obtenidos (E6.1). Informe técnico final de la aplicación desarrollada (E6.2), así como el material realizado para la presentación de la misma (E6.3).

**E7** Portafolio o carpeta con el material generado durante el desarrollo del proyecto y que el grupo considere adecuado para su mejor interpretación. Aunque se entregará la versión definitiva al finalizar el proyecto, se entregará una primera versión con el material generado hasta el puzle, que será revisado para que pueda ser mejorado.

Todos los entregables son cooperativos, grupales, salvo el entregable E5 (examen de conocimientos mínimos), que será un entregable individual.

### 1.6 SISTEMA DE EVALUACIÓN

El proyecto (segunda parte de la asignatura) se evalúa en 6 puntos, de acuerdo a los siguientes criterios:

- Presentación individual. En el proyecto se contemplan dos presentaciones: puzle y aplicación desarrollada. Cada grupo hará una de las presentaciones. La persona que realice la presentación será elegida en el momento. Dado que esta actividad es colaborativa, la nota obtenida será la nota de todo el grupo. Las presentaciones serán evaluadas por el profesorado y el alumnado, y su nota será de **1 punto**.
- Examen individual de conocimientos mínimos. Su valoración será de **2 puntos**. Para superar el proyecto, se deberá obtener una nota mínima en este examen de 30%.

- Informe técnico que describa la aplicación desarrollada y analice el rendimiento obtenido. Esta actividad valdrá **3 puntos**.

- Carpeta o portafolio del proyecto. Esta actividad es de tipo filtro: se calificará como APTA o NO APTA, pero no afecta a la calificación final. En cualquier caso, debe ser superada.

Se considerarán como puntos extra actividades desarrolladas de forma extraordinaria (entre otros aspectos, por ejemplo, un portafolio bien estructurado, actualizado, etc.).

**Para aprobar la asignatura hay que obtener al menos 3 puntos en el proyecto**. La siguiente tabla resume la evaluación de cada una de las actividades y quién la efectúa.

| | **Nota** | **Docentes** | | **Estudiantes** | |
|---|---|---|---|---|---|
| **Individual** | 2 puntos | Examen (E5) | 2 p. | | |
| | | Presentaciones (E3.2/E6.3) | 0,5 p. | Presentaciones (E4.1/E4.2) | 0,5 p. |
| **Grupo** | 4 puntos | Informe técnico (E6.2) | 3 p. | | |
| | | Portafolio final (E7) | filtro | | |
| **Total** | **6 puntos** | | **5,5 puntos** | | **0,5 puntos** |

### 1.7 PLANIFICACIÓN DEL TRABAJO

La siguiente tabla muestra un resumen de las actividades a desarrollar, de la carga de trabajo estimada (presencial y no presencial), los entregables a desarrollar, y los hitos de evaluación del proyecto, semana por semana.

En el curso 2015/16, las actividades presenciales se desarrollan en tres sesiones de 1,5 horas los lunes, martes y miércoles de cada semana. Las semanas del 21-23 de marzo (semana santa) y del 16-20 de mayo (fin de trabajos) no tienen actividades presenciales, pero son lectivas. En la semana de horario agrupado, 18-22 de abril, está prevista una visita al centro de cálculo del DIPC (*Donostia International Physics Center*).

## PLANIFICACIÓN DEL TRABAJO DEL ALUMNADO

| Semana | Clase | Actividades presenciales | t/día | Actividades no presenciales (por semana) | t/sem | Entregables | Evaluación |
|---|---|---|---|---|---|---|---|
| 29 -4/03 | 1 | Entrega del escenario y reflexión por grupos | 40 m | | | Acta: constitución de grupo | |
| | | Discusión PBL, póster final | 30 m | | | (E1.1) | |
| | | Planificación del proyecto | 20 m | | | Póster (din-A4) (E2) | |
| | 2 | Programación en MPI | 90 m | | 3 h | | |
| | 3 | Programación en MPI | 60 m | Estudio programación en MPI | | | |
| | | Delimitación de tareas concretas: puzle | 30 m | Estudio individual del problema asignado en el puzle | | | |
| 7-11/03 | 4 | Estudio individual del problema asignado en el puzle | 90 m | Estudio individual del problema asignado en el puzle | 4 h | | |
| | 5 | | 90 m | | | | |
| | 6 | | 90 m | | | | |
| 14-18/03 | 7 | Reunión de grupo: puesta en común | 90 m | Estudio de todos los conceptos del puzle | 4 h | | |
| | 8 | Reunión de expertos: puesta en común | 90 m | Preparación del informe y presentación del puzle | | | |
| | 9 | Reunión de grupo: puesta en común | 80 m | Ejercicios de representación gráfica | | | |
| | | Enunciado de ejercicios de representación gráfica | 10 m | | | | |
| 21-23/03 | | | | Preparación del informe y presentación del puzle<br>Ejercicios de representación gráfica | 7 h | | |
| 4-8/04 | 10 | Ejercicios de representación gráfica | 60 m | Preparación presentación puzle | 4 h | | |
| | | Reunión de grupo: puesta en común | 30 m | | | Puzle: informe (E3.1), | |
| | 11 | Puzle: presentación | 60 m | | | presentación (E3.2), acta de | [1 p. (E3.2, E4.1) ] |
| | | Puzle: debate aclaratorio | 30 m | | | eval. (E4.1) | |
| | 12 | Puzle: debate aclaratorio | 60 m | Implementación de la aplicación (Fase 1) | | Portafolio para revisión (E7) | |
| | | Enunciado de la aplicación (Fase 1) | 30 m | | | | |
| 11-15/04 | 13 | Trabajo: desarrollo de la aplicación | 90 m | Desarrollo de la aplicación | 4 h | | |
| | 14 | | 90 m | | | | |
| | 15 | | 90 m | | | | |
| 18-22/04 | | (visita DIPC) | | Desarrollo de la aplicación | 6 h | | |
| 25-29/04 | 16 | Trabajo: desarrollo de la aplicación | 90 m | Desarrollo de la aplicación | 4 h | | |
| | 17 | Trabajo: desarrollo de la aplicación | 90 m | | | | |
| | 18 | Teoría: punto a punto, *deadlock* | 50 m | Desarrollo de la aplicación (Fase 2) | | Resultados preliminares: Fase 1 | |
| | | Debate de resultados preliminares | 30 m | | | de la aplicación(E6.1) | |
| | | Enunciado de la aplicación (Fase 2) | 10 m | | | | |
| 2-6/05 | 19 | Trabajo: desarrollo de la aplicación (Fase 2) | 90 m | Desarrollo de la aplicación | 4 h | | |
| | 20 | | 90 m | | | | |
| | 21 | | 90 m | | | | |
| 9-13/05 | 22 | Demo: Jumpshot | 90 m | Desarrollo / documentación | 4 h | | |
| | | Trabajo: desarrollo de la aplicación, documentación | | | | | |
| | 23 | | 90 m | | | | |
| | 24 | | 90 m | | | | |
| 16-20/05 | | | | Preparación de la presentación / Estudio | 7 h | Infor. técnico: Fases 1/2 (E6.2) | 3 p. |
| 24/05 | 25 | Defensa de la aplicación desarrollada | 90 m | | 3 h | Acta: eval. de la present. (E4.2) | [1 p.(E6.3, E4.2)] |
| | | Examen de conocimientos mínimos | 90 m | | | Examen (E5) | 2 p. (E.5) |
| | | | | | | Portafolio (E7, filtro) | |

# 2. RECURSOS

Este apartado incorpora todos los recursos entregados al alumnado para la realización del proyecto: material de laboratorio, enunciados del puzle, enunciados de los ejercicios a resolver y el enunciado de la aplicación a desarrollar.

## 2.1 MATERIAL PARA EL LABORATORIO DE INTRODUCCIÓN A MPI Y AL *CLUSTER*

Antes de comenzar con el puzle, vamos a trabajar en dos sesiones de laboratorio cómo usar el *cluster* y los conceptos básicos de MPI.

### 2.1.1 Puesta en marcha del *cluster*

Vamos a trabajar con un *cluster* sencillo de 32 nodos (Intel Core 2 6320 - 1,8 GHz - 2 GB RAM - 4 kB cache) más un nodo similar para desarrollo, conectados todos mediante Gigabit Ethernet en una red local. Es un sistema muy simple, pero permite ejecutar todo tipo de aplicaciones paralelas mediante paso de mensajes, analizar su comportamiento, etc.

El nodo de desarrollo hace las veces de servidor de ficheros y de punto de entrada al *cluster*; su dirección externa es `g002615.gi.ehu.eus`, y su dirección en el *cluster* es `servidor01` (`nodo00`). El resto de los nodos —`nodo01`, `nodo02`... `nodo32`— solo son accesibles desde el nodo de entrada (no tienen conexión al exterior). En el laboratorio realizaremos la conexión remota desde una sesión local Linux utilizando el comando:

```
> ssh cuenta@g002615.gi.ehu.eus
```

El directorio `templates` contiene en tres carpetas —ejemplos, puzle, aplicación— los ejemplos y programas con los que vamos a trabajar. Haz una copia de esos ficheros a una carpeta en el directorio de trabajo; por ejemplo:

```
> mkdir ejemplos
> cp templates/ejemplos/* ejemplos
```

**▪ Generación del entorno de ejecución remoto**

Para poder ejecutar aplicaciones utilizando diferentes máquinas necesitamos que el sistema pueda entrar en ellas usando `ssh` pero sin *password*. Para ello, hay que haber entrado una primera vez en cada máquina, para que nos reconozca como "usuarios autorizados".

1. En el directorio de entrada, se ejecuta:

   ```
   > ssh-keygen -t rsa
   ```
   (respondiendo con `return` las tres veces)

   Se genera el directorio `.ssh` con los ficheros con claves `id.rsa` e `id_rsa.pub`. Pasa a ese nuevo directorio y ejecuta:

   ```
   > cp id_rsa.pub authorized_keys
   > chmod go-rw authorized_keys
   ```

De nuevo en el directorio principal hay que entrar y salir, una por una, en las máquinas del *cluster*:

```
> ssh nodoxx            (xx = 01 ... 32)
      (yes)
> exit
```

Estas operaciones las hemos definido en el fichero de comandos `cluster-ssh.sh`, para poder ejecutarlas automáticamente.

**▪ MPICH2**

Vamos a utilizar la implementación MPICH2 de MPI (de libre distribución, la podéis instalar en vuestro PC; otra implementación de gran difusión es Open MPI). Previo a ejecutar programas en paralelo, hay que:

2. Crear un fichero de nombre `.mpd.conf` que contenga una línea con el siguiente contenido:

   `secretword=xxxx` (xxxx = cualquier palabra)

   y cambiarle los permisos: `> chmod 600 .mpd.conf`

3. Crear un fichero con las lista de las máquinas que vamos a utilizar (por defecto, de nombre `mpd.hosts`). En este caso basta con escribir:

```
nodo00
nodo01
…
nodo32
```

Estas dos operaciones también están incluidas en el fichero de comandos `cluster-ssh.sh`.

4. Ya solo queda poner en marcha el "entorno" MPICH2 (*daemons* `mpd` que se van a ejecutar en cada máquina del *cluster*):

   `> mpdboot -v -n num_proc [-f fichero_maquinas]`

   (`-f` si el fichero con la lista de las máquinas no es `mpd.hosts`).

Otros comandos útiles:

| | |
|---|---|
| `> mpdtrace` | devuelve las máquinas "activas" |
| `> mpdlistjobs` | lista los trabajos en ejecución en el anillo de *daemons* |
| `> mpdringtest 2` | recorre el anillo de máquinas (2 veces) y devuelve el tiempo |
| `> mpd &` | lanza un solo *daemon* en el procesador local |
| `> mpdhelp` | información de los comandos |

Si ha habido algún problema con la generación de *daemons*, etc., ejecutad `mpdcleanup` y repetid el proceso.

5. A continuación podemos compilar y ejecutar programas:

| | |
|---|---|
| `> mpicc [-Ox] -o pr1 pr1.c` | [x=1-3 nivel de optimización; por defecto, 2] |
| `> mpiexec -n xx pr1` | xx = número de procesos |

Ejecuta el programa `pr1` en `xx` procesadores (con el *flag* `-1` no se utiliza el `nodo00` para ejecutar los procesos)

```
> mpiexec -n 1 -host nodo00 p1 : -n 1 -host nodo01 p2
> mpiexec -configfile procesos
```

Lanza dos programas, `p1` y `p2`, en los nodos 00 y 01, o bien tal como se especifica en el fichero procesos.

6. Finalmente, para terminar una sesión de trabajo ejecutamos:

```
> mpdallexit
```

(para más información sobre estos comandos: comando `--help`)

### 2.1.2 Conceptos básicos de MPI: ejemplos y ejercicios

Antes de hacer frente a la primera parte del proyecto, el puzle, presentaremos mediante transparencias y ejemplos en el laboratorio las características básicas de MPI.

- Definición. Arquitectura de la máquina y paso de mensajes.
- Las funciones básicas:

  Comienzo y final del programa: `MPI_Init`, `MPI_Finalize`

  Identificación de procesos: `MPI_Comm_rank`, `MPI_Comm_size`

  Envío y recepción de mensajes punto a punto: `MPI_Send`, `MPI_Recv`, `MPI_Probe`

Los siguientes cuatro programas —`hola.c`, `circu.c`, `envio.c` y `probe.c`— los trabajaremos en las dos primeras sesiones de laboratorio.

```
/*************************************************************
       hola.c
       programa MPI: activacion de procesos
*************************************************************/

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

int main(int argc, char *argv[])
{
  int   lnom;
  char  nombrepr[MPI_MAX_PROCESSOR_NAME];
  int   pid, npr;                       // identificador y numero de proc.
  int   A = 21;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);

  MPI_Get_processor_name(nombrepr, &lnom);

  A = A + 1;
  printf(" >> Proceso %2d de %2d activado en %s, A = %d\n", pid,npr,nombrepr,A);

  MPI_Finalize();
  return (0);
 }



/*************************************************************************
       circu.c
       paralelizacion MPI de un bucle
*************************************************************************/

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

#define DECBIN(n,i) ((n&(1<<i))?1:0)

void test (int pid, int z)
{
  int v[16], i;

  for (i=0; i<16; i++) v[i] = DECBIN(z,i);

  if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3]) && (!v[3] || !v[4])
    && (v[4] || !v[5]) && (v[5] || !v[6]) && (v[5] || v[6]) && (v[6] || !v[15])
    && (v[7] || !v[8]) && (!v[7] || !v[13]) && (v[8] || v[9]) && (v[8] || !v[9])
    && (!v[9] || !v[10]) && (v[9] || v[11]) && (v[10] || v[11])
    && (v[12] || v[13]) && (v[13] || !v[14]) && (v[14] || v[15]))
  {
    printf("  %d)  %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d  (%d)\n",  pid,  v[15],v[14],v[13],
        v[12],v[11],v[10],v[9],v[8],v[7],v[6],v[5],v[4],v[3],v[2],v[1],v[0], z);
    fflush(stdout);
  }
}

int main (int argc, char *argv[])
{
  int   i, pid, npr;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);

  for (i=pid; i<65536; i += npr)  test(pid, i);

  MPI_Finalize();
  return (0);
}
```

```
/*****************************************************************************
        envio.c
        se envia un vector desde el procesador 0 al 1
*****************************************************************************/

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

#define N 10

int main (int argc, char **argv)
{
  int   pid, npr, origen, destino, tag, ndat;
  int   VA[N], i;
  MPI_Status    info;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);

  for (i=0; i<N; i++) VA[i] = 0;

  if (pid == 0)
  {
    for (i=0; i<N; i++) VA[i] = i;

    destino = 1; tag = 0;
    MPI_Send(&VA[0], N, MPI_INT, destino, tag, MPI_COMM_WORLD);
  }

  else if (pid == 1)
  {
    printf("\n Valor de VA en P1 antes de recibir datos\n\n");
    for (i=0; i<N; i++) printf("%4d", VA[i]);
    printf("\n\n");

    origen = 0; tag = 0;
    MPI_Recv(&VA[0], N, MPI_INT, origen, tag, MPI_COMM_WORLD, &info);

    MPI_Get_count(&info, MPI_INT, &ndat);
    printf(" P1 recibe VA de P%d: tag %d, ndat %d \n\n",
            info.MPI_SOURCE, info.MPI_TAG, ndat);
    for (i=0; i<N; i++) printf("%4d", VA[i]);
    printf("\n\n");
  }

  MPI_Finalize();
  return (0);
}
```

```
/*************************************************************
       probe.c
       ejemplo de uso de la funcion probe de MPI
*************************************************************/

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


int main (int argc, char **argv)
{
  int   pid, npr, origen, destino, tag;
  int   i, longitud, tam;
  int   *VA, *VB;
  MPI_Status    info;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);

  if (pid == 0)
  {
    srand(time(NULL));
    longitud = rand() % 100;
    VA = (int *) malloc (longitud*sizeof(int));
    for (i=0; i<longitud; i++) VA[i] = i;

    printf("\n Valor de VA en P0 antes de enviar los datos\n\n");
    for (i=0; i<longitud; i++) printf("%4d", VA[i]);
    printf("\n\n");

    destino = 1; tag = 0;
    MPI_Send(&VA[0], longitud, MPI_INT, destino, tag, MPI_COMM_WORLD);
    free(VA);
  }

  else if (pid == 1)
  {
    origen = 0;  tag = 0;
    MPI_Probe(origen, tag, MPI_COMM_WORLD, &info);
    MPI_Get_count(&info, MPI_INT, &tam);

    if(tam != MPI_UNDEFINED)
    {
      VB = (int *) malloc(tam*sizeof(int));
      MPI_Recv(&VB[0], tam, MPI_INT, origen, tag, MPI_COMM_WORLD, &info);
    }

    printf("\n Valor de VB en P1 tras recibir los datos\n\n");
    for (i=0; i<tam; i++) printf("%4d", VB[i]);
    printf("\n\n");
    free(VB);
  }

  MPI_Finalize();
  return (0);
}
```

**▪ Ejercicios a realizar en el laboratorio (fase inicial)**

Como complemento de las primeras sesiones de laboratorio, tienes que realizar estos dos ejercicios:

**0.1** En el programa MPI `envio.c`, el proceso P0 genera el vector VA de 10 elementos y se lo envía a P1, que lo recibe e imprime.

Modifica el programa para que P1 devuelva a P0 la suma de los elementos del vector recibido, y este último imprima el resultado.

**0.2** El programa `circu.c` ejecuta en paralelo un bucle en el que se buscan las soluciones de una función lógica de 16 variables, testeando el resultado de la función para todos los posibles valores de entrada. El reparto de las iteraciones del bucle es estático entrelazado, y se imprimen las combinaciones de entrada que hacen que la función valga 1.

Modifica el programa para que P0 imprima el número total de soluciones encontradas.

### 2.1.3 MPI (2): otros modos de envío de mensajes punto a punto; bloqueos en la comunicación.

Tras la finalización de la fase 1 de la aplicación y el análisis de los primeros resultados, ampliaremos algunos conceptos de MPI relativos a la comunicación punto a punto. En concreto:

- Comunicación síncrona: `MPI_Ssend`. Problemas de *deadlock*.
- Comunicación inmediata: solapamiento del cálculo y la comunicación.

  `MPI_Isend`, `MPI_IRecv`, `MPI_Test`, `MPI_Wait`

Para ello utilizaremos los siguientes programas: `dlock1.c`, `dlock2.c`, `dlock3.c`, `dlock3s.c`, `dlock4.c`, `dlock5.c`, `send-dead.c`, `ssend.c`, `isend.c`.

```
/*****************************************************************************
       dlock1.c
       intercambio de dos variables
*****************************************************************************/

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

int main (int argc, char **argv)
{
  int   pid, origen, destino, tag;
  int   A, B, C;
  MPI_Status  info;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &pid);

  if (pid == 0)
  {
    A = 5;

    printf ("\n    >> recibiendo datos de P1  \n");
    origen = 1;  tag = 1;
    MPI_Recv (&B, 1, MPI_INT, origen, tag, MPI_COMM_WORLD, &info);

    printf ("\n    >> enviando datos a P1  \n");
    destino = 1; tag = 0;
    MPI_Send (&A, 1, MPI_INT, destino, tag, MPI_COMM_WORLD);

    C = A + B;
    printf ("\n   C es %d en proc 0 \n\n", C);
  }
  else if (pid == 1)
  {
    B = 6;

    printf ("\n    >> recibiendo datos de P0  \n");
    origen = 0;  tag = 1;
    MPI_Recv (&A, 1, MPI_INT, origen, tag, MPI_COMM_WORLD, &info);

    printf ("\n    >> enviando datos a P0  \n");
    destino = 0; tag = 0;
    MPI_Send (&B, 1, MPI_INT, destino, tag, MPI_COMM_WORLD);

    C = A + B;
    printf ("\n   C es %d en proc 1 \n\n", C);
  }
  MPI_Finalize ();
  return (0);
}
```

**EJERCICIOS: `dlock`**

Analiza las diferentes versiones de este programa, indicando si son correctas o no, razonando tu respuesta.

```
/**************************************************************************
       send-dead.c
       prueba para ver el tamaño del buffer de la funcion send
       el programa se bloquea al enviar un paquete mas grande
**************************************************************************/

#include <stdio.h>
#include "mpi.h"
#define N 100000

int main (int argc, char** argv)
{
  int         pid, kont;                // Identificador del proceso
  int         a[N], b[N], c[N], d[N];
  MPI_Status  status;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &pid);

  for (kont=100; kont<=N; kont=kont+100)
  {
    if (pid == 0)
    {
      MPI_Send (&a[0], kont, MPI_INT, 1, 0, MPI_COMM_WORLD);
      MPI_Recv (&b[0], kont, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
      printf ("emisor %d \n", kont);
    }
    else
    {
      MPI_Send (&c[0], kont, MPI_INT, 0, 0, MPI_COMM_WORLD);
      MPI_Recv (&d[0], kont, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
      printf ("                  receptor %d \n", kont);
    }
  }

  MPI_Finalize ();
  return 0;
} /*  main  */
```

```
/*****************************************************************************
       ssend.c  Ping-pong entre dos procesadores. Comunicacion sincrona ssend
*****************************************************************************/

#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <unistd.h>
#define VUELTAS 4

double calculo ()
{
  double aux;
  sleep (1);
  aux = rand () % 100;

  return (aux);
}


int main (int argc, char** argv)
{
  double      t0, t1, dat= 0.0, dat1, dat_rec;
  int         pid, i;
  MPI_Status  status;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &pid);

  if (pid == 0) t0 = MPI_Wtime ();
  for (i=0; i<VUELTAS; i++)
  {
    if (pid == 0)
    {
      MPI_Ssend (&dat, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
      dat1 = calculo ();

      MPI_Recv (&dat_rec, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status);
      dat = dat1 + dat_rec;
    }
    else
    {
      dat1 = calculo ();

      MPI_Recv (&dat_rec, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
      dat = dat1 + dat_rec;
      MPI_Ssend (&dat, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
  }

  if (pid == 0)
  {
    t1 = MPI_Wtime ();
    printf ("\n   Tiempo de ejecucion = %f s \n", t1-t0);
    printf ("\n   Dat = %1.3f \n\n", dat);
  }

  MPI_Finalize ();
  return (0);
} /*  main  */
```

```
/*****************************************************************************
       isend.c
       Ping-pong entre dos procesadores. Comunicacion inmediata isend
*****************************************************************************/

  ...

  for (i=0; i<VUELTAS; i++)
  {
    if (pid == 0)
    {
      MPI_Isend (&dat, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &request);

      dat1 = calculo ();

      MPI_Recv (&dat_rec, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status);
      dat = dat1 + dat_rec;
      MPI_Wait (&request, &status);
    }
    else
    {
      dat1 = calculo ();

      MPI_Recv (&dat_rec, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

      dat = dat1 + dat_rec;
      if (i != 0) MPI_Wait (&request, &status);
      MPI_Isend (&dat, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,&request);
    }
  }

…
```

### 2.1.4 Jumpshot

Jumpshot es la herramienta de análisis de la ejecución de programas paralelos que se distribuye junto con MPICH. Permite analizar gráficamente ficheros de trazas (tipo "log") obtenidos de la ejecución de programas MPI a los que previamente se les ha añadido una serie de llamadas a MPE para recoger información.

Para generar el fichero log podemos añadir puntos de muestreo en lugares concretos (añadiendo funciones de MPE), o, en casos sencillos, tomar datos de todas las funciones MPI.

```
> mpicc -mpe=mpilog -o prog prog.c
> mpiexec –n xx prog
```

Una vez compilado, lo ejecutamos y obtenemos un fichero de trazas, de tipo clog2. Ahora podemos ejecutar jumpshot para analizar el fichero de trazas obtenido:

```
> jumpshot
```

Jumphot4 trabaja con un formato de tipo slog2, por lo que previamente hay que efectuar una conversión a dicho formato. Esa conversión puede hacerse ya dentro de la aplicación, o ejecutando:

```
> clog2TOslog2 prog.clog2
```

La ventana inicial de jumpshot nos permite escoger el fichero que queremos visualizar. En una nueva ventana aparece un esquema gráfico de la ejecución, en el que las diferentes funciones de MPI se representan con diferentes colores. En el caso de las comunicaciones

punto a punto, una flecha une la emisión y recepción de cada mensaje. Con el botón derecho del ratón podemos obtener datos de las funciones ejecutadas y de los mensajes transmitidos.

NOTA: para poder ejecutar esta aplicación gráfica (u otras) de manera remota:

-- Windows: >> ejecutar previamente `X-Win32` (o una aplicación similar)

-- Linux: >> entrar en la máquina ejecutando `ssh –X cuenta@máquina`

## 2.2 PUZLE SOBRE CONCEPTOS DE PROGRAMACIÓN PARALELA, MPI

De cara a estudiar cómo construir programas paralelos eficientes mediante MPI, y para preparar el camino para el diseño y programación de la aplicación que define el proyecto sobre MPI, hemos dividido las cuestiones más relevantes en tres partes, con las que organizar un puzle.

Cada persona del grupo debe preparar una de la partes del puzle, reunirse con la persona equivalente del resto de grupos para debatir y aclarar cuestiones, y finalmente compartir con el resto de personas del grupo lo aprendido.

Cada parte del puzle conlleva el estudio de las cuestiones que se refieren y la realización, ejecución y comprobación de los resultados de los programas que se proponen.

Tras la introducción general que hemos realizado en el laboratorio viendo cómo utilizar el *cluster* y analizando las funciones de comunicación básicas de MPI, las tres partes del puzle son las siguientes:

1. **Comunicaciones colectivas**

2. **Tipos de datos derivados y comunicadores**

3. **Reparto dinámico de la carga y problemas en las fronteras del reparto de datos.**

### 2.2.1 Comunicaciones colectivas

La comunicación es un aspecto importante en la programación de aplicaciones en sistemas paralelos. Como hemos estudiado, en estos sistemas la comunicación entre procesos se realiza mediante paso de mensajes. En las sesiones de laboratorio hemos visto las funciones básicas de comunicación punto a punto en MPI (`MPI_Send` y `MPI_Recv`), pero existe otro tipo de funciones que permiten una comunicación más eficiente entre procesos, y que simplifican la programación paralela en la mayoría de las aplicaciones. Se trata de las funciones de comunicación colectivas.

Mediante esta actividad deberás entender en qué consiste la comunicación colectiva, qué tipo de funciones existen, y su utilización en la programación en MPI. Para ello, te indicamos unas referencias (puedes encontrar fácilmente muchas más) para poder consultar y preparar una breve exposición del tema para tu grupo.

De cara a asentar los conceptos utilizados, deberás entregar las soluciones de los tres ejercicios que te proponemos, explicando la resolución de los mismos en un breve documento (1 o 2 caras máximo). El código base de los programas está en tu cuenta, en el directorio `templates/puzle`.

> Referencias generales

- www.mpi-forum.org/docs/docs.html
- computing.llnl.gov/tutorials/mpi/
- Pacheco P.: *Parallel Programming with MPI*. Morgan Kaufmann, 1997
- Gropp W., Lusk E., Skjellum A.: *Using MPI. Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1999.

> Referencias sobre comunicaciones colectivas

- Pacheco P.: *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011. Capítulo 3, apartado 4.
- Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J. *MPI: The Complete Reference, Volume 1, The MPI Core*. The MIT Press, 1999. Capítulo 4.

**> Ejercicios de la primera parte del puzle**

**P1.1** Hay que repartir un vector de `N` elementos entre `npr` procesos. Completa el programa serie `P11-distribute0.c`, para que genere el tamaño de cada trozo que hay que repartir del vector y el desplazamiento desde el origen del vector al comienzo de cada trozo, en estos dos casos:

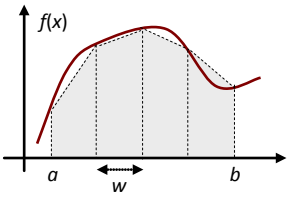
**a.** los posibles restos se añaden al último trozo.

**b.** los posibles restos se añaden uno a uno a diferentes trozos.

Por ejemplo, si ejecutas el programa con estos datos: `N` = 17, `npr` = 5, debe imprimir:

```
Data to distribute: N and npr
17
5

FIRST DISTRIBUTION: the remainder for the last process
  3   0
  3   3
  3   6
  3   9
  5   12

SECOND DISTRIBUTION: the remainder distributed (+1) among the first processes
  4   0
  4   4
  3   8
  3   11
  3   14
```

**P1.2** El programa `P12-inteser.c` calcula el valor de una integral mediante el conocido método de sumar las áreas de *n* trapecios bajo la curva que representa a la función. A mayor valor de *n*, más preciso es el resultado.




Completa el programa `P12-intepar0.c` para realizar esa misma función entre *P* procesos, utilizando funciones de comunicación colectiva. Compara el resultado con el de la versión serie. Por ejemplo, si ejecutas el programa en 4 procesadores con estos datos, el resultado debe ser:

```
Introduce a, b (limits) and n (num. of trap.)
0
10
10000000

  Integral, from 0.0 to 10.0, 10000000 trap.) = 3.869022947101
  Execution time (4 proc.) = 47.474 ms
```

**P1.3** En una ejecución con cuatro procesos, P2 reparte datos del vector `B` (de 16 enteros) de la siguiente manera: a P0: `B[3]`, `B[4]`, `B[5]`; a P1: `B[7]`, `B[8]`; a P2: `B[10]`; y a P3: `B[12]`, `B[13]`, `B[14]`, `B[15]`. Tras ello, cada proceso suma 100 a los elementos recibidos, y, finalmente, se recopilan los datos finales en P2, en las mismas posiciones iniciales del vector `B`.

Completa el programa `P13-scatter-gather0.c` para que realice esa función; al principio, P2 debe inicializar el vector a `B[i] = i`, y, al final, imprimir el nuevo vector `B`. El programa debe imprimir:

```
B in pid=2 after the calculation
   0   1   2 103 104 105   6 107 108   9 110  11 112 113 114 115
```

```
/*****************************************************************************
      P11-distribute0.c
      >>> TO DO: MODIFY AND COMPLETE <<<
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
  int    N, Nloc, npr, remainder, i;

  printf ("\n Data to distribute:  N and npr\n");
  scanf ("%d %d" , &N, &npr);


// FIRST DISTRIBUTION:  the remainder for the last process
//     >>> TO DO: CALCULATE SIZE AND SHIFTS FOR THE FIRST DISTRIBUTION <<<




  printf ("\n FIRST DISTRIBUTION: the remainder for the last process \n");
  for (i=0; i<npr; i++) printf ("\n  %d   %d", size[i], shift[i]);

// SECOND DISTRIBUTION: the remainder distributed (+1) among the first processes
//     >>> TO DO: CALCULATE SIZE AND SHIFTS FOR THE SECOND DISTRIBUTION <<<




  printf ("\n\n SECOND DISTRIBUTION: the remainder distributed (+1) among the first
           processes \n");
  for (i=0; i<npr; i++) printf ("\n  %d   %d", size[i], shift[i]);
  printf ("\n");
  return 0;
}
```

```
/*****************************************************************
       P12-inteser.c
       Integral of a function by sums of areas of trapezoids
******************************************************************/

#include <stdio.h>
#include <sys/time.h>

struct timeval   t0, t1;
double texec;

void   Read_data (double* a_ptr, double* b_ptr, int* n_ptr);
double Integrate (double a, double b, int n, double w);
double f (double x);

int main (int argc, char** argv)
{
  double      a, b, w;
  int         n;
  double      resul;          // Result for the integral

  Read_data (&a, &b, &n);
  w = (b-a) / n;

// Integral calculation
  gettimeofday (&t0,0);
  resul = Integrate (a, b, n, w);

// Print results
  gettimeofday (&t1,0);
  texec = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec-t0.tv_usec) / 1e6;
  printf ("\n  Integral (= ln x+1 + atan x), from %1.1f to %1.1f, %d trap. = %1.12f\n",
       a,b,n,resul);
  printf ("  Execution time (serie) = %1.3f ms \n\n", texec*1000);
  return (0);
} /*  main  */


// FUNCION Read_data
void Read_data (double* a_ptr, double* b_ptr, int* n_ptr)
{
  float a, b;

  printf ("\n  Introduce a, b (limits) and n (num. of trap.)  \n");
  scanf ("%f %f %d", &a, &b, n_ptr);
  (*a_ptr) = (double)(a);
  (*b_ptr) = (double)(b);
} /* Read_data */


// FUNCTION Integrate: local calculation of the integral
double Integrate (double a, double b, int n, double w) {
  double resul, x;
  int    i;

// Integral calculation
  resul = (f(a) + f(b)) / 2.0;
  x = a;

  for (i=1; i<n; i++) {
    x = x + w;
    resul = resul + f(x);
  }
  resul = resul * w;
  return (resul);
} /*  Integrate  */


// FUNCTION f: Function to integrate
double f (double x) {
  double y;

  y = 1.0 / (x + 1.0) + 1.0 / (x*x + 1.0);
  return (y);
} /* f function */
```

```
/*****************************************************************
       P12-intepar0.c
       Integral of a function by sums of areas of trapezoids
       Using broadcast for data sending and Reduce for data receiving
       >>> TO DO: MODIFY AND COMPLETE <<<
*****************************************************************/

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

double   t0, t1;

void   Read_data (double* a_ptr, double* b_ptr, int* n_ptr);
double Integrate (double a_loc, double b_loc, int n_loc, double w);
double f (double x);

int main (int argc, char** argv)
{
  int         pid, npr;
  double      a, b, w, a_loc, b_loc;
  int         n, n_loc, remainder;
  double      resul, resul_loc;             // Result of the integral: global and local

// MPI Initializations
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &pid);
  MPI_Comm_size (MPI_COMM_WORLD, &npr);

// Reading parameters and distributing them to all processes
//      >>> TO DO <<<

  read_data (&a, &b, &n);

// Dividing the calculation among the processes
  w = (b-a) / n;
  n_loc = n / npr;
  remainder = n % npr;

  if (pid < remainder) n_loc = n_loc + 1;

  a_loc = a + pid * n_loc * w;
  if (pid >= remainder) a_loc = a_loc + remainder * w;
  b_loc = a_loc + n_loc * w;

// Local calculation of the integral
  resul_loc = Integrate (a_loc, b_loc, n_loc, w);

// Adding the partial results
//      >>> TO DO: MERGE ALL THE PARTIAL RESULTS <<<


// Print results
  if (pid == 0)
  {
    t1 = MPI_Wtime();
    printf ("\n  Integral (= ln x+1 + atan x), from %1.1f to %1.1f, %d trap.) =
        %1.12f\n",a,b,n,resul);
    printf ("  Execution time (%d proc.) = %1.3f ms \n\n", npr, (t1-t0)*1000);
  }

  MPI_Finalize ();
  return (0);
} /*  main  */


// FUNCION Read_data
void Read_data (double* a_ptr, double* b_ptr, int* n_ptr)
{
  float a, b;

  printf ("\n  Introduce a, b (limits) and n (num. of trap.)  \n");
  scanf ("%f %f %d", &a, &b, n_ptr);

  (*a_ptr) = (double)(a);
  (*b_ptr) = (double)(b);
} /* Read_data */
```

```
// FUNCTION Integrate: local calculation of the integral
double Integrate (double a_loc, double b_loc, int n_loc, double w)
{
  double resul_loc, x;
  int    i;

  // Integral calculation
  resul_loc = (f(a_loc) + f(b_loc)) / 2.0;
  x = a_loc;

  for (i=1; i<n_loc; i++)
  {
    x = x + w;
    resul_loc = resul_loc + f(x);
  }
  resul_loc = resul_loc * w;

  return (resul_loc);
} /*  Integrate  */


// FUNCTION f: function to integrate
double f (double x)
{
  double y;

  y = 1.0 / (x + 1.0) + 1.0 / (x*x + 1.0);
  return (y);
} /* f function */
```

```
/*********************************************
       P13-scatter-gather0.c
       [4 processes]
       >>> TO DO: MODIFY AND COMPLETE <<<
*********************************************/

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

int main (int argc, char** argv)
{
  int  pid, npr, i;
  int  B[16],  Bloc[16];

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &pid);
  MPI_Comm_size (MPI_COMM_WORLD, &npr);

  if (pid == 2) for (i=0; i<16; i++) B[i] = i;

// Scattering of B from pid=2
//      >>> TO DO <<<


// Local calculation
//      >>> TO DO: INCREMENT WITH 100 <<<


// Gathering of Bloc in pid=2
//      >>> TO DO <<<


// Print results
  if (pid == 2)
  {
     printf ("\n B in pid=2 after the calculation \n");
     for (i=0; i<16; i++) printf ("%4d", B[i]);
     printf ("\n\n");
  }
  MPI_Finalize ();
  return (0);

} /*  main  */
```

### 2.2.2 Tipos de datos derivados / Comunicadores

En los ejemplos que hemos realizado en el laboratorio se han intercambiado datos con una estructura muy simple (enteros, flotantes, etc. consecutivos), pero en muchas ocasiones es necesario disponer de mayor flexibilidad en la definición de los datos que se desean enviar y recibir (por ejemplo, datos no consecutivos en memoria, de tipos diferentes…). Además, el coste de enviar múltiples datos en varios mensajes es mayor que el coste de enviar la misma cantidad de datos en un único mensaje. Por ello, MPI ofrece diferentes alternativas para el envío de datos en diferentes "formatos", que permiten reducir las latencias de la comunicación entre los procesos.

Por otra parte, otro aspecto importante en la comunicación entre procesos es la agrupación de dichos procesos en comunicadores diferentes al `MPI_COMM_WORLD` inicial. Esto permite que cada proceso se pueda identificar de maneras diferentes, según al grupo o comunicador con el que quiera trabajar.

Mediante esta actividad deberás comprender qué alternativas ofrece MPI para estructurar los datos de los mensajes y cómo se utilizan, así como la manera de definir y gestionar grupos de procesos diferentes al inicial. Para ello, te indicamos unas referencias (puedes encontrar fácilmente muchas más) para poder consultar y preparar una breve exposición del tema para tu grupo.

De cara a asentar los conceptos utilizados, deberás entregar las soluciones de los tres ejercicios que te proponemos, explicando la resolución de los mismos en un breve documento (1 o 2 caras máximo). El código base de los programas está en tu cuenta, en el directorio `templates/puzle`.

> Referencias generales

- www.mpi-forum.org/docs/docs.html
- computing.llnl.gov/tutorials/mpi/
- Pacheco P.: *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.
- Gropp W., Lusk E., Skjellum A.: *Using MPI. Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1999.

> Referencias sobre tipos de datos derivados y comunicadores

- Pacheco P.: *Parallel Programming with MPI*. Morgan Kaufmann, 1997. Capítulo 6, apartados 2, 3 y 5; capítulo 7, apartados 3-5.
- Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J.: *MPI: The Complete Reference, Volume 1, The MPI Core*. The MIT Press, 1999. Capítulo 3, apartados 4, 5 y 12; capítulo 5, apartados 1, 2 y 4.

**> Ejercicios de la segunda parte del puzle**

**P2.1** En un programa MPI, el proceso P3 tiene una matriz MAT de 5x5 enteros, de la que tiene que enviar la diagonal al resto de procesos. Completa el programa `P21-diagonal0.c` para que ejecute es operación en estos dos casos:

- la diagonal se recibe en los otros procesos como un simple vector, y se calcula e imprime la suma de los elementos recibidos.
- la diagonal se recibe sustituyendo a la diagonal de la matriz local MAT.

Ejecuta el programa con 4 procesos. Debe imprimir:

```
The sum of the received data in P1 is: 40
The new diagonal of MAT in P0 is:  0  4   8  12  16
```

**P2.2** Completa el programa `P22-pack0.c`, para que P0 envíe a P1 tres elementos en un solo mensaje: una matriz A de 100x100 enteros, un vector B de 2.000 flotantes, y C, un *double*. Para ello, P0 empaqueta los datos y envía el paquete a P1; por su parte, P1 recibe el mensaje y desempaqueta los datos.

Ejecuta el programa con 2 procesos. Debe imprimir:

```
Received in P1: A[10][10] = 100   B[33] = 13.2   C = 2.2
```

**P2.3** Una aplicación paralela se ejecuta en 8 procesos. En un momento dado, necesitamos construir dos grupos diferentes, de 4 procesos cada uno: los procesos 0 a 3 por un lado, y los procesos 4 a 7 por otro. En cada grupo, los procesos tendrán un nuevo identificador.

Tras ello, en cada grupo se efectúa una operación de recogida de datos, de tal manera que, partiendo de vectores V de 5 enteros en cada proceso (inicializados al valor del `pid` del proceso), al final todos ellos dispongan del vector `W` de 20 elementos, formado por la concatenación de los vectores V de los 4 procesos de cada grupo.

Completa el programa `P23-groups0.c` para que realice esa función. Ejecuta el programa con 8 procesos; debe imprimir:

```
W(0,5,10,15) data in pid 1 [pid2 1, group 0]: 0 1 2 3
W(0,5,10,15) data in pid 5 [pid2 1, group 1]: 4 5 6 7
```

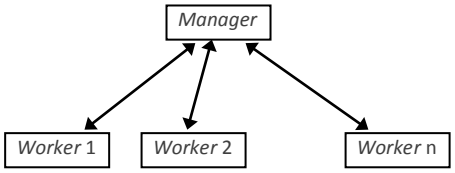
```
/*************************************************************
       P21-diagonal0.c
       MPI program for sending a diagonal of a matrix
       [4 processes]
       >>> TO DO: MODIFY AND COMPLETE <<<
*************************************************************/

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

#define N 5

int main (int argc, char **argv)
{
  int   i, j, sum;
  int   MAT[N][N], buf[N];
  int   pid;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &pid);

// Initialisation of the matrices in all the processes
  for (i=0; i<N; i++)
  for (j=0; j<N; j++)  MAT[i][j] = pid*i + j;

// Defining the diagonal type
//     >>> TO DO <<<



// 1. Sending the diagonal of the matrix MAT in P3 to P0, P1 and P2
// It is received as a vector in a buffer
//     >>> TO DO <<<



// In order to check, P1 adds the received data and print the result

   sleep (2);

// 2. Sending the diagonal of the matrix MAT in P3 to P0, P1 and P2
// It is received in the MAT matrix in each process
//     >>> TO DO <<<



// In order to check, P0 prints the new diagonal of the MAT matrix

  MPI_Finalize ();
  return 0;
}
```

```
/*************************************************************
       P22-pack0.c
       MPI program using pack/unpack
       [2 processes]
       >>> TO DO: MODIFY AND COMPLETE <<<
*************************************************************/

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

#define sizeA 100
#define sizeB 2000

#define sizebuf 50000

int main (int argc, char **argv)
{
  int   pid, i, j;
  int   A[sizeA][sizeA];
  float B[sizeB];
  double C;

  char buf[sizebuf];
  int pos = 0;
  MPI_Status info;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &pid);

// P1 initialises the data
//    >>> TO DO <<<

    for (i=0; i<sizeA; i++)
    for (j=0; j<sizeA; j++) A[i][j] = i*j;
    for (i=0; i<sizeB; i++) B[i] = (float)i*0.4;
    C = 2.2;

// Packing the data in P1 and sending the packet to P2
//     >>> TO DO <<<


// Receiving the packet and unpacking the data in P2
//     >>> TO DO <<<


    printf ("\n Received in P2: A[10][10] = %d  B[33] = %3.1f  C = %3.1f\n\n",
            A[10][10], B[33], C);

  MPI_Finalize ();
  return 0;
}
```

```
/*****************************************************************
       P23-groups0.c
       MPI program using communicators (Split function)
       [8 processes]
       >>> TO DO: MODIFY AND COMPLETE <<<
*****************************************************************/

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

#define N 20

int main (int argc, char* argv[])
{
  int       npr, pid, i;
  int       V[N/4], W[N];
  MPI_Status info;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &npr);
  MPI_Comm_rank (MPI_COMM_WORLD, &pid);

  for (i=0; i<N/4; i++) V[i] = pid;

// Groups and communicators creation
//      >>> TO DO <<<



// Sending from pid=0 in each group to the others processes
//      >>> TO DO <<<



// The process 0 in each group print  W(0,5,10,15) values and
// its pid in the global communicator and in the local one
//      >>> TO MODIFY <<<



  MPI_Finalize ();
  return (0);
}  /* main */
```

### 2.2.3 Reparto dinámico de carga / El problema de la "frontera"

Para lograr un buen reparto de la carga de trabajo, cuando no conocemos a priori el tiempo de ejecución de las tareas, es necesario efectuar un reparto dinámico de las mismas, es decir, en tiempo de ejecución.

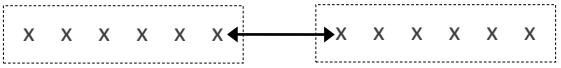



En esos casos, es habitual utilizar el conocido modelo de programación "*manager/worker*", en el que uno de los procesos, el *manager*, se encarga, por un lado, de repartir tareas bajo demanda y, por otro, de recoger resultados. El resto de procesos, los *workers*, solicitan tareas, las ejecutan, y devuelven resultados, hasta completar entre todos la carga inicial de trabajo.

Este puede ser el algoritmo general:

```
si manager
    mientras haya trabajo
        espera peticiones
        envía trabajo
        recoge resultados / envía trabajo
    si no hay más trabajo
        recoge resultados / envía código fin de trabajo
si worker
    mientras no fin de trabajo
        pide trabajo
        ejecuta trabajo
        envía resultados / pide trabajo
```

Por otro lado, otro problema típico de las aplicaciones paralelas es el de las "fronteras" en el reparto de los datos. Por ejemplo, al repartir un vector el último elemento de un trozo y el primero del siguiente pasan de ser consecutivos en memoria a estar en diferentes procesadores, por lo que si hay alguna relación entre ellos es necesario efectuar operaciones explicitas de comunicación.




Mediante esta actividad deberás de comprender los dos problemas planteados: el reparto dinámico de tareas mediante un modelo de programación *manager/worker* y el intercambio de datos en las fronteras del reparto, y preparar una breve exposición de estas cuestiones para tu grupo.

De cara a asentar los conceptos utilizados, deberás entregar las soluciones de los dos ejercicios que te proponemos, explicando la resolución de los mismos en un breve documento (1 o 2 caras máximo). El código base de los programas está en tu cuenta, en el directorio `templates/puzle`.

**> Ejercicios de la tercera parte del puzle**

**P3.1** El programa `P31-collatzser.c` aplica una función basada en el algoritmo de Collatz a números enteros desde 1 a 320, con una carga de trabajo proporcional al número de iteraciones necesarias para que los números converjan a 1.

Hay que hacer dos versiones paralelas del programa. En la primera, las tareas (procesar los números) se reparten entre todos los procesos de modo estático consecutivo, procesando cada uno de ellos 320/`npr` números consecutivos. En la segunda, el reparto de tareas es dinámico, bajo demanda. Uno de los procesos (P0, por ejemplo) funciona como *manager* y reparte a los restantes procesos (*workers*) números a procesar, uno a uno, cuando se lo solicitan. Cada *worker* procesa ese número y devuelve al *manager* el número de iteraciones que ha necesitado para converger. Si quedan números por analizar, se le envía una nueva tarea, hasta terminar de analizar entre todos los *workers* todos los números. El proceso *manager* debe controlar cuántos números ha procesado cada *worker*, y el número que ha necesitado más iteraciones para converger.

Una vez verificados ambos programas, ejecuta la versión estática con 2, 4, 8, 16, 32 y 64 procesos, y la dinámica con 1+1, 1+2, 1+4, 1+8, 1+16, 1+32 y 1+63 procesos. Obtén los tiempos de ejecución y calcula los *speed-ups* y las eficiencias obtenidas. Dibuja ambos parámetros en función del número de procesos y explica los resultados.

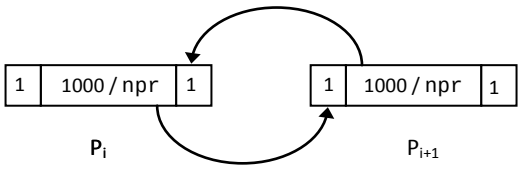
**P3.2** En una determinada aplicación (`P32-convoser.c`) se procesa un vector de $N$ = 1.002 elementos al que se aplica una operación de convolución de manera iterativa

```
for (i=1; i<N-1; i++) Aux[i] = (A[i-1] + 2*A[i] + A[i+1]) / 4
for (i=1; i<N-1; i++) A[i] = Aux[i]
```

hasta que el valor máximo de los elementos del vector sea menor al 60% del valor máximo inicial. El primer y último elemento del vector no se procesan.

El programa se va a ejecutar con un número de procesos divisor de 1.000. Escribe una versión paralela del programa, en el que:

a. P0 reparte los 1.000 elementos del vector que hay que procesar, en trozos de 1000/`npr` elementos: el primero lo procesará él mismo, el segundo P1…;
b. cada proceso procesa el trozo de vector que le ha tocado y calcula el máximo local;
c. los procesos envían su máximo local a P0, que calcula el máximo global y, tras ello, avisa a todos los procesos si hay que efectuar una nueva iteración de convolución o si la operación ha terminado;
d. al acabar, los procesos envían a P0 su trozo de vector procesado, para que éste lo reconstruya.




Cada procesador va a necesitar, para procesar su trozo, los datos que le han correspondido y 2 más, el anterior al primer elemento y el posterior al último, que estarán en el procesador anterior y en el siguiente. Por tanto, te sugerimos que cada proceso utilice un buffer de 1 + 1000/`npr` + 1 elementos, y que previo a la operación de convolución se intercambien con `pid`-1 y `pid`+1 los datos que necesiten (iteración a iteración).

```
/*************************************************************
       P31-collatzser.c
*************************************************************/

#include <stdio.h>
#include <sys/time.h>
#include <stdlib.h>

#define NUMBER 320


int collatz (int  n)
{
  int steps=0;

  while (n > 1)
   {
     if ((n % 2) == 1) n = 3*n + 1;
     else n = n/2;

     steps++;
   }
  return (steps);
}


void work (int steps)
{
  usleep (2000*steps);
}



main (int argc, char *argv[])
{

  int n, steps, total_steps=0, max_steps=0, n_max_steps;
  struct timeval t0,t1;
  double tex;

  printf ("\n COLLATZ (serial): 1 - %d\n\n", NUMBER);

  gettimeofday (&t0, 0);

  for (n=1; n<=NUMBER; n++)
  {
    steps = collatz(n);
    work (steps);
    total_steps += steps;
    if (steps > max_steps) {n_max_steps = n; max_steps = steps;}
  }

  gettimeofday (&t1, 0);

  tex = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec) / 1e6;
  printf (">>Total steps: %d >>Num_max_steps: %d (%d steps) >>Execution time:
       %1.3f ms\n\n", total_steps, n_max_steps, max_steps, tex*1000);
}
```

```
/*************************************************************
       P32-convoser.c
*************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <values.h>

#define NELEM 1002


main (int argc, char*argv[])
{
  int i, end, steps=0;
  float lim, max, max_ini;

  float A[NELEM], Aux[NELEM];


// initial values for A
  max_ini = MINFLOAT;
  A[0] = A[NELEM-1] = 150;
  for (i=1; i<NELEM-1; i++)
  {
    A[i] = (rand() % 1000) / 3.0;
    if (A[i] > max_ini) max_ini = A[i];
  }


// limit to converge
  lim = 0.6 * max_ini;

  printf("\n > Initial state (last 10 elements of A)\n");
  for (i=1; i<11; i++) printf("A[%d]: %.3f\n", NELEM-i, A[NELEM-i]);
  printf ("\n Initial max: %.3f\n", max_ini);


// convolution phase
  end = 0;
  while (!end)
  {
    max = MINFLOAT;

  // convolution
    for (i=1; i<NELEM-1; i++)
    {
      Aux[i] = (A[i-1] + 2*A[i] + A[i+1]) / 4;
      if (Aux[i] > max) max = Aux[i];
    }
    for (i=1; i<NELEM-1; i++) A[i] = Aux[i];

    steps++;

  // convergence control
    if (max < lim) end = 1;
  }

  printf ("\n\n Convolution steps: %d   Final max: %.3f\n\n", steps, max);
  printf (" > Final state (last 10 elements of A)\n");
  for (i=1; i<11; i++) printf("A[%d]: %.3f\n", NELEM-i, A[NELEM-i]);
}
```

## 2.3 EJERCICIOS COMPLEMENTARIOS

Aquí tienes unos ejercicios por si quieres probar algo más tras la puesta en común del puzle.

**C1** En una aplicación paralela, el proceso `pid` = 0 recibe de los otros procesos mensajes de diferente longitud con datos (enteros), que procesa ejecutando la función `PROC(*dat, tam)`, donde `*dat` es la dirección de comienzo de los datos y `tam` su tamaño. Al recibir un mensaje, se responde con un mensaje corto (un entero) para confirmar la recepción y se procesan los datos. Escribe el código que ejecutará P0 para realizar esta tarea si:

- hay que recibir y procesar los mensajes en orden estricto de `pid` (P1, luego P2…).
- queremos recibir y procesar los mensajes en el orden en que llegan.

**C2** En una aplicación paralela, el proceso P0 envía una matriz `M[N][N]` a P1. P1 no conoce el tamaño de la matriz que va a recibir, por lo que primero mira en el buzón, y con esa información reserva espacio para la matriz y a continuación la recibe. Escribe el trozo de código MPI que realiza esa función. Ejecuta un caso concreto y comprueba que el resultado es el esperado.

**C3** En una ejecución en paralelo, el procesador P3 va a recibir un mensaje del procesador P1, pero no conoce ni el tamaño de los datos ni el `tag` del mensaje. Si el `tag` es 0, entonces el mensaje llevará un vector de 100 enteros; si el `tag` es 1, el mensaje llevará solamente un flotante. Escribe el código correspondiente para que envíen y reciban correctamente esos mensajes entre P1 y P3. Ejecuta un caso concreto y comprueba que es correcto.

**C4** En un momento dado de la ejecución de una aplicación en paralelo, cada proceso dispone de una matriz A de 10×10 enteros, y todos ellos necesitan, para continuar con la ejecución, la matriz suma de todas esas matrices. Escribe el código necesario para esa operación, utilizando funciones de comunicación colectiva. Como prueba, ejecuta el programa con 8 procesos, inicializa las matrices al valor del `pid` de cada proceso, y haz que un par de procesos (P2 y P4 por ejemplo) impriman el resultado.

**C5** El programa `matvecser.c` efectúa el siguiente cálculo con matrices y vectores:

```
C[N] = A[N][N] * B[N]        todas las variables son double
D[N] = A[N][N] * C[N]
PE = ∑(C[i]· D[i])
```

Al principio se pide el tamaño de los vectores, `N`, para reservar memoria para los mismos.

Escribe y ejecuta una versión paralela del programa serie. La inicialización de los datos previa al cálculo se realiza en un solo procesador (P0); al final, `C`, `D` y `PE` tienen que quedar en P0. La versión paralela debe permitir un reparto de datos de tamaño variable (funciones _v), es decir, que debe funcionar para cualquier `N` y para cualquier número de procesos. Previo a escribir código, analiza las operaciones que se ejecutan, y decide con qué datos va a trabajar cada proceso y cómo quieres repartirlos, para que puedas efectuar correctamente las reservas de memoria en cada proceso y las funciones de comunicación. Comprueba los resultados comparándolos con los de la versión serie.

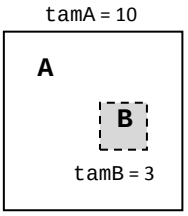
**C6** En un programa MPI, el proceso P0 tiene una matriz A de 10x10 enteros, de la que tiene que enviar al resto de procesos los elementos de la periferia (primera y última fila y primera y última columna), quienes copian esos datos en las posiciones correspondientes de sus matrices. Define los tipos necesarios, empaqueta las dos filas y las dos columnas, y envía el paquete a todos los procesos. Como comprobación, P1 imprime la nueva matriz.

**C7** En una ejecución en paralelo, el proceso P0 tiene una matriz A de 10x10 enteros, de la que a menudo debe enviar a P1, en un solo mensaje, submatrices B (trozos 2D) de tamaño 3x3.

Escribe un programa paralelo que realice esa función. Para ello: **(a)** define un tipo de datos derivado adecuado para esa estructura; y **(b)** envía a P1 la correspondiente matriz B de 3x3 que comienza en el elemento (2,5) de la matriz A.

Inicializa la matriz A a estos valores:

```
for (i=0; i<tamA; i++)
for (j=0; j<tamA; j++)
    A[i][j] = i + j;
```




Imprime la matriz recibida en P1, que debe ser:

```
7     8     9
8     9     10
9     10    11
```

**C8** Queremos enviar una matriz A de P0 a P1, de tal manera que P1 se quede con la matriz A traspuesta. Para ello:

1 Define el tipo columna.

2 P0 envía las columnas de A una a una a P1.

3. P1 recibe las columnas y las guarda como filas de una matriz, que terminará siendo la traspuesta de A.

Como verificación, ejecuta el caso de una matriz pequeña de 4x4, inicialízala en P0 a números aleatorios y haz que ambos procesos impriman la matriz.

¿Se puede hacer esa operación con un solo envío? ¿Cómo?

```
/*************************************************************************
        matvecser.c
         C() = A()() x B()
        D() = A()() x C()
        PE = sum (C() . D())
*************************************************************************/

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  int    N, i, j;
  double PE = 0.0;
  double *Am, *B, *C, *D;

  printf("\n Vector lengths  (<1000) = ");
  scanf("%d" , &N);

/*   memory allocation */
  Am = (double *) malloc (N*N * sizeof(double));

  B = (double *) malloc (N * sizeof(double));
  C = (double *) malloc (N * sizeof(double));
  D = (double *) malloc (N * sizeof(double));

/* initial values  */
  for(i=0; i<N; i++)
  {
    for(j=0; j<N; j++) Am[i*N+j] = (double) (N-i)*0.1/N;
    B[i] = (double) i*0.05/N;
  }

/* calculus */
  for (i=0; i<N; i++)
  {
    C[i] = 0.0;
    for (j=0; j<N; j++) C[i] = C[i] + Am[i*N+j] * B[j];
  }

  PE = 0.0;
  for (i=0; i<N; i++)
  {
    D[i] = 0.0;
    for (j=0; j<N; j++) D[i] = D[i] + Am[i*N+j] * C[j];
    PE = PE + D[i]*C[i];
  }

/* results */
  printf("\n\n   PE = %1.3f \n", PE);
  printf("   D[0] = %1.3f, D[N/2] = %1.3f, D[N-1)] = %1.3f\n\n", D[0],D[N/2],D[N-1]);

  free(Am);
  free(B);
  free(C);
  free(D);
  return (0);
}
```

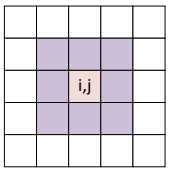
# 3. APLICACIÓN A PARALELIZAR

Tras la fase de aprendizaje desarrollada en formato "puzle" disponemos ya de las herramientas y estrategias necesarias para afrontar con éxito la paralelización de una determinada aplicación. Como vimos en la presentación del proyecto, se trata de una aplicación que simula el comportamiento térmico de una tarjeta de circuitos impresos, buscando la mejor distribución de los chips en la misma, aquella que minimice la temperatura media final de la tarjeta.

## 3.1 DESCRIPCIÓN DE LA APLICACIÓN

El programa `heat_s.c` es la versión serie de la aplicación que hay que paralelizar. Se trata de un caso concreto de resolución de ecuaciones en diferencias parciales (tipo Poisson), que son muy habituales en muchas aplicaciones técnico-científicas.

En nuestro caso, partimos de la definición de una tarjeta en la que se colocan varios chips, cada uno de los cuales inyecta una determinada cantidad de calor. Este calor se va a distribuir por toda la tarjeta, hasta llegar a una situación estacionaria. Para calcular la temperatura media se divide la tarjeta en una rejilla 2D de puntos, y la temperatura de cada punto se va modificando, de manera iterativa, en función de su temperatura y de la de sus vecinos, de acuerdo a la siguiente función:




$$\mathbf{T^1_{i,j}} = T^0_{i,j} + 0{,}1 \times [\Sigma T^0 \text{ de los 8 vecinos} - 8 \times T^0_{i,j}]$$

Tras calcular, de acuerdo a la expresión anterior, la nueva temperatura de cada punto (`i, j`) de la rejilla a partir de las temperaturas anteriores, se inyecta calor en los puntos que ocupan los chips y se disipa calor en posiciones concretas de la tarjeta, que están ventiladas. El proceso de "actualización" de calor y de "difusión" del mismo se repite en toda la rejilla hasta que la temperatura media se estabilice o se haya efectuado un número de iteraciones máximo prefijado. Estas dos operaciones se realizan respectivamente en las funciones `difussion` y `thermal_update`.

**>> thermal_update**

```
// heat injection at chip positions
 for (i=1; i<NROW-1; i++)
 for (j=1; j<NCOL-1; j++)
   if (grid_chips[i*NCOL+j] > grid[i*NCOL+j])
     grid[i*NCOL+j] += 0.05 * (grid_chips[i*NCOL+j] - grid[i*NCOL+j]);

 // air cooling at the middle of the card
 for (i=1; i<NROW-1; i++)
 for (j=0.45*(NCOL-2)+1; j<0.55*(NCOL-2)+1; j++)
     grid[i*NCOL+j] -= 0.01 * (grid[i*NCOL+j] - param.t_ext);
```

**>> diffusion**

```
while (end == 0)
{
  niter++;
  Tmean = 0.0;

  // heat injection and air cooling
  thermal_update (param, grid, grid_chips);

  // thermal diffusion
  for (i=1; i<NROW-1; i++)
  for (j=1; j<NCOL-1; j++)
  {
    T = grid[i*NCOL+j] +
      0.10 * (grid[(i+1)*NCOL+j] + grid[(i-1)*NCOL+j] + grid[i*NCOL+(j+1)] +
          grid[i*NCOL+(j-1)] + grid[(i+1)*NCOL+j+1] + grid[(i-1)*NCOL+j+1] +
          grid[(i+1)*NCOL+(j-1)] + grid[(i-1)*NCOL+(j-1)]
          - 8*grid[i*NCOL+j]);

    grid_aux[i*NCOL+j] = T;
    Tmean += T;
  }

  //new values for the grid
  for (i=1; i<NROW-1; i++)
  for (j=1; j<NCOL-1; j++)
    grid[i*NCOL+j] = grid_aux[i*NCOL+j];

  // convergence every 10 iterations
  if (niter % 10 == 0)
  {
    Tmean = Tmean / ((NCOL-2)*(NROW-2));
    if ((fabs(Tmean - Tmean0) < param.t_delta) || (niter > param.max_iter))
         end = 1;
    else Tmean0 = Tmean;
  }
} // end while
```

Los puntos de la rejilla 2D (`grid`) que representan la tarjeta se inicializan a una temperatura ambiente prefijada, que se va modificando en función de la temperatura de los chips. En todo caso, los puntos que representan los bordes horizontales y verticales de la tarjeta no se procesan, por lo que mantienen siempre su temperatura inicial.

Una matriz 2D del mismo tamaño (`grid_chips`) representa las temperaturas de los puntos que ocupan los chips en la tarjeta, puntos en los que se inyecta calor. Esta matriz se inicializa a partir de un fichero de entrada que define las posiciones, tamaños, temperaturas, etc. de los chips de la tarjeta. En sentido contrario, una franja vertical central de la tarjeta está ventilada, por lo que en esos puntos se reduce la temperatura tras cada iteración. Ambas operaciones, inyección de calor y ventilación, se reflejan en la función `thermal_update`.

El algoritmo de difusión del calor utiliza dos matrices, una con las temperaturas actuales en cada punto (`grid`) y otra con los nuevos valores que se están calculando en esa iteración (`grid_aux`). Al final de la iteración, se vuelca una matriz en la otra.

La actualización de la temperatura media de la tarjeta, `Tmean`, se puede hacer en cada iteración o tras varias iteraciones; en este caso, se hace cada 10 iteraciones. La variable

Tmean0 representa la anterior temperatura media (inicialmente, la temperatura ambiente); si la diferencia entre la actual temperatura media y la anterior es menor que un cierto valor predeterminado, finalizamos la simulación.

El programa principal es sencillo:

```
{
  ...
  read_data (argv[1], &param, &chips, &chip_coord);
  ...

  // loop to process chip configurations
  for (conf=0; conf<param.nconf; conf++)
  {
    gettimeofday (&t0, 0);

    // initial values for grids
    init_grid_chips (conf, param, chips, chip_coord, grid_chips);
    init_grids (param, grid, grid_aux);

    // main loop: thermal injection/disipation until convergence (t_delta or max_iter)
    diffusion (param, grid, grid_chips, grid_aux);

    // writing configuration results
    gettimeofday (&t1, 0);
    tej[conf] = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec)/1e6;
    results_conf (conf, param, grid, grid_chips, &BT);
  }

  // writing best configuration results
  results (param, &BT, argv[1]);
  for (conf=0; conf<param.nconf; conf++) tsim += tej[conf];
  printf ("   > Time (serial): %1.3f s \n\n", tsim);
}
```

La función read_data lee el fichero con las diferentes configuraciones de la tarjeta que hay que simular. Los datos se guardan en las siguientes variables:

>> param: un *struct* de tipo info_param, con los parámetros generales de la simulación (primera línea del fichero de entrada)

```
struct info_param {
  int    nconf, nchip, max_iter, scale;
  float  t_ext, tmax_chip, t_delta;
};
```

| | |
|---|---|
| nconf: | número de configuraciones diferentes que hay que simular; cada configuración está compuesta por los mismos chips, pero en diferentes posiciones de la tarjeta. |
| nchip: | número de chips que tiene la tarjeta. |
| max_iter: | criterio de finalización de la simulación: número máximo de iteraciones. |
| scale: | factor de escala de la simulación (de 1 a 12); el valor 1 representa una rejilla de 200x100 puntos, que usaremos para las pruebas durante el desarrollo de la aplicación; un valor 10 representa una rejilla de 2000 x1000 puntos, y es el que utilizaremos para la obtención de resultados, una vez programada la aplicación. |
| t_ext: | temperatura ambiente a la que se inicializa la rejilla de puntos. |
| tmax_chip: | temperatura máxima de los chips. |
| t_delta: | criterio de finalización: diferencias de temperatura media menores que ese valor. |

>> `chips`: un *struct* de tipo `info_chips`, con las definiciones de los chips de la tarjeta: tamaño (`h`, `w`) y temperatura (`t_chip`).

```
struct info_chips {
  int    h, w;
  float  t_chip;
};
```

>> `chip_coord`: una matriz de tamaño (`nconf` × 2×`nchip`), con una línea por cada configuración a simular, en la se indican las posiciones de los chips de esa configuración en la tarjeta (coordenadas `x` e `y` del vértice superior izquierdo).

**▪ Definición de la tarjeta**

El **fichero de entrada** que define la tarjeta, y de donde se leen los datos de partida, tiene la siguiente estructura (es un ejemplo):

| | |
|---|---|
| 3 4 20.0 160.0 0.01 10000 10 | **1º bloque de datos** (`param`), parámetros generales de la simulación: nº de configuraciones a simular (3); nº de chips (4); temperatura ambiente (20.0); temperatura máxima de un chip (160.0); criterios de convergencia: temperatura (0.01) y nº máximo de iteraciones (10000); factor de escala (10). |
| 40 40 100.0<br>50 20 160.0<br>30 60 120.0<br>20 20 80.0 | **2º bloque de datos** (`chips`), definición de los chips de la tarjeta, una línea por cada chip: por ejemplo, 4 chips, el primero de tamaño (40x40) y temperatura máxima (100.0), etc. |
| 86 15<br>135 49<br>21 27<br>90 59 | **3º bloque de datos** (`chip_coord`): coordenadas (`x`, `y`) del vértice superior izquierdo de cada chip de una configuración. P.ej., primer chip: 86, 15; al ser de tamaño 40, 40 ocupará las posiciones (86-125, 15-54) en la rejilla básica de 200x100 puntos. |
| 126 40<br>26 72<br>168 29<br>62 23 | Segunda configuración |
| 67 35<br>129 2<br>22 11<br>119 84 | Tercera configuración |

El fichero `card` contiene la descripción de las configuraciones que hay que simular. Se trata de 20 configuraciones diferentes de 4 chips, con una rejilla de 2000x1000 puntos (factor de escala 10). Dado que el tiempo de ejecución es elevado, para las pruebas iniciales vamos a usar el fichero `card0`, que contiene solo cuatro configuraciones en el tamaño base de la rejilla, 200x100 puntos.

Para ejecutar el programa hay que indicar la tarjeta a simular junto con el ejecutable. Por ejemplo:

```
> heat_s card0
```

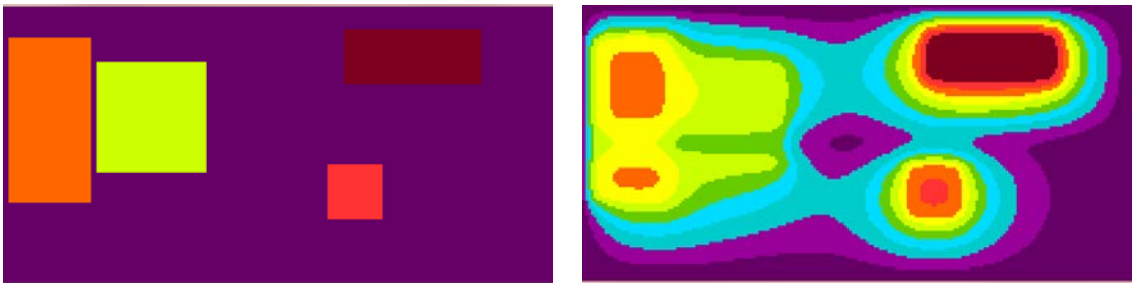
**▪ Resultados**

Como resultado de la simulación, se guarda en un *struct* (BT) los resultados de la configuración que menor temperatura media produce: número de configuración, temperatura media, matriz inicial de chips y matriz final de temperaturas. El programa genera con esos resultados dos ficheros: `card_ser.chips` (la matriz de los chips) y `card_ser.res` (la matriz final de temperaturas).

Una aplicación sencilla de visualización permite representar esas dos matrices para el caso de pruebas con factor de escala 1 (`card0`, matrices de 200x100 puntos), ejecutando:

```
> vfinder card0_ser.res
```

que abre una ventana y dibuja en diferentes colores la distribución de temperaturas obtenida tras la simulación (que puedes comparar con la inicial, que se encuentra en el fichero `card0_ser.chips`).



Distribución inicial de las temperaturas en la tarjeta `card0` (4 chips) y resultado final tras el proceso de simulación de la difusión térmica.

**▪ Estructura del programa**

El programa serie está dividido en tres módulos: `heat_s.c` (programa principal), `difussion.c` (con las funciones `thermal_update` y `difussion`), y `faux.c` (con las rutinas auxiliares de lectura de datos y generación de resultados). Para generar el ejecutable hay que compilar los tres programas; por ejemplo:

```
>icc -o heat_s heat_s.c diffusion.c faux.c
```

Todos los ficheros (módulos fuente `.c`, ficheros de cabecera `.h`, y definición de tarjetas) los tienes en el directorio `templates/aplicacion`.

## 3.2 TAREAS A REALIZAR

Hay que paralelizar el programa serie para poder ejecutarlo en el *cluster*. Se propone realizar dos versiones del código paralelo.

▪ **Fase 1**

Cada configuración de las 20 se va a ejecutar en paralelo entre todos los procesos; tras terminar la primera, se pasa a simular la segunda, la tercera, etc., hasta acabar con todas.

La rejilla de puntos de la tarjeta la vamos a repartir entre los procesos por franjas horizontales. Ten en cuenta que, de manera similar a como has resuelto el problema 3.2 del puzle, en cada iteración cada proceso va a necesitar datos, los correspondientes a las fronteras, que ese encuentran en los procesos `pid+1` y `pid-1`.

Puedes comprobar que los ficheros de resultados que obtienes son correctos, por ejemplo, mediante una comparación entre ellos con el comando `diff`:

```
> diff card0_ser.res card0_par.res
```

y visualizar la distribución de temperaturas ejecutando:

```
> vfinder card0_par.res        (o card0_par.chips)
```

Una vez verificado que el programa es correcto, ejecútalo con la tarjeta de configuraciones completa (`card`), en serie y con 2, 4, 8, 16, 24 y 32 procesos. Obtén los tiempos de ejecución, y calcula el factor de aceleración y los *speed-ups* conseguidos. Representa gráficamente esos datos y extrae las conclusiones pertinentes. En base a esos resultados, estima cuál es el número de procesos (*P*) más adecuado para este problema en este *cluster*.

▪ **Fase 2**

Una vez completada la primera, hay que realizar una segunda versión con una estrategia diferente. En lugar de que todos los procesos se dediquen a ejecutar en paralelo cada configuración de chips, vamos a distribuir los procesos en un proceso *manager* y grupos de *P workers* (el valor estimado en la primera fase), y efectuar un planificación dinámica de las tareas, tal como has hecho en el ejercicio 3.1 del puzle.

Así, el *manager* distribuye una configuración a cada grupo bajo demanda de éstos. Cada grupo de *P* procesos simula una configuración y devuelve el resultado obtenido al *manager*, para que le envíe una nueva configuración para simular, hasta terminar con todas las configuraciones entre todos los grupos.

Para esta versión, tienes que definir y utilizar los grupos de procesos, para que se intercambien información entre ellos. En cada grupo de *P*, uno de ellos será el encargado de solicitar tareas al *manager* y de devolverle resultados. En cada grupo, la simulación de la tarjeta se realiza con el mismo procedimiento de la primera versión.

Una vez verificado el programa (utilizando el fichero `card0`), ejecútalo con el fichero de entrada `card`. Mide los tiempos de ejecución para el caso de 1+1x*P*, 1+2x*P*, 1+3x*P*, 1+4x*P*… procesos, y calcula los *speed-ups* y eficiencias conseguidos. Compara los resultados de ambas versiones y justifica los resultados que has obtenido.

▪ **Informe técnico**

Como resultado del proyecto hay que escribir un informe técnico que describa el problema a resolver, cómo se ha resuelto, los resultados obtenidos y las conclusiones (para ambas fases). El informe debe contener las gráficas, tablas de datos y trozos de código comentados necesarios para su correcta explicación, de acuerdo a las directrices del documento guía. Como anexo, hay que incluir el código completo de la aplicación.

### 3.3 ENTREGABLES Y PLAZOS

E6.1 Resultados preliminares de la fase 1 de la aplicación: resultados numéricos y gráficos obtenidos en la fase 1 (un par de hojas). Fecha: **26 de abril**.

E6.2 Informe técnico definitivo. Fecha: **20 de mayo**.

E6.3/E7 Presentación oral del trabajo, y entrega de la carpeta final. Fecha: **24 de mayo**.

## 3.4 CÓDIGO DE LA VERSIÓN SERIE

**/* File: defines.h */**

```
// minimal card and maximun size
#define RSIZE 200
#define CSIZE 100
#define MAX_GRID_POINTS 3000000

#define NROW (RSIZE*param.scale + 2)  // extended row number
#define NCOL (CSIZE*param.scale + 2)  // extended column number

struct info_param {
  int    nconf, nchip, max_iter, scale; // n. config., n. chips, max. n. iter., card size scale
  float  t_ext, tmax_chip, t_delta; // external t., max. t. of a chip, t. incr. for convergence
};

struct info_chip {
  int    h, w;         // size (h, w)
  float  t_chip;       // temperatura
};

struct info_results {
  double Tmean;                     // mean temp.
  int    conf;                      // configuration number
  float  bgrid[MAX_GRID_POINTS];    // final grid
  float  cgrid[MAX_GRID_POINTS];    // initial grd (chips)
};
```

**/* File: heat_s.c */**

```
#include <stdio.h>
#include <values.h>
#include <sys/time.h>

#include "defines.h"
#include "faux.h"
#include "diffusion.h"

// global variables
float grid_chips[MAX_GRID_POINTS], grid[MAX_GRID_POINTS], grid_aux[MAX_GRID_POINTS];
struct info_results BT;


/*****************************************************************************/
void init_grid_chips (int conf, struct info_param param, struct info_chips *chips,
                      int **chip_coord, float *grid_chips)
{
  int i, j, n;

  for (i=0; i<NROW; i++)
  for (j=0; j<NCOL; j++)
    grid_chips[i*NCOL+j] = param.t_ext;

  for (n=0; n<param.nchip; n++)
  for (i=chip_coord[conf][2*n]*param.scale; i<(chip_coord[conf][2*n]+chips[n].h)*param.scale; i++)
  for (j=chip_coord[conf][2*n+1]*param.scale; j<(chip_coord[conf][2*n+1]+chips[n].w)*param.scale; j++)
    grid_chips[(i+1)*NCOL+(j+1)] = chips[n].tchip;
}


/*****************************************************************************/
void init_grids (struct info_param param, float *grid, float *grid_aux)
{
  int i, j;

  for (i=0; i<NROW; i++)
  for (j=0; j<NCOL; j++)
    grid[i*NCOL+j] = grid_aux[i*NCOL+j] = param.t_ext;
}
```

```
/*****************************************************************************/
/*****************************************************************************/
int main (int argc, char *argv[])
{
  struct info_param param;
  struct info_chips *chips;
  int    **chip_coord;

  int    conf, i;

  struct timeval t0, t1;
  double *tej, tsim = 0.0;


 // reading initial data file
  if (argc != 2) {
    printf ("\n\nERROR: needs a card description file \n\n");
    exit (-1);
  }

  read_data (argv[1], &param, &chips, &chip_coord);

  printf ("\n  ======================================================================");
  printf ("\n    Thermal diffusion - SERIAL version ");
  printf ("\n    %d x %d points, %d chips", RSIZE*param.scale, CSIZE*param.scale, param.nchip);
  printf ("\n    T_ext = %1.1f, Tmax_chip = %1.1f, T_delta: %1.3f, Max_iter: %d", param.t_ext,
              param.tmax_chip, param.t_delta, param.max_iter);
  printf ("\n  ======================================================================\n\n");

  BT.Tmean = MAXDOUBLE;
  tej = (double *) malloc(param.nconf * sizeof(double));


  // loop to process chip configurations
  for (conf=0; conf<param.nconf; conf++)
  {
    gettimeofday (&t0, 0);

    // inintial values for grids
    init_grid_chips (conf, param, chips, chip_coord, grid_chips);
    init_grids (param, grid, grid_aux);

    // main loop: thermal injection/disipation until convergence (t_delta or max_iter)
    diffusion (param, grid, grid_chips, grid_aux);

    // processing configuration results
    gettimeofday (&t1, 0);
    tej[conf] = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec)/1e6;
    results_conf (conf, param, grid, grid_chips, &BT);
  }


  // writing best configuration results
  results (param, &BT, argv[1]);
  for (conf=0; conf<param.nconf; conf++) tsim += tej[conf];
  printf ("   > Time (serial): %1.3f s \n\n", tsim);

  free (tej);
  free (chips);
  for (i=0; i<param.nconf; i++) free (chip_coord[i]);
  free (chip_coord);
}
```

**/* File: difussion.c */**

```
#include "defines.h"


/*****************************************************************************/
void thermal_update (struct info_param param, float *grid, float *grid_chips)
{
  int i, j;

  // heat injection at chip positions
  for (i=1; i<NROW-1; i++)
  for (j=1; j<NCOL-1; j++)
    if (grid_chips[i*NCOL+j] > grid[i*NCOL+j])
      grid[i*NCOL+j] += 0.05 * (grid_chips[i*NCOL+j] - grid[i*NCOL+j]);

  // air cooling at the middle of the card
  int a = 0.45*(NCOL-2) + 1;
  int b = 0.55*(NCOL-2) + 1;

  for (i=1; i<NROW-1; i++)
  for (j=a; j<b; j++)
      grid[i*NCOL+j] -= 0.01 * (grid[i*NCOL+j] - param.t_ext);
}


/*****************************************************************************/
void diffusion (struct info_param param, float *grid, float *grid_chips, float *grid_aux)
{
  int    i, j, end, niter;
  float  T;
  double Tmean, Tmean0 = param.t_ext;

  end = 0; niter = 0;

  while (end == 0)
  {
    niter++;
    Tmean = 0.0;

    // heat injection and air cooling
    thermal_update (param, grid, grid_chips);

    // thermal diffusion
    for (i=1; i<NROW-1; i++)
    for (j=1; j<NCOL-1; j++)
    {
      T = grid[i*NCOL+j] +
          0.10 * (grid[(i+1)*NCOL+j] + grid[(i-1)*NCOL+j] + grid[i*NCOL+(j+1)] +
                  grid[i*NCOL+(j-1)] + grid[(i+1)*NCOL+j+1] + grid[(i-1)*NCOL+j+1] +
                  grid[(i+1)*NCOL+(j-1)] + grid[(i-1)*NCOL+(j-1)]
                 - 8*grid[i*NCOL+j]);

      grid_aux[i*NCOL+j] = T;
      Tmean += T;
    }

    //new values for the grid
    for (i=1; i<NROW-1; i++)
    for (j=1; j<NCOL-1; j++)
      grid[i*NCOL+j] = grid_aux[i*NCOL+j];

    // convergence every 10 iterations
    if (niter % 10 == 0)
    {
      Tmean = Tmean / ((NCOL-2)*(NROW-2));
      if ((fabs(Tmean - Tmean0) < param.t_delta) || (niter > param.max_iter))
           end = 1;
      else Tmean0 = Tmean;
    }
  } // end while
  printf ("Iter: %d\t", niter);
}
```

```
/* File: faux.c */

#include <stdio.h>
#include <values.h>
#include "defines.h"


/*****************************************************************************/
void read_data (char *file_name, struct info_param *param, struct info_chips **chips,
                      int ***chip_coord)
{
  int   i, j, h, w;
  float tchip;
  FILE  *fdin;

  fdin = fopen (file_name, "r");

  // simulation parameters (param)
  fscanf (fdin, "%d %d %f %f %f %d %d", &param->nconf, &param->nchip, &param->t_ext,
       &param->tmax_chip, &param->t_delta, &param->max_iter, &param->scale);
  if (param->scale > 12) {
    printf("\n\nERROR: maximum scale factor is 12 \n\n");
    exit (-1);
  }

  // chip sizes and temperatures
  *chips = (struct info_chips *) malloc (param->nchip * sizeof(struct info_chips));

  for (i=0; i<param->nchip; i++) {
    fscanf (fdin, "%d %d %f", &h, &w, &tchip);
    (*chips)[i].h = h;
    (*chips)[i].w = w;
    (*chips)[i].tchip = tchip;
  }

  // chip positions
  *chip_coord = (int **) malloc (param->nconf * sizeof(int*));
  for (i=0; i<param->nconf; i++)
     (*chip_coord)[i] = (int*) malloc (2 * param->nchip * sizeof(int));

  for (i=0; i<param->nconf; i++)
  for (j=0; j<param->nchip; j++)
    fscanf (fdin, "%d %d", &(*chip_coord)[i][2*j], &(*chip_coord)[i][2*j+1]);

  fclose (fdin);
}


/*****************************************************************************/
void results_conf (int conf, struct info_param param, float *grid, float *grid_chips,
                      struct info_results *BT)
{
  int    i, j;
  float  Tmax = MINFLOAT, Tmin = MAXFLOAT;
  double Tmean = 0.0;

  for (i=1; i<NROW-1; i++)
  for (j=1; j<NCOL-1; j++)
    Tmean += grid[i*NCOL+j];

  Tmean = Tmean / ((NROW-2)*(NCOL-2));

  if (BT->Tmean > Tmean)
  {
    BT->Tmean = Tmean;
    BT->conf = conf;
    for (i=1; i<NROW-1; i++)
    for (j=1; j<NCOL-1; j++) {
      BT->bgrid[i*NCOL+j] = grid[i*NCOL+j];
      BT->cgrid[i*NCOL+j] = grid_chips[i*NCOL+j];
    }
  }
  printf ("Config: %2d \t Tmean: %1.2f\n", conf+1, Tmean);
}
```

```
/*****************************************************************************/
void fprint_grid (FILE *fd, float *grid, struct info_param param)
{
  int i, j;

  // j - i order for better visualitation
  for (j=NCOL-2; j>0; j--)
  {
    for (i=1; i<NROW-1; i++) fprintf (fd, "%1.2f ", grid[i*NCOL+j]);
    fprintf (fd, "\n");
  }
  fprintf (fd, "\n");
}


/*****************************************************************************/
void results (struct info_param param, struct info_results *BT, char *finput)
{
  FILE  *fd;
  char  name[100];

  printf ("\n\n >>> BEST CONFIGURATION: %2d\t Tmean: %1.2f\n\n", BT->conf+1, BT->Tmean);

  sprintf (name, "%s_ser.res", finput);
  fd = fopen (name, "w");
  fprintf (fd, "Tmin_ini %1.1f  Tmax_ini %1.1f  \n", param.t_ext, param.tmax_chip);
  fprintf (fd, "%d\t  %d \n", NCOL-2, NROW-2);

  fprint_grid (fd, BT->bgrid, param);

  fprintf (fd, "\n\n >>> BEST CONFIGURATION: %d\t Tmean: %1.2f\n\n", BT->conf+1, BT->Tmean);
  fclose (fd);

  sprintf (name, "%s_ser.chips", finput);
  fd = fopen (name, "w");
  fprintf (fd, "Tmin_chip %1.1f  Tmax_chip %1.1f  \n", param.t_ext, param.tmax_chip);
  fprintf (fd, "%d\t  %d \n", NCOL-2, NROW-2);

  fprint_grid (fd, BT->cgrid, param);

  fclose (fd);
}
```

## ANEXO: actas y documentos

Este Anexo incorpora todos los documentos que deben ser cumplimentados por el alumnado en el desarrollo del proyecto: actas de constitución y documento de compromisos del grupo, actas de las reuniones de trabajo, plantilla de dedicación no presencial al proyecto y encuestas de satisfacción. Así mismo, se han incluido las rúbricas de evaluación utilizadas para realizar la co-evaluación de las presentaciones, y las guías para la elaboración de informes y mantenimiento del portafolio.

### A1. ACTA DE CONSTITUCIÓN DE GRUPO - DOCUMENTO DE COMPROMISOS

**▪ Participantes**

| | Nombre y apellidos | Firma |
|---|---|---|
| 1. | | |
| 2. | | |
| 3. | | |

**▪ Compromisos**

- Asistir a las reuniones de grupo que se realicen, tanto en clase, en sesiones presenciales, como fuera de ella, en actividades no presenciales.
- Realizar el trabajo asignado dentro del grupo en los plazos fijados.
- Llevar preparados a las reuniones los trabajos que se hayan comprometido a realizar.
- Asegurarse de que todos los miembros del grupo entienden todo el trabajo desarrollado.
- Hacer todo lo posible por conseguir un buen funcionamiento del grupo.
- Si surgen conflictos, comentarlos con franqueza, pero con respeto, con el objetivo de resolverlos.
- En caso de no cumplir con las obligaciones acordadas para el buen funcionamiento del grupo, asumir las posibles consecuencias: cambio de grupo, expulsión del grupo…
-

Fecha ..... Profesor/a .....

## A2 ACTA DE SESIÓN DE TRABAJO DE GRUPO

**▪ Asistentes**

| | Nombre y apellidos | Firma |
|---|---|---|
| 1. | | |
| 2. | | |
| 3. | | |

**▪ Temas tratados y decisiones tomadas**

1.

**▪ Temas pendientes. Distribución de tareas para la siguiente reunión y decisiones tomadas.**

1.

Fecha

Hora comienzo — Hora final

## A3 ESTIMACIÓN DEL TIEMPO DE TRABAJO NO PRESENCIAL ASOCIADO AL PROYECTO

(en periodos de 15 minutos; por ejemplo: 2 h 15 m, 3 h 30 m, 45 m, etc.)

| **Actividad** | **Tiempo** |
|---|---|
| **Tareas preliminares** | |
| estudio de MPI | |
| | |
| Total tareas preliminares | |
| **Puzle** | |
| lectura y búsqueda de información | |
| resolución de los ejercicios | |
| puesta en común en el grupo | |
| preparación de la presentación | |
| | |
| Total puzle | |
| **Aplicación** | |
| lectura y búsqueda de información | |
| desarrollo: diseño, programación, análisis de opciones/tiempos | |
| escritura de la memoria | |
| preparación de la presentación | |
| | |
| Total aplicación | |
| **Examen** | |
| preparación del examen | |
| | |
| Total examen | |
| **Total** | |

Muchas gracias por tu colaboración

## A4 RÚBRICA PARA LA COEVALUACIÓN DE PRESENTACIONES ORALES

| Aspecto a evaluar | **Excelente** [10 - 9] | **Satisfactorio** [8 - 7] | **Mejorable** [6 - 4] | **Deficiente** [3 - 0] |
|---|---|---|---|---|
| **Contenidos**<br>Correctos, adecuados, suficientes. | Cumple todo lo indicado. Demuestra un completo entendimiento del tema. | Hay alguna incorrección en los contenidos, y/o éstos no son completos. En todo caso, demuestra un buen entendimiento del tema. | Hay bastantes errores y lagunas en el contenido. Se da excesiva importancia a información secundaria. Parece que solo entiende algunas partes del tema. | Son claramente incorrectos e insuficientes. No parece entender el tema. |
| **Organización**<br>Claridad, lógica, estructuración, razonamiento. | La presentación es clara, lógica y está bien estructurada. Se puede seguir fácilmente la línea de razonamiento. | En general es clara, lógica y está bien estructurada, aunque algunos aspectos pueden resultar confusos. | Algunas ideas no están claras. Salta de unas a otras sin orden. Cuesta seguir la lógica del discurso. | No es nada clara ni lógica. No tiene estructura. Es casi imposible entender nada. |
| **Estilo**<br>Exposición de ideas, ritmo, postura, volumen, tono, pausas. | La presentación es adecuada para la audiencia. Expone las ideas a un ritmo adecuado. Salvo de manera puntual, no consulta sus notas. Se le ve cómodo/a delante del grupo y puede ser escuchado/a por todos/as. Realiza pausas en los momentos oportunos. | La presentación es, en general, adecuada para la audiencia. El ritmo es variable. A veces ha tenido que recurrir a sus notas. Se le ve con cierta incomodidad y ha habido algunos problemas para escucharle. Realiza pocas pausas. | Algunos aspectos de la presentación son muy elementales o muy sofisticados para la audiencia. El ritmo a veces es muy rápido o muy lento. Recurre muy a menudo a sus notas. Se le ve incómodo/a y hay más problemas para escucharle. Realiza pocas pausas. | La presentación es totalmente inadecuada para la audiencia. El ritmo es inexistente. Lee en todo momento sus notas, se le ve muy incómodo/a y hay que estar muy atento para poder escucharle. No realiza pausas. |
| **Lenguaje, vocabulario**<br>Científico, correcto. | Emplea un vocabulario adecuado. | El vocabulario es bastante adecuado. | El vocabulario es poco adecuado. | El vocabulario es completamente inadecuado. |
| **Recursos audiovisuales**<br>Homogeneidad, tamaño de letra, claridad. | El material utilizado es homogéneo y de calidad: el tamaño de letra es adecuado, la información está bien organizada, facilita la comprensión del tema, y resalta los aspectos principales. | El material utilizado es bueno: el tamaño de letra es adecuado, y la información está bien organizada, aunque algunos aspectos no han quedado bien reflejados. | La presentación es confusa y desorganizada. El tamaño de letra es pequeño y se ha incluido excesiva información, lo que dificulta el seguimiento. | Los materiales preparados para la presentación son muy deficientes. |
| **Tiempo**<br>Se ajusta al tiempo máximo asignado; capacidad de síntesis. | Distribución temporal equilibrada, adecuada para cada concepto. Se ajusta bien al tiempo disponible. | Distribución temporal algo descompensada, ya que ha sobrado algo de tiempo o ha tenido que apresurarse al final para cubrir todos los contenidos. | Distribución temporal más descompensada. Podría haber explicado conceptos con más calma o profundidad, o tendría que haber ido más rápido en aspectos secundarios. | Distribución temporal muy descompensada, por defecto o exceso. No es capaz de explicar los conceptos con cierto detalle, o de ceñirse a los más importantes. |

## A5 ACTA DE COEVALUACIÓN DE PRESENTACIÓN ORAL

Calificación de 0 a 10 de los diferentes aspectos de la presentación oral de cada grupo, de acuerdo a la rúbrica correspondiente:

**10 - 9** [ excelente ] **8 - 7** [ satisfactorio ] **6 - 4** [ mejorable ] **3 - 0** [ deficiente ]

La calificación final se obtiene teniendo en cuenta el peso de cada aspecto evaluado.

▪ **Grupo evaluador:**

| | % | Grupo 1 | Grupo 2 | Grupo 3 | Grupo 4 | Grupo 5 | Grupo 6 | Grupo 7 |
|---|---|---|---|---|---|---|---|---|
| Contenidos | 30 | | | | | | | |
| Organización | 25 | | | | | | | |
| Estilo | 15 | | | | | | | |
| Lenguaje | 10 | | | | | | | |
| Recursos audiovis. | 15 | | | | | | | |
| Tiempo | 5 | | | | | | | |
| **Calificación final** (no rellenar) | | | | | | | | |

Fecha:

Firmas:

Evaluador/a Evaluador/a Evaluador/a

## A6 SUGERENCIAS GENERALES SOBRE LA ESTRUCTURA Y CONTENIDO DEL PORTAFOLIO

Como idea general, el portafolio no debería ser una yuxtaposición de material generado durante el desarrollo del proyecto, sino que debería contener la información de manera estructurada, con un formato determinado (con cierta homogeneidad), y contextualizar la información contenida (por ejemplo, no incluir una tabla sin su correspondiente párrafo explicativo o sin un análisis de la misma, aunque sea breve).

La carpeta del proyecto no es estática e invariable, sino dinámica y mejorable. Por ejemplo, incluir una solución corregida de los ejercicios del puzle tras la puesta en común podría ser una mejora a incluir en ese momento. Así, la información que quede al final del proyecto en el portafolio será completa y correcta. Por ello, en cada entrega de la carpeta conviene indicar en una hoja las mejoras, añadidos, etc. que se hayan realizado desde la última revisión.

La carpeta tendrá dos formatos: en papel y digital. Los documentos (salvo las actas), además de en papel, se entregarán en versión digital: por ejemplo, los códigos fuente, transparencias utilizadas en las presentaciones, memoria de la aplicación desarrollada, etc. deberán estar también en la versión digital.

La primera versión del portafolio (en papel) se entregará el día de la presentación del puzle, para que sea revisada. La versión definitiva se entregará el 25 de mayo (día de la presentación del proyecto); por una parte, se entregará la versión en papel, y, por otra parte, se subirá a través de eGela la versión digital.

Como referencia, el portafolio podría estructurarse de la siguiente manera (es solo una opción):

- Portada: título del proyecto e identificación del grupo (componentes, cuenta de trabajo…).
- Índice con los contenidos del portafolio.
- Enunciado del proyecto y fuentes de información consultadas (información directa o referencias).
- Actividades durante el proyecto:
  - Póster inicial.
  - Puzle: (a) Temas planteados y resumen de las soluciones encontradas; (b) Ejercicios resueltos: explicación de la solución planteada, código fuente comentado y análisis de los resultados obtenidos.
  - Aplicación desarrollada. Informe técnico de acuerdo a las recomendaciones indicadas para el mismo.
- Apéndice 1. Actas de constitución del grupo y actas de las reuniones realizadas.
- Apéndice 2. Horas no presenciales dedicadas al proyecto por cada persona del grupo (de acuerdo a la plantilla entregada).

## A7 DIRECTRICES PARA LA REDACCIÓN DE INFORMES TÉCNICOS DE RESULTADOS

El trabajo desarrollado en el proyecto se recoge finalmente en un **informe técnico de resultados**, en el que se explica el problema, la solución propuesta, los resultados obtenidos, las consecuencias de los mismos, etc. El informe técnico es el **producto final del proyecto**, y hay que **dedicarle un mínimo de atención**, tanto al fondo como a la forma. Un buen trabajo (contenido) necesita ser bien explicado (forma). A modo orientativo, la **estructura** del informe podría ser la siguiente (adáptala a tu caso):

- Índice
- Introducción. Objetivos del proyecto y resumen de contenidos/resultados. Características principales de las herramientas utilizadas —hardware (arquitectura, procesador, memoria, frecuencia de reloj...) y software (S.O., compilador, nivel de optimización, versiones, aplicaciones…)— para poder colocar los resultados en un determinado contexto, reproducirlos, y compararlos con otros.
- Fundamentos teóricos. En función del tipo de informe, un resumen de los contenidos teóricos en los que se basa el trabajo desarrollado.
- Aplicación. Es la parte principal del informe de resultados. Debe incluir: (a) problema a resolver; (b) soluciones propuestas (desechadas), mostrando y explicando las partes de código más significativas; (c) resultados obtenidos, y explicación/justificación de los mismos; (d) en su caso, posibles alternativas, mejoras, cuestiones pendientes…
- Conclusiones generales, bibliografía y apéndices.

Estas **sugerencias** te pueden ayudar a evitar ciertos errores habituales en este tipo de informes.

**> Explicaciones**

- La explicación de los resultados obtenidos debe ser clara y concisa; no basta con indicar qué ocurre ("la curva primero sube y luego baja"), sino por qué ocurre. Si no se tiene explicación para un determinado comportamiento hay que dejar constancia de ello o aventurar una hipótesis. En muchos casos, una hipótesis lleva a hacer algún otro experimento y a verificar los resultados obtenidos con los esperados. Efectuar hipótesis y pruebas más allá de lo inicialmente previsto —ser creativo— es una práctica adecuada y bien valorada, aunque también es necesario considerar el tiempo disponible para ello.
- Si has usado algún tipo de documentación para el desarrollo del proyecto, debes indicarla siempre. No todas las referencias tienen el mismo nivel de "autoridad/calidad" técnica o científica; documéntate en fuentes adecuadas. En todo caso, salvo que sea una cita literal, no copies/traduzcas directamente; reescribe el texto con tus propias palabras una vez que lo hayas entendido y asimilado.

**> Datos**

- Si al repetir los experimentos obtienes tiempos de ejecución "significativamente" diferentes, hay que indicar no solo un valor (el mínimo, p. ej.) sino también el máximo, la media y la desviación típica. Esos cuatro valores ofrecen en algunos casos información complementaria sobre el resultado obtenido. Puedes añadir un bucle al programa para ejecutarlo *n* veces, y obtener los tiempos de cada ejecución. Es habitual eliminar de esos datos la primera ejecución "en frío", y aquellos que se aparten muy significativamente del resto (salvo, claro está, que tengan una explicación en el contexto del problema).
- Las tablas o gráficos con resultados deben indicar siempre las unidades (ms, byte, MB/s...).

**> Representación de los datos**

- En muchos casos, una tabla con resultados es más que suficiente para expresar un determinado comportamiento. Utiliza los gráficos cuando faciliten la compresión de los resultados.
- La representación gráfica de los datos es útil si permite visualizar de un "golpe" el comportamiento del sistema, o centrarnos en partes concretas del mismo. Por ello, las escalas de los ejes X e Y deben ser las adecuadas para que se observe bien en todo su rango de valores la función que se está representando. En general, la escala suele ser lineal o logarítmica. Los ejes deben ir etiquetados, incluyendo las unidades de las medidas que representan.
- En la mayoría de los casos, el eje X debe seguir una determinada escala para que la función dibujada pueda ser interpretada correctamente. P. ej., no se pueden representar igualmente espaciados en el eje X tiempos de ejecución correspondientes a tamaños de un vector de 100, 500, 600 y 4000 enteros.
- Conviene añadir un pie a las figuras y tablas del informe, del estilo de: "Figura/Tabla 3. Tiempo de ejecución (ms) del programa pr1 en función del tamaño de los vectores (double)".

**> Código**

- En muchos casos es necesario incluir en el informe trozos de código. Si es así, incluye solo aquello que es estrictamente necesario para interpretar lo realizado, y coméntalo adecuadamente. Para el código, utiliza un tipo de letra de paso constante y tamaño más reducido que el del texto general (por ejemplo, `courier 8`), e interlineado simple. Indexa el código para que sea fácilmente legible.

  En todo caso, incluye en un apéndice el código completo (si no es muy largo). Si el código queda recogido en alguna máquina, indica dónde (cuenta, directorio) y describe brevemente cada fichero.

**> Estilo, formato**

- Dado que el contenido de un informe técnico/científico puede ser complejo, la redacción del mismo debe ser lo más sencilla posible —frases no muy largas, simples, gramaticalmente correctas...— sin renunciar por ello a explicar cuestiones complejas. Un texto farragoso e ininteligible hace imposible analizar su contenido. Si puedes, deja el texto un cierto tiempo y vuelve a releerlo; te darás cuenta de los posibles fallos del mismo. También es útil que otra persona lea el texto, para ver si lo entiende. Recuerda que un informe de este tipo va dirigido a terceras personas, no al que lo escribe.
- Las herramientas actuales de edición permiten generar textos con un alto nivel de calidad formal: una maquetación adecuada, gráficos de calidad, sin faltas de ortografía, etc. El "texto con fallos 0" es casi imposible de conseguir, pero no debemos estar muy lejos del mismo.

  Utiliza el formato que más te guste, pero que sea simple, sin muchas “florituras”. Por ejemplo: tipo de letra “times”, "calibri" o similares (o los de LaTeX, según la herramienta que utilices); tamaño de 10/11 puntos para el texto, algo mayor (12/14 puntos) para las cabeceras, y algo menor para los pies de tablas y figuras (8/10); interlineado sencillo; márgenes de 2,5/3 cm; impresión a doble cara.
- Las páginas deben ir numeradas, y, si el documento es extenso, incluir un índice. No por tener muchas hojas es mejor el informe, aunque tampoco hay que pasarse en sentido contrario. Si haces referencia a colores (de un gráfico, p. ej.), recuerda imprimir al menos ese trozo en color. Finalmente, unas grapas, una carpeta o un encuadernado sencillo son siempre preferibles a hojas sueltas.

En resumen, el **informe de resultados debe ser conciso, completo, claro y legible**, como para que pueda ser entendido y valorado por cualquier otra persona con unos mínimos conocimientos sobre el tema (por ejemplo, un compañero o compañera que no haya cursado esta asignatura).

## A8 ENCUESTA DE SATISFACCIÓN ERAGIN SOBRE EL PROYECTO

**CUESTIONARIO DE OPINIÓN SOBRE LA METODOLOGÍA ABP**

Te pedimos que nos des tu opinión sobre varios aspectos de la metodología que se ha seguido en el aula. Tus respuestas serán analizadas, y nos permitirán mejorar nuestras propuestas en el futuro. Por eso, te pedimos que le dediques el tiempo necesario, y contestes con sinceridad. Muchas gracias.

Teniendo en cuenta todos los aspectos de la metodología que hemos trabajado, tu **valoración global** del planteamiento y desarrollo de la experiencia es:

☐ nada satisfactoria ☐ poco satisfactoria
☐ bastante satisfactoria ☐ muy satisfactoria

**Justifica** tu valoración:

Valora el grado en que consideras que la metodología seguida **te ha ayudado a aprender**, en **comparación** con planteamientos metodológicos más tradicionales:

☐ me ha ayudado menos ☐ me ha ayudado igual
☐ me ha ayudado más ☐ me ha ayudado mucho más

Valora el grado en que consideras que el **uso de esta metodología te ha ayudado** a:
("1" muy poco, "2" poco, "3" bastante, "4" mucho)

| | | | | |
|---|---|---|---|---|
| Comprender contenidos teóricos | 1 | 2 | 3 | 4 |
| Establecer relaciones entre teoría y práctica | 1 | 2 | 3 | 4 |
| Relacionar los contenidos de la asignatura y obtener una visión integrada | 1 | 2 | 3 | 4 |
| Aumentar el interés y la motivación por la asignatura | 1 | 2 | 3 | 4 |
| Analizar situaciones de la práctica profesional | 1 | 2 | 3 | 4 |
| Indagar por tu cuenta en torno al trabajo planteado | 1 | 2 | 3 | 4 |
| Tomar decisiones en torno a una situación real | 1 | 2 | 3 | 4 |
| Resolver problemas o ofrecer soluciones a situaciones reales | 1 | 2 | 3 | 4 |
| Desarrollar tus habilidades de comunicación (oral o escrita) | 1 | 2 | 3 | 4 |
| Desarrollar tu autonomía para aprender | 1 | 2 | 3 | 4 |
| Tomar una actitud participativa respecto a tu aprendizaje | 1 | 2 | 3 | 4 |
| Mejorar tus capacidades de trabajo en grupo | 1 | 2 | 3 | 4 |
| Desarrollar competencias necesarias en la práctica profesional | 1 | 2 | 3 | 4 |
| El sistema de evaluación seguido ha sido adecuado a la metodología | 1 | 2 | 3 | 4 |

La orientación proporcionada por el/la profesor/a durante el proceso, ¿ha satisfecho tus necesidades?

☐ Poco ☐ Suficiente
☐ Bastante ☐ Mucho

¿Cambiarías algo? ¿Se te ocurre alguna propuesta de mejora?

Si el próximo curso/módulo/cuatrimestre pudieras elegir, ¿optarías por esta metodología?

☐ Sí ☐ No