# Portfolio

Jose Ángel Gumiel y Mikel Dalmau

24 de Mayo de 2016

Año: 2015 - 2016
Profesor(a):
Teléfono:
E-mail: mdalmau002@ikasle.ehu.es

GIPUZKOAKO CAMPUSA
CAMPUS DE GUIPUZCOA
Paseo Manuel de Lardizabal, 1
20018 Donostia (Gipuzkoa)

# Índice

# 1 La aplicación

La aplicación a paralelizar se trata de un algoritmo que estudia placas con microchips, precisamente lo que realiza es una simulación de distribución de calor, con el objetivo de hallar la configuración que una vez estabilizada, tenga la temperatura media mínima.

Véase en el Anexo apartado 3.4 *Escenario y Puzle* una introducción más extensa al problema y a las herramientas de trabajo.

## 1.1 El programa serie

El programa serie está compuesto por los siguientes ficheros:

- *heats.c* contiene el main y se encarga de leer el fichero de configuraciones de la placa así como de algunas inicializaciones, tiene un bucle principal que ejecuta cada configuración y recoge resultados.

- La ejecución de las configuraciones se realiza en el fichero *difussion.c* que implementa la función de difusión del calor y es donde se realiza la mayor parte del cálculo.

- Por otro lado, están los ficheros *faux.c* y *defines.h* que se encargan de algunas funciones básicas de lectura y síntesis de resultados y de algunas definiciones de variables y estructuras.

En el Anexo apartado 3.5 *Aplicación a Paralelizar* puede hallarse mas información sobre cada fichero y su código fuente.

## 1.2 Fase 1: Paralelización del programa serie

En este apartado se muestran los cambios más significativos realizados para adaptar el programa serie al modelo paralelo. En *heat.c* se ha cambiado la *Lectura de los datos* (1.2.1) y se realiza el *Reparto del dominio* (1.2.2), esto es, un proceso reparte y recoge los datos tras el cálculo.

En *diffusion.c* se ha adaptado el código para que cada proceso calcule tantos datos como le corresponden y es aquí donde se lidia con el *Problema de la frontera* (1.2.3). También se ha adaptado el *Cálculo de la temperatura media* (1.2.4).

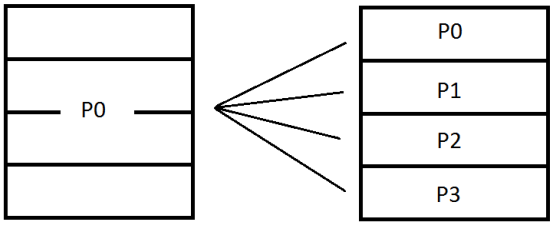
### 1.2.1 Lectura de los datos:

Con múltiples procesos de trabajo solo uno de ellos realiza la lectura del fichero de configuraciones. De los datos leídos, empaqueta y envía al resto los parámetros que necesitan.

```
if(pid == 0){
    ...
    ...
    ...
    MPI_Pack(&param.scale, 1, MPI_INT, buf,
        sizebuf, &pos, MPI_COMM_WORLD);
    MPI_Pack(&param.nconf, 1, MPI_INT, buf,
        sizebuf, &pos, MPI_COMM_WORLD);
    MPI_Pack(&param.t_ext, 1, MPI_FLOAT, buf
        , sizebuf, &pos, MPI_COMM_WORLD);
    MPI_Pack(&param.t_delta, 1, MPI_FLOAT,
        buf, sizebuf, &pos, MPI_COMM_WORLD);
    MPI_Pack(&param.max_iter, 1, MPI_INT,
        buf, sizebuf, &pos, MPI_COMM_WORLD);
}
MPI_Bcast(&buf, sizebuf, MPI_PACKED, 0,
    MPI_COMM_WORLD);

if(pid!=0){
    MPI_Unpack(buf, sizebuf, &pos, &param.
        scale,    1, MPI_INT, MPI_COMM_WORLD
        );
    MPI_Unpack(buf, sizebuf, &pos, &param.
        nconf,    1, MPI_INT, MPI_COMM_WORLD
        );
    MPI_Unpack(buf, sizebuf, &pos, &param.
        t_ext,    1, MPI_FLOAT,
        MPI_COMM_WORLD);
    MPI_Unpack(buf, sizebuf, &pos, &param.
        t_delta,  1, MPI_FLOAT,
        MPI_COMM_WORLD);
    MPI_Unpack(buf, sizebuf, &pos, &param.
        max_iter, 1, MPI_INT, MPI_COMM_WORLD
        );
}
```

### 1.2.2 Reparto del dominio:

El reparto de los datos de la placa se ha hecho por bloques de filas, la placa está ordenada de esa manera en un array de gran longitud. Cualquier otro tipo de reparto requeriría de costosas operaciones sobre los datos. Cabe destacar que la placa es rectangular y el tamaño de las filas corresponde al lado más corto, de no ser así habría sido mejor repartir bloques de columnas.



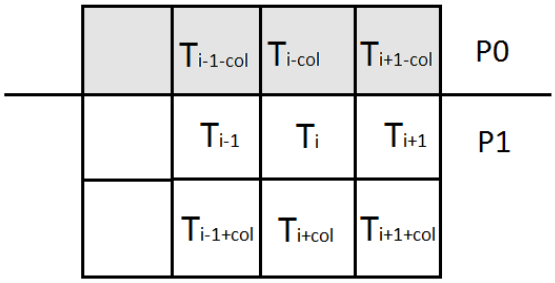

Primero cada proceso calcula los vectores de tamaño y desplazamiento para saber cuantos datos le corresponden y cual es su primer dato. De esta manera, para cada configuración, cuando el proceso líder construya la placa de chips podrá realizar una llamada colectiva Scatter y distribuir cada trozo a su proceso correspondiente.

```
MPI_Scatterv(&grid_chips[NCOL], &size[0], &
    displacement[0],  MPI_FLOAT,&grid_aux[
    NCOL], size[pid], MPI_FLOAT, 0,
    MPI_COMM_WORLD);

//Update values of the grid
for (i=1; i<=nrows; i++)
for (j=1; j<NCOL-1; j++)
    grid_chips[i*NCOL+j] = grid_aux[i*NCOL+j
        ];

init_grids(param, grid, grid_aux, nrows);

// main loop: thermal injection/dispation
    until convergence (t_delta or max_iter)
diffusion (param, &grid[0], &grid_chips[0],
    &grid_aux[0], nrows, npr, pid);

// Gathering of grid
MPI_Gatherv(&grid_aux[NCOL], size[pid],
    MPI_FLOAT, &grid[NCOL], &size[0], &
    displacement[0], MPI_FLOAT, 0,
    MPI_COMM_WORLD);
```

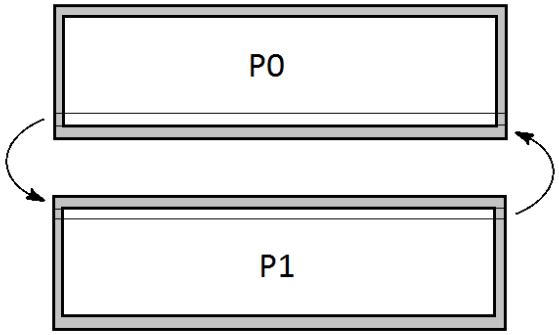
Notese que en la función colectiva, se comienza a enviar desde NCOL, esto es, desde la segunda línea, y equivalentemente cada proceso recoge los datos comenzando desde la segunda línea. En el siguiente apartado se explica el motivo de esto.

### 1.2.3 Problema de la frontera:

La función de difusión térmica que se implementa, para actualizar la temperatura de una casilla, necesita conocer la temperatura de las que la rodean. Esto supone un problema con las casillas que requieren de datos de otro proceso. En la imagen siguiente se muestra una ejemplo del problema de la frontera en la difusión, donde se quiere calcular $T_i$ en P1 y se necesita conocer tres puntos que corresponden a P0.




Para trabajar con este problema, se han añadido dos filas más a cada bloque, una en la parte superior y otra en la inferior. Estas filas corresponden a los datos de frontera de los bloques contiguos, y se actualizan mediante envíos y recepciones antes de procesar cada iteración.




En el siguiente código aparecen una serie de envíos y recepciones, donde los procesos primero y último, solo realizarán un envío/recepción de frontera, la inferior o la superior dependiendo del caso. El resto de procesos comunicarán ambas fronteras.

```
if(pid < npr - 1){
    MPI_Send(&grid[NCOL*nrows], NCOL,
        MPI_FLOAT, pid+1 , 0, MPI_COMM_WORLD
        );
}
if(pid > 0){
    MPI_Recv(&grid[0], NCOL, MPI_FLOAT,
        pid-1, 0, MPI_COMM_WORLD,&info);
    MPI_Send(&grid[NCOL], NCOL, MPI_FLOAT,
        pid-1 , 0, MPI_COMM_WORLD);
}
if(pid < npr - 1){
    MPI_Recv(&grid[NCOL*(nrows+1)], NCOL,
        MPI_FLOAT, pid+1, 0, MPI_COMM_WORLD,
         &info);
}
```

### 1.2.4 Cálculo de la temperatura media

Cada 10 iteraciones del búcle de difusión del calor, se calcula la temperatura media de la placa para ver si esta se ha estabilizado. Ahora, la temperatura está dividida entre los distintos procesos. Hemos utilizado la función MPI_AllReduce para que todos los procesos tengan la temperatura total de la placa y todos alcancen así el criterio de convergencia a la vez.

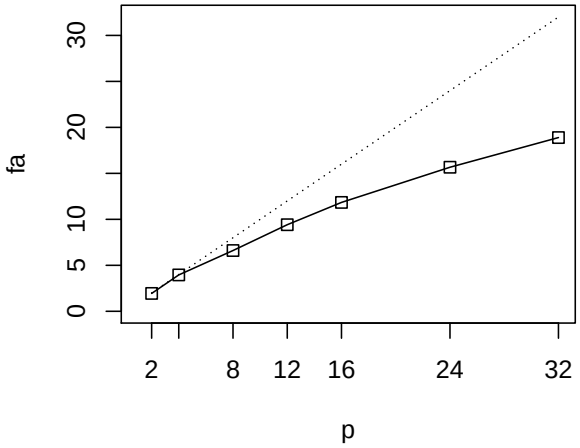
```
// convergence every 10 iterations
if (niter % 10 == 0){

    MPI_Allreduce(&Tmean, &tmean, 1,
        MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD)
        ;

    tmean = tmean / ((NCOL-2)*(NROW-2));
    if ((fabs(tmean - Tmean0) < param.
        t_delta) || (niter > param.max_iter)
        )
        end = 1;
    else
        Tmean0 = tmean;
}
```

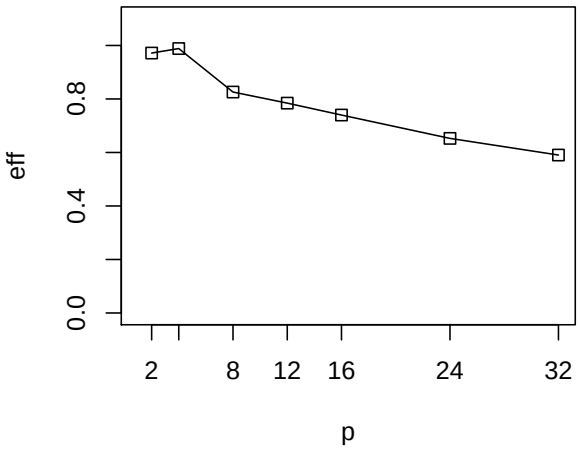
### 1.2.5 Resultados de la paralelización

Para ejecutar las pruebas se ha utilizado un fichero de configuraciones llamado *card.txt* que contiene 20 configuraciones distintas. Cada placa tiene 2000 x 1000 bloques y cuatro chips de distintos tamaños.

La siguiente tabla muestra los tiempos medios de cinco ejecuciones para distintos números de procesadores. De los gráficos se puede distinguir la eficiencia y speed-up de cada caso. El descenso de la eficiencia se explica por el sobrecoste de comunicación que se da al enviar las fronteras. Podemos concluir que 8 es el número de procesadores optimo par el tamaño de grid utilizado.

| **Proc.** | **T**(s) | **speed-up** | **efficiency** |
|---|---|---|---|
| serial | 2400 | - | - |
| 2 | 1235.07 +- 17 | 1.94 | 0.97 |
| 4 | 607.01 +- 1 | 3.95 | 0.99 |
| 8 | 363.25 +- 4 | 6.61 | 0.83 |
| 12 | 254.95+- 1 | 9.41 | 0.78 |
| 16 | 202.72 +- 1 | 11.84 | 0.74 |
| 24 | 153.23 +- 2 | 15.66 | 0.65 |
| 32 | 127.04 +- 1 | 18.89 | 0.59 |



## 1.3 Fase 2: Mejoras del programa paralelo

### 1.3.1 Comunicaciones inmediatas

Las funciones de MPI ISend y Irecv, permiten avanzar al programa sin haber finalizado la comunicación. Esta funcionalidad puede aprovecharse para solapar tiempos de cálculo y comunicación. Así hemos hecho con el envío de las fronteras, se trata de la operación de comunicación más costosa, y era necesario esperar a que se finalizara para recalcular las temperaturas.

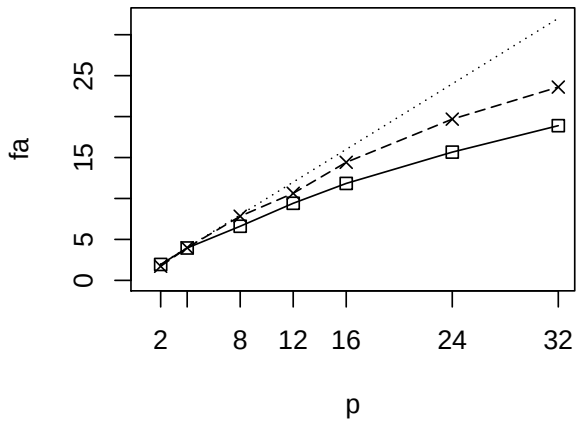
Ahora se procede a realizar el cálculo directamente, obviando la primera y última líneas, que se calcularán al final del todo, tras comprobar que la comunicación está realizada.

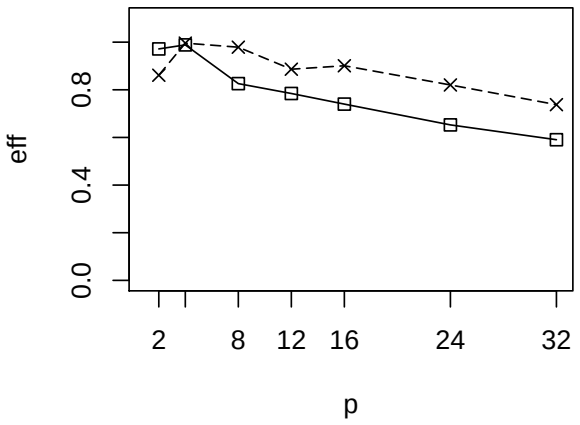
El modelo de paso de mensajes implica que el emisor inicia la comunicación. Tal y como se indica en [4], las comunicaciones tendrán en general menores costos si las funciones Receive ya han sido publicadas en el momento en que el emisor inicia la comunicación, los datos pueden ser movidos directamente al buffer de recepción y no hay necesidad de poner en cola una petición de envío. Así hemos procedido, publicando primero todas las llamadas Irecv y luego las Isend.

```
if(pid > 0){
    MPI_Irecv(&grid[0], NCOL, MPI_FLOAT, pid
        -1, 0, MPI_COMM_WORLD,&reqs[0]);
}
if(pid < npr -1 ){
    MPI_Irecv(&grid[NCOL*(nrows+1)], NCOL,
        MPI_FLOAT, pid+1, 0, MPI_COMM_WORLD,
         &reqs[1]);
}
if(pid > 0){
    MPI_Isend(&grid[NCOL], NCOL, MPI_FLOAT,
        pid-1 , 0, MPI_COMM_WORLD,&reqs[2]);
}
if(pid < npr - 1){
    MPI_Isend(&grid[NCOL*nrows], NCOL,
        MPI_FLOAT, pid+1 , 0, MPI_COMM_WORLD
        ,&reqs[3]);

    ...
    //Difusion del calor
    ...

if( (pid > 0) && (pid < npr -1) ){

    MPI_Testall(4,reqs, &flag, stats);
    if(!flag){MPI_Waitall(4,reqs,stats);}

}else if( pid == 0 ){

    MPI_Test(&reqs[1], &flag, &stats[1]);
    if(!flag){MPI_Wait(&reqs[1], &stats[1])
        ;}
    MPI_Test(&reqs[3], &flag, &stats[3]);
    if(!flag){MPI_Wait(&reqs[3], &stats[3])
        ;}

}else{

    MPI_Test(&reqs[0], &flag, &stats[0]);
    if(!flag){MPI_Wait(&reqs[0], &stats[0])
        ;}
    MPI_Test(&reqs[2], &flag, &stats[2]);
    if(!flag){MPI_Wait(&reqs[2], &stats[2])
        ;}

}

    ...
    //Difusion primera y/o ultima linea
    ...
```

La siguiente tabla, compara los tiempos obtenidos con los anteriores. En las gráficas, la línea continua con puntos cuadrados corresponde a los valores anteriores y la línea discontinua con cruces a los nuevos valores. La mejora en los factores de aceleración y eficiencia es considerable en casi todos los casos, sobre todo en los de muchos procesos, que eran los más afectados por el problema de la frontera, y aunque la eficiencia también desciende ya no se aprecia ese brusco descenso a partir de los 8 procesos.

| **Proc.** | **T**(s) | **T I**(s) | **speed-up** | **speed-up I** | **efficiency** | **efficiency I** |
|---|---|---|---|---|---|---|
| serial | 2400 | - | - | - | - | - |
| 2 | 1235.07 +- 17 | 1393.69 +- 0 | 1.94 | 1.72 | 0.97 | 0.86 |
| 4 | 607.01 +- 1 | 602.47 +- 0.2 | 3.95 | 3.98 | 0.99 | 1 |
| 8 | 363.25 +- 4 | 306.49 +- 1 | 6.61 | 7.83 | 0.83 | 0.98 |
| 12 | 254.95 +- 1 | 225.63+- 5 | 9.41 | 10.64 | 0.78 | 0.89 |
| 16 | 202.72 +- 1 | 166.49 +- 1 | 11.84 | 14.42 | 0.74 | 0.9 |
| 24 | 153.23 +- 2 | 121.91 +- 4 | 15.66 | 19.69 | 0.65 | 0.82 |
| 32 | 127.04 +- 1 | 101.68 +- 3 | 18.89 | 23.6 | 0.59 | 0.74 |




### 1.3.2 Modelo manager - worker

Aplicando este modelo de comunicación se pretende conseguir un reparto de las tareas más eficiente. Se trata de un reparto dinámico de las tareas, inicialmente, el proceso 0 leía el fichero de configuraciones e inicializaba las mallas y las enviaba al resto de procesos. Ahora el proceso manager lee el fichero de configuraciones y envía configuraciónes a los procesos *pid_w* 0 de cada grupo de workers, luego cada uno de estos se encarga de inicializar su configuración y distribuir la malla entre los trabajadores de su grupo.

#### 1.3.2.1 Tipo de dato Chip

Para enviar y recibir las configuraciones, necesitabamos una manera sencilla de enviar la información de los microchips. El siguiente código crea un tipo de dato struct, igualito al del chip para utilizarlo en las funciones de comunicación.

```
/* Creation of chip data type */
MPI_Datatype mpi_chip_type;
MPI_Datatype types[nitems];
MPI_Aint offsets[nitems];

for(j=0; j<nitems; j++)
            blocklengths[j]=1;
offsets[0] = offsetof(struct info_chip, x);
         types[0] = MPI_INT;
offsets[1] = offsetof(struct info_chip, y);
         types[1] = MPI_INT;
offsets[2] = offsetof(struct info_chip, h);
         types[2] = MPI_INT;
offsets[3] = offsetof(struct info_chip, w);
         types[3] = MPI_INT;
offsets[4] = offsetof(struct info_chip,
    tchip); types[4] = MPI_FLOAT;

MPI_Type_create_struct(nitems, &
    blocklengths[0], &offsets[0], &types
    [0], &mpi_chip_type);
MPI_Type_commit(&mpi_chip_type);
```

#### 1.3.2.2 Creación de los comunicadores

En este programa creamos 4 grupos de comunicadores que es sencillamente adaptable ya que utilizamos la aritmética modular par asignar las claves dentro de cada comunicador.

En el siguiente código se crean los communicadores de los workers, se utiliza la función MPI_Comm_split obviando al proceso 0, por eso es necesario hacer *npr-1* en las operaciones para calcular el *color* y el *key*.

Es una condición que el número de procesadores sea múltiplo de 4 + 1 y mayor o igual a 9 para que funcione correctamente. De ser cinco, el programa se ejecutaría en serie en cada proceso trabajador y lo que queremos es probar el programa paralelo que hemos creado.

```
/* Creation of worker communicators */
int color,key,pid_w,npr_w;
int groupSize = (npr-1)/groupAmount;

if( pid == 0){
   color = MPI_UNDEFINED; key = 0;
}else{
   color = 1 + floor((pid-1)/groupSize);
```

```
    key =  (pid-1) % groupSize;
}
MPI_Comm_split(MPI_COMM_WORLD, color, key,
    &worker_comm);
```

#### 1.3.2.3 El Mánager

Como ya hemos indicado el mánager se encarga de leer y distribuir bajor de demanda las configuraciones, también calcula el tiempo global de ejecución.

Este es pseudocódigo del proceso mánager, puede verse el código original en *Anexos 3.2.1.*

```
get Time 0
Send one configuration to each worker 0
        responses ++
while waiting responses
        Receive end of work message
        responses –
        if configurations
                Send work message
                Send confuration
                responses++
        else
                Send end of work message

get Time 1
All reduce workers 0 for best configuration
Receive best configuration from best owner
```

#### 1.3.2.4 El Worker

El worker se parece mucho al código básico utilizado hasta ahora, el anterior proceso 0 ahora es el proceso 0 de los workers y es quien recibe la configuración inicializa la placa y la distribuye.

Este es pseudocódigo del proceso worker, puede verse el código original en *Anexos 3.2.1.*

```
if worker 0
        Allocate memory for one configuration
        Receive work message from manager
0 broadcast work message to rest
while work = 0
        if worker = 0
                Receive configuration
                Init grid chips
        Init grid
       0 Scatter grid chips
        difussion
        Gather grid in 0
        if worker 0
                process results
                Request for work to manager
                Receive manager answer
        0 broadcast work message to rest
if worker 0
        All reduce workers 0 for best configuration
        if Best
                Send best configuration to manager
```

#### 1.3.2.5 Otros cambios en el código

Se ha añadido un núevo parámetro a la función difusión, un parámetro MPI_Comm workers_comm, ahora la función difusión ya no utiliza MPI_COMM_WORLD como comunicador de sus funciones sino el recogido por parámetro.

Se ha eliminado el vector *Tej* que contenía los tiempos de ejecución de cada configuración, ahora contamos solamente el tiempo desde el envío de la primera configuración hasta la recepción de la última respuesta.

#### 1.3.2.6 Pruebas del nuevo modelo

La siguiente tabla muestra los resultados obtenidos siguiendo el procedimiento descrito en el apartado 1.2.5.

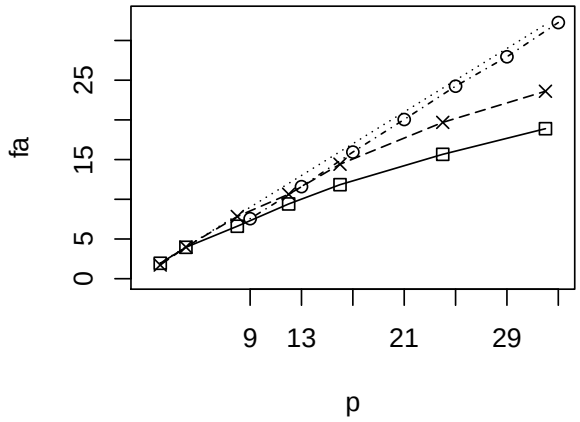
| **Proc.** | **T**(s) | **speed-up** | **efficiency** |
|---|---|---|---|
| serial | 2400 | - | - |
| 9 | 317.61 +- 0 | 7.556 | 0.84 |
| 13 | 207.03 +- 0 | 11.593 | 0.89 |
| 17 | 150.49 +- 0 | 15.948 | 0.94 |
| 21 | 119.71+- 0 | 20.049 | 0.95 |
| 25 | 99 +- 1 | 24.243 | 0.97 |
| 29 | 85.87 +- 0 | 27.948 | 0.96 |
| 33 | 74.4 +- 0 | 32.258 | 0.98 |

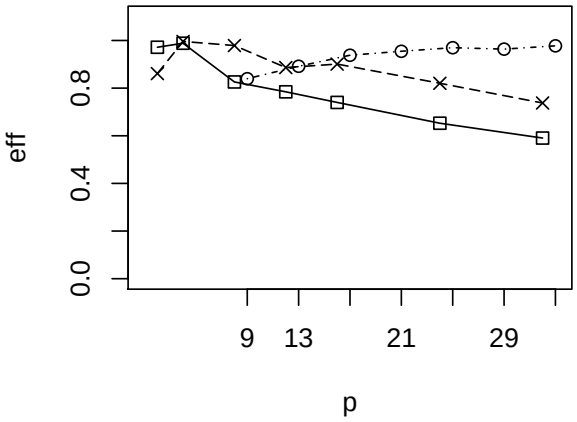
Los resultados de esta última muestran que el nuevo modelo es muy superior al anterior ya que su eficiencia aumenta con P y alcanza factores de aceleración casi óptimos. Además podemos intuir de los valores de la eficiencia que alcanza su máximo o se acerca a el con los 33 procesadores(4 grupos de 8), esto coincide con los resultados anteriores donde la eficiencia era máxima con 8 procesadores, debido al problema de la frontera y al tamaño de la placa.

Esto significa que si desearamos escalar el programa, añadiendo muchas más configuraciones y utilizando muchos procesadores, habría que crear más grupos pero mantener el tamaño de grupo en 8. Si desearamos aumentar el grano de la malla, crearíamos grupos de más procesadores.

Por otro lado, el nuevo modelo pierde eficiencia con pocos procesadores, no es hasta 13 procesadores más o menos donde comienza a superar al resto. Además, debido a la restrición del tamaño de grupos de procesos (2 procesos mínimo por grupo) implica disponer al menos de 9 procesadores para ejecutarlo.

En la tabla siguiente, se muestran las eficiencias obtenidas con distintos números de grupos y distintos tamaños de grupos, esperabamos probar que efectivamente los grupos de tamaño 8 deberían ser los más eficientes. Si que se puede apreciar en el caso de 2 grupos, como la eficiencia disminuye (aunque poco) al subir a 16 procesos por grupo, en el resto de casos la eficiencia no deja de crecer, necesitaríamos de más procesos para demostrar que los grupos de 8 son los óptimos.




| **Proc. 2 groups** | **efficiency** | **Proc. 3 groups** | **efficiency** | **Proc. 5 groups** | **efficiency** |
|---|---|---|---|---|---|
| 1 + 3*2 | 0.85 | 1 + 2*3 | 0.82 | 1 + 2*5 | 0.83 |
| 1 + 6*2 | 0.92 | 1 + 3*3 | 0.86 | 1 + 3*5 | 0.86 |
| 1 + 8*2 | 0.94 | 1 + 4*3 | 0.9 | 1 + 4*5 | 0.9 |
| 1 + 10*2 | 0.97 | 1 + 5*3 | 0.93 | 1 + 5*5 | 0.94 |
| 1 + 12*2 | 0.97 | 1 + 6*3 | 0.94 | 1 + 6*5 | 0.94 |
| 1 + 15*2 | 0.96 | 1 + 7*3 | 0.89 | - | - |
| 1 + 16*2 | 0.95 | 1 + 8*3 | 0.97 | - | - |
| - | - | 1 + 9*3 | 0.97 | - | - |
| - | - | 1 + 10*3 | 0.97 | - | - |

## 1.4 Código primera fase

### 1.4.1 *heats.c*

```
/* File: heats.c */

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

#include "defines.h"
#include "faux.h"
#include "difussion.h"

// global variables
float grid_chips[MAX_GRID_POINTS], grid[MAX_GRID_POINTS], grid_aux[MAX_GRID_POINTS],
    grid_chips_aux[MAX_GRID_POINTS], grid_sol_aux[MAX_GRID_POINTS];
struct temp BT;

/*****************************************************************************/
void init_grid_chips (int conf, struct info_param param, 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=param.chips[conf][n].x*param.scale; i<(param.chips[conf][n].x+param.chips[conf][n].h)
      *param.scale; i++)
  for (j=param.chips[conf][n].y*param.scale; j<(param.chips[conf][n].y+param.chips[conf][n].w)
      *param.scale; j++)
    grid_chips[(i+1)*NCOL+(j+1)] = param.chips[conf][n].tchip;
}

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

        for (i=0; i<= nrows+1; i++)
            for (j=0; j<NCOL; j++)
                grid[i*NCOL+j] = grid_aux[i*NCOL+j] = param.t_ext;
}

/*****************************************************************************/
void calculate_chop_points (struct info_param param,  int *displacement, int *size, int npr){
 /* Compute the division and remainder of rows */
        int reparto = (NROW - 2)/npr;
        int remainder = ((NROW - 2) % npr);
        int i;

        /* Initializations of size and displacement */
        displacement[0] = 0;
        if(remainder>0){
            size[0] = (reparto+1)*NCOL;
            remainder --;
        }else{
            size[0] = reparto*NCOL;
        }

        for(i=1; i<npr; i++){
            if(remainder>0){
                size[i] = (reparto+1)*NCOL;
```

```
                remainder --;
            }else{
                size[i] = reparto*NCOL;
            }
            displacement[i] = displacement[i-1]+size[i-1];
        }
}

/*****************************************************************************/
int main (int argc, char *argv[]){
  int    conf, nconf, pid, npr, i, j;
  struct info_param param;
  struct timeval t0, t1;
  double *tej, tsim = 0.0;
  int pos=0;
  int sizebuf=1024;
  char buf[sizebuf];

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

  if( pid == 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);

     printf ("\n  ======================================================================");
     printf ("\n    Thermal difussion - Paralel 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));

     /* EMPAQUETAR */
    MPI_Pack(&param.scale, 1, MPI_INT, buf, sizebuf, &pos, MPI_COMM_WORLD);
    MPI_Pack(&param.nconf, 1, MPI_INT, buf, sizebuf, &pos, MPI_COMM_WORLD);
    MPI_Pack(&param.nchip, 1, MPI_INT, buf, sizebuf, &pos, MPI_COMM_WORLD);
    MPI_Pack(&param.t_ext, 1, MPI_FLOAT, buf, sizebuf, &pos, MPI_COMM_WORLD);
    MPI_Pack(&param.t_delta, 1, MPI_FLOAT, buf, sizebuf, &pos, MPI_COMM_WORLD);
    MPI_Pack(&param.max_iter, 1, MPI_INT, buf, sizebuf, &pos, MPI_COMM_WORLD);
  }

  /* Unpacking necesary parameters */
  MPI_Bcast(&buf[0], sizebuf, MPI_PACKED, 0, MPI_COMM_WORLD);

  if(pid!=0){
     MPI_Unpack(buf, sizebuf, &pos, &param.scale,   1, MPI_INT, MPI_COMM_WORLD);
     MPI_Unpack(buf, sizebuf, &pos, &param.nconf,   1, MPI_INT, MPI_COMM_WORLD);
     MPI_Unpack(buf, sizebuf, &pos, &param.nchip,   1, MPI_INT, MPI_COMM_WORLD);
     MPI_Unpack(buf, sizebuf, &pos, &param.t_ext,   1, MPI_FLOAT, MPI_COMM_WORLD);
     MPI_Unpack(buf, sizebuf, &pos, &param.t_delta,     1, MPI_FLOAT, MPI_COMM_WORLD);
     MPI_Unpack(buf, sizebuf, &pos, &param.max_iter, 1, MPI_INT, MPI_COMM_WORLD);
  }

  //Declaration of size and displacement of individual grids
```

```
  int displacement[npr];
  int size[npr];
  calculate_chop_points(param, &displacement[0], &size[0], npr);
  int nrows = size[pid]/NCOL;

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

    init_grids(param, grid, grid_aux, nrows);

    /* Scattering of grid chips among the processes */
    MPI_Scatterv(&grid_chips_aux[NCOL], &size[0], &displacement[0],  MPI_FLOAT,
                  &grid_chips[NCOL], size[pid], MPI_FLOAT, 0, MPI_COMM_WORLD);

    /* main loop: thermal injection/disipation until convergence (t_delta or max_iter) */
    diffusion (param, &grid[0], &grid_chips[0], &grid_aux[0], nrows, npr, pid);

    /* Gathering of grid */
    MPI_Gatherv(&grid[NCOL], size[pid],MPI_FLOAT, &grid_sol_aux[NCOL],
                       &size[0], &displacement[0], MPI_FLOAT, 0, MPI_COMM_WORLD);

    if(pid == 0){
        // 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_sol_aux[0],&grid_chips_aux[0], &BT);
    }
  }
  if(pid == 0){
      // writing best configuration results
      results (param, &BT, argv[1]);
      for (conf=0; conf<param.nconf; conf++) tsim += tej[conf];
      printf ("   > Time (Paralel %d) : %1.3f s \n\n",npr, tsim);
  }
  //MPI Finalisation
  MPI_Finalize();
  return(0);
}
```

### 1.4.2 *difussion.c*

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

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

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

    // heat injection at chip positions
    for (i=1; i<=nrows; 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<=nrows; 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
    nrows, int npr, int pid){
    int    i, j, end, niter,flag = 0;
    float  T;
    double Tmean, tmean, Tmean0 = param.t_ext;
    MPI_Status info;

    MPI_Status stats[4];
    MPI_Request reqs[4] ;

    end = 0; niter = 0;

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

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

        if(pid > 0){
            MPI_Irecv(&grid[0], NCOL, MPI_FLOAT, pid-1, 0, MPI_COMM_WORLD,&reqs[0]);
        }
        if(pid < npr -1 ){
            MPI_Irecv(&grid[NCOL*(nrows+1)], NCOL, MPI_FLOAT, pid+1, 0, MPI_COMM_WORLD, &reqs
                [1]);
        }
        if(pid > 0){
            MPI_Isend(&grid[NCOL], NCOL, MPI_FLOAT, pid-1 , 0, MPI_COMM_WORLD,&reqs[2]);
        }
        if(pid < npr - 1){
            MPI_Isend(&grid[NCOL*nrows], NCOL, MPI_FLOAT, pid+1 , 0, MPI_COMM_WORLD,&reqs[3]);
        }

        // thermal difussion
        for (i=2; i<nrows; 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;
        }

        if( (pid > 0) && (pid < npr -1) ){

            MPI_Testall(4,reqs, &flag, stats);
            if(!flag){MPI_Waitall(4,reqs,stats);}

        }else if( pid == 0 ){

            MPI_Test(&reqs[1], &flag, &stats[1]);
            if(!flag){MPI_Wait(&reqs[1], &stats[1]);}
            MPI_Test(&reqs[3], &flag, &stats[3]);
            if(!flag){MPI_Wait(&reqs[3], &stats[3]);}
        }else{

            MPI_Test(&reqs[0], &flag, &stats[0]);
            if(!flag){MPI_Wait(&reqs[0], &stats[0]);}
            MPI_Test(&reqs[2], &flag, &stats[2]);
            if(!flag){MPI_Wait(&reqs[2], &stats[2]);}
            i = nrows;
        }

        // Finish with first and last lines
        for (i=1; i<=nrows; i=i+nrows-1)
            if( ((pid != 0) && (pid != npr-1)) || (((pid == 0)&&(i==nrows)) || ((pid == npr-1)
                &&(i==1))) )
            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;
        }

        //Update values of the grid
        for (i=1; i<=nrows; 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){

            MPI_Allreduce(&Tmean, &tmean, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

            tmean = tmean / ((NCOL-2)*(NROW-2));
            if ((fabs(tmean - Tmean0) < param.t_delta) || (niter > param.max_iter))
                end = 1;
            else
                Tmean0 = tmean;
        }
    }
}
```

## 1.5 Código segunda fase

### 1.5.1 *heats.c*

```
/* File: heats.c */

    #include <stdio.h>
    #include <stddef.h>
    #include <values.h>
    #include <sys/time.h>
    #include <mpi.h>

    #include "defines.h"
    #include "faux.h"
    #include "difussion.h"

    float grid_chips[MAX_GRID_POINTS], grid[MAX_GRID_POINTS], grid_aux[MAX_GRID_POINTS],
        grid_chips_aux[MAX_GRID_POINTS], grid_sol_aux[MAX_GRID_POINTS];
    struct temp BT;

    /*****************************************************************************/
    void init_grid_chips (int conf, struct info_param param, 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=param.chips[conf][n].x*param.scale; i<(param.chips[conf][n].x+param.chips[
                conf][n].h)*param.scale; i++)
                for (j=param.chips[conf][n].y*param.scale; j<(param.chips[conf][n].y+param.
                    chips[conf][n].w)*param.scale; j++)
                    grid_chips[(i+1)*NCOL+(j+1)] = param.chips[conf][n].tchip;
    }
    /*****************************************************************************/
    void init_grids (struct info_param param, float *grid, float *grid_aux, int nrows){
        int i, j;

        for (i=0; i<= nrows+1; i++)
            for (j=0; j<NCOL; j++)
                grid[i*NCOL+j] = grid_aux[i*NCOL+j] = param.t_ext;
    }
    /*****************************************************************************/
    void calculate_chop_points (struct info_param param,  int *displacement, int *size, int
        npr){
        /* Compute the division and remainder of rows */
        int reparto = (NROW - 2)/npr;
        int remainder = ((NROW - 2) % npr);
        int i;

        /* Initializations of size and displacement */
        displacement[0] = 0;
        if(remainder>0){
            size[0] = (reparto+1)*NCOL;
            remainder --;
        }else{
            size[0] = reparto*NCOL;
        }

        for(i=1; i<npr; i++){
            if(remainder>0){
                size[i] = (reparto+1)*NCOL;
                remainder --;
            }else{
                size[i] = reparto*NCOL;
```

```
            }
            displacement[i] = displacement[i-1]+size[i-1];
        }
    }
    /*****************************************************************************/
    int main (int argc, char *argv[]){
        int    conf, nconf, groupAmount, pid, npr, i, j,msg=0;
        struct info_param param;
        struct timeval t0, t1;
        double timeTot=0.0;

        int pos=0;
        int sizebuf=1024;
        char buf[sizebuf];
        const int nitems=5;
        int blocklengths[nitems];
        BT.Tmean = MAXDOUBLE;
        MPI_Status info;
        MPI_Comm worker_comm,worker_leaders_comm;
        MPI_Group worker_leaders,all_nodes;

        MPI_Datatype mpi_chip_type;
        MPI_Datatype types[nitems];
        MPI_Aint offsets[nitems];

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

        if( pid == 0){
            /* reading initial data file */
            if (argc != 3) {
                printf ("\n\n ERROR: needs a card description file \n\n");
                printf ("\n\n ERROR: needs a group size parameter \n\n");
                exit (-1);
            }

            read_data (argv[1], &param);

            char *p;
            int errno = 0;
            long conv = strtol(argv[2], &p, 10);

            // or the integer is larger than int
            if (errno != 0 || *p != '\0' || conv > INT_MAX) {
                printf ("\n\n ERROR: needs a valid group size parameter \n\n");
                exit (-1);
            } else {
                groupAmount = conv;
            }

            printf ("\n  ======================================================================")
                ;
            printf ("\n    Thermal difussion - Parallel Version ");
            printf ("\n  Groups %d  %d x %d points, %d chips", groupAmount, 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");


            /* EMPAQUETAR */
            MPI_Pack(&param.scale, 1, MPI_INT, buf, sizebuf, &pos, MPI_COMM_WORLD);
            MPI_Pack(&param.nconf, 1, MPI_INT, buf, sizebuf, &pos, MPI_COMM_WORLD);
```

```
        MPI_Pack(&param.nchip, 1, MPI_INT, buf, sizebuf, &pos, MPI_COMM_WORLD);
        MPI_Pack(&param.t_ext, 1, MPI_FLOAT, buf, sizebuf, &pos, MPI_COMM_WORLD);
        MPI_Pack(&param.t_delta, 1, MPI_FLOAT, buf, sizebuf, &pos, MPI_COMM_WORLD);
        MPI_Pack(&param.max_iter, 1, MPI_INT, buf, sizebuf, &pos, MPI_COMM_WORLD);
        MPI_Pack(&groupAmount, 1, MPI_INT, buf, sizebuf, &pos, MPI_COMM_WORLD);
    }

     /* Unpacking necesary parameters */
     MPI_Bcast(&buf[0], sizebuf, MPI_PACKED, 0, MPI_COMM_WORLD);
     if(pid!=0){
        MPI_Unpack(buf, sizebuf, &pos, &param.scale,    1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(buf, sizebuf, &pos, &param.nconf,    1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(buf, sizebuf, &pos, &param.nchip,    1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(buf, sizebuf, &pos, &param.t_ext,    1, MPI_FLOAT, MPI_COMM_WORLD);
        MPI_Unpack(buf, sizebuf, &pos, &param.t_delta,  1, MPI_FLOAT, MPI_COMM_WORLD);
        MPI_Unpack(buf, sizebuf, &pos, &param.max_iter, 1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(buf, sizebuf, &pos, &groupAmount, 1, MPI_INT, MPI_COMM_WORLD);
     }

     /* Creation of chip data type */
     for(j=0; j<nitems; j++)
         blocklengths[j]=1;
     offsets[0] = offsetof(struct info_chip, x);     types[0] = MPI_INT;
     offsets[1] = offsetof(struct info_chip, y);     types[1] = MPI_INT;
     offsets[2] = offsetof(struct info_chip, h);     types[2] = MPI_INT;
     offsets[3] = offsetof(struct info_chip, w);     types[3] = MPI_INT;
     offsets[4] = offsetof(struct info_chip, tchip); types[4] = MPI_FLOAT;

     MPI_Type_create_struct(nitems, &blocklengths[0], &offsets[0], &types[0], &
         mpi_chip_type);
     MPI_Type_commit(&mpi_chip_type);

    /* Creation of worker communicators */
    int color,key,pid_w,npr_w;
    int groupSize = (npr-1)/groupAmount;
    if( pid == 0){
        color = MPI_UNDEFINED; key = 0;
    }else{
        color = 1 + floor((pid-1)/groupSize);
        key =  ((pid-1) % groupSize);
    }
    MPI_Comm_split(MPI_COMM_WORLD, color, key, &worker_comm);
    if(pid > 0){
        MPI_Comm_rank( worker_comm, &pid_w) ;
        MPI_Comm_size ( worker_comm, &npr_w) ;
    }

    /* Creation of communicator for processes ranked as 0 */
    int npr_w0,pid_w0,l;
    if( (pid % (groupSize) == 1) || (pid == 0)){
        color=1;
        if(pid==0){
            key =0;
        }else{
            key= floor(pid/groupSize)+1;
        }
    }else{
        color=MPI_UNDEFINED;
        key=pid;
    }
    MPI_Comm_split(MPI_COMM_WORLD, color, key, &worker_leaders_comm);;
    if( (pid % (groupSize) == 1) || (pid == 0)){
        MPI_Comm_size(worker_leaders_comm, &npr_w0);
        MPI_Comm_rank(worker_leaders_comm, &pid_w0);
    }
```

```
        /* Manager process code */
        if(pid == 0){
            gettimeofday (&t0, 0);
            int responses = 0;
            /* Initially send one configuration to each workers leader */
            for(i=0; i<groupAmount; i++){
                if(i < param.nconf){
                    conf = i;
                    /* Work message  tag = 0*/
                    MPI_Send(&conf,1,MPI_INT, i+1,0,worker_leaders_comm);
                    /* Send chips data */
                    MPI_Send(&param.chips[i][0], param.nchip, mpi_chip_type, i+1 , 0,
                        worker_leaders_comm);
                    responses++;
                }else{
                    /* End of work message  tag = 1*/
                    MPI_Send(&msg,1,MPI_INT,i+1,1,worker_leaders_comm);
                }
            }
            /* Until enf work */
            while(responses>0){
                /* Receive work request */
                MPI_Recv(&conf, 2, MPI_DOUBLE, MPI_ANY_SOURCE, 0, worker_leaders_comm, &info);
                responses--;
                if(i < param.nconf){
                    MPI_Send(&i,1,MPI_INT,info.MPI_SOURCE,0,worker_leaders_comm);
                    MPI_Send(&param.chips[i][0], param.nchip, mpi_chip_type, info.MPI_SOURCE ,
                         0, worker_leaders_comm);
                    responses++;
                    i++;
                }else{
                    MPI_Send(&i,1,MPI_INT,info.MPI_SOURCE,1,worker_leaders_comm);
                }
            }
            gettimeofday (&t1, 0);
            timeTot = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec)/1e6;
            /* Find best configuration */
            double tmin=MAXDOUBLE;
            /* Gather best configuration results */
            MPI_Allreduce(&BT.Tmean, &tmin, 1, MPI_DOUBLE, MPI_MIN, worker_leaders_comm);

            printf ("   > Time (Paralel %d) : %1.3f s \n",npr, timeTot);

        }else{
            /* Distribution of problem domain*/
            int displacement[npr_w];
            int size[npr_w];
            calculate_chop_points(param, &displacement[0], &size[0], npr_w);
            int nrows = size[pid_w]/NCOL;
            int work=1;

            /* Worker leader process code */
            if(pid_w == 0){
                /* Allocate memmory for one chip configuration */
                param.chips = (struct info_chip **) malloc(1*sizeof(struct info_chip*));
                param.chips[0] = (struct info_chip *) malloc(param.nchip * sizeof(struct
                    info_chip));

                /* Receive work message and with configuration number */
                MPI_Recv(&conf, 1, MPI_INT, 0, MPI_ANY_TAG, worker_leaders_comm, &info);//
                    Recibe trabajo
                if(info.MPI_TAG == 0){
                    work = 0;
                }else{
                    work = 1;
                }
```

```
            }
            /* Broadcast work message*/
            MPI_Bcast(&work, 1, MPI_INT, 0, worker_comm);
            while(work == 0){
                if(pid_w == 0){
                    /* Receive configuration */
                    MPI_Recv(&param.chips[0][0], param.nchip, mpi_chip_type, 0, 0,
                        worker_leaders_comm, &info);

                    init_grid_chips (0, param, grid_chips_aux);
                }
                init_grids(param, grid, grid_aux, nrows);

                /* Scattering of grid chips among the processes */
                MPI_Scatterv(&grid_chips_aux[NCOL], &size[0], &displacement[0],  MPI_FLOAT,
                    &grid_chips[NCOL], size[pid_w], MPI_FLOAT, 0, worker_comm);

                /* main loop: thermal injection/disipation until convergence (t_delta or
                    max_iter) */
                diffusion (param,worker_comm, &grid[0], &grid_chips[0], &grid_aux[0], nrows,
                    npr_w, pid_w);

                /* Gathering of grid */
                MPI_Gatherv(&grid[NCOL], size[pid_w],MPI_FLOAT, &grid_sol_aux[NCOL],
                        &size[0], &displacement[0], MPI_FLOAT, 0, worker_comm);

                if(pid_w==0){
                    /* processing configuration results */
                    results_conf (conf, param, &grid_sol_aux[0], &grid_chips_aux[0], &BT);

                    /* Request for work */
                    MPI_Send(&conf,1,MPI_DOUBLE,0,0,worker_leaders_comm);
                    /* Receive work message */
                    MPI_Recv(&conf, 1, MPI_INT, 0, MPI_ANY_TAG, worker_leaders_comm, &info);//
                        Recibe trabajo
                    if(info.MPI_TAG == 0){
                        work = 0;
                    }else{
                        work = 1;
                    }
                }
                MPI_Bcast(&work, 1, MPI_INT, 0, worker_comm);
            }
            if(pid_w==0){
                double tmin=MAXDOUBLE;
                /* Gather best configuration results */
                MPI_Allreduce(&BT.Tmean, &tmin, 1, MPI_DOUBLE, MPI_MIN, worker_leaders_comm);

                if(BT.Tmean == tmin){
                    results(param, &BT, argv[1]);
                    printf ("\n\n >>> BEST CONFIGURATION: %2d\t Tmean: %1.2f\n\n", BT.conf, BT
                        .Tmean);
                }
            }
        }

        //MPI Finalisation
        MPI_Finalize();
        return(0);
    }
```

### 1.5.2 *difussion.c*

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

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

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

    // heat injection at chip positions
    for (i=1; i<=nrows; 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<=nrows; 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, MPI_Comm worker_comm, float *grid, float *grid_chips,
     float *grid_aux, int nrows, int npr, int pid){
    int    i, j, end, niter,flag = 0;
    float  T;
    double Tmean, tmean, Tmean0 = param.t_ext;
    MPI_Status info;

    MPI_Status stats[4];
    MPI_Request reqs[4] ;

    end = 0; niter = 0;

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

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

        if(pid > 0){
            MPI_Irecv(&grid[0], NCOL, MPI_FLOAT, pid-1, 0, worker_comm,&reqs[0]);
        }
        if(pid < npr -1 ){
            MPI_Irecv(&grid[NCOL*(nrows+1)], NCOL, MPI_FLOAT, pid+1, 0, worker_comm, &reqs[1])
                ;
        }
        if(pid > 0){
            MPI_Isend(&grid[NCOL], NCOL, MPI_FLOAT, pid-1 , 0, worker_comm,&reqs[2]);
        }
        if(pid < npr - 1){
            MPI_Isend(&grid[NCOL*nrows], NCOL, MPI_FLOAT, pid+1 , 0, worker_comm,&reqs[3]);
        }

        // thermal difussion
        for (i=2; i<nrows; 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;
        }

        if( (pid > 0) && (pid < npr -1) ){

            MPI_Testall(4,reqs, &flag, stats);
            if(!flag){MPI_Waitall(4,reqs,stats);}

        }else if( pid == 0 ){

            MPI_Test(&reqs[1], &flag, &stats[1]);
            if(!flag){MPI_Wait(&reqs[1], &stats[1]);}
            MPI_Test(&reqs[3], &flag, &stats[3]);
            if(!flag){MPI_Wait(&reqs[3], &stats[3]);}
        }else{

            MPI_Test(&reqs[0], &flag, &stats[0]);
            if(!flag){MPI_Wait(&reqs[0], &stats[0]);}
            MPI_Test(&reqs[2], &flag, &stats[2]);
            if(!flag){MPI_Wait(&reqs[2], &stats[2]);}
            i = nrows;
        }

        // Finish with first and last lines
        for (i=1; i<=nrows; i=i+nrows-1)
            if( ((pid != 0) && (pid != npr-1)) || (((pid == 0)&&(i==nrows)) || ((pid == npr-1)
                &&(i==1))) )
            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;
        }

        //Update values of the grid
        for (i=1; i<=nrows; 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){

            MPI_Allreduce(&Tmean, &tmean, 1, MPI_DOUBLE, MPI_SUM, worker_comm);

            tmean = tmean / ((NCOL-2)*(NROW-2));
            if ((fabs(tmean - Tmean0) < param.t_delta) || (niter > param.max_iter))
                end = 1;
            else
                Tmean0 = tmean;
        }
    }
}
```

# 2 Puzle

## 2.1 Comunicaciones colectivas

En la comunicación entre un grupo de individuos, existen mecanismos que permiten una comunicación más eficiente. En este trabajo en el apartado 1.1 *Comunicación en forma de árbol* se muestran varios modelos de comunicación compuestos por árboles.

En MPI, la comunicación colectiva consiste en una serie de funciones que sirven para que un grupo de procesos se comuniquen entre ellos. En el apartado 1.2 *Comunicaciones colectivas frente a comunicaciones Punto a Punto* se muestran las diferencias principales entre las funciones de comunicación colectiva y las de punto a punto. En el apartado 1.3 *Funciones de Comunicación Colectiva en MPI* se describen las funciones más utilizadas y sus parámetros.

### 2.1.1 Comunicación en forma de árbol

En [1] se describe esta forma de comunicación, mostrando como ejemplo un problema de suma *one-to-all*. Aunque inicialmente pueda parecer que no mejora demasiado, ya que la mitad de los nodos realizan las mismas comunicaciones que realizarían punto a punto(esto es, cuando todos los nodos envían su valor a un único nodo), mejora considerablemente reduciendo la cantidad de recepciones y sumas que tiene que realizar el nodo líder.

En el ejemplo de la imagen, el nodo 0 pasa de realizar *n-1* recepciones y sumas a realizar 3 recepciones y sumas, esto es, la altura del árbol $log_2\ n (log_d\ n$ en el caso general, siendo $d$ el grado de árbol). De esta forma la carga de trabajo de los nodos crece logarítmica-mente con el número de procesadores, la mejora es notable frente al crecimiento lineal de la comunicación centralizada.

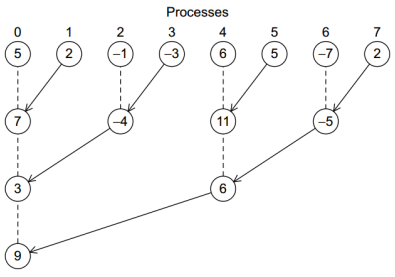



**FIGURE 3.6**
A tree-structured global sum

El mismo modelo invertido se puede utilizar para comunicación *one-to-all*.

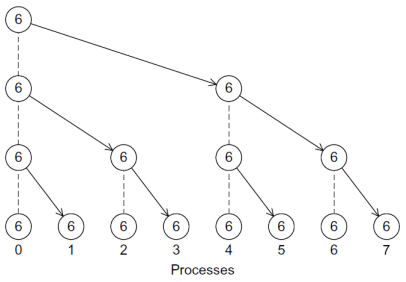



**FIGURE 3.10**
A tree-structured broadcast

Este último ejemplo muestra un modelo de comunicación all-to-all en el que se construyen n árboles siendo cada nodo raíz de uno de ellos.

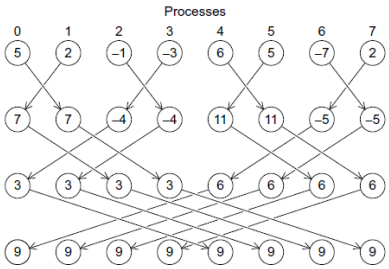



**FIGURE 3.9**
A butterfly-structured global sum

El problema de este modelo de comunicación es la dificultad de programarlo, existen incontables maneras distintas de hacerlo pero no sabríamos cual es la mejor, para que tipos y tamaños de problemas cual es la óptima. En [2] se sugiere también el uso de árboles de profundidad logarítmica para la comunicación, pero tampoco indican o desconocen si las funciones MPI hacen uso de ellos.

### 2.1.2 Comunicaciones colectivas frente a comunicaciones Punto a Punto

1. Todos los procesos en el comunicador deben llamar a la misma función colectiva independientemente del tipo de comunicación que se realice. Mientras en la comunicación punto a punto queda definido por la llamada quien es emisor y quien es receptor, MPI_Recv y MPI_Send.
2. Todos los procesos en el comunicador tienen que tener los parámetros compatibles, esto es, si en una llamada MPI_Reduce dos procesos tiene *root* distinto el programa va a fallar.
3. Las funciones de comunicación punto a punto se conectan mediante etiquetas (*tags*) y comunicadores, en las colectivas solamente mediante

comunicadores y el orden en el que son llamadas.

4. Otra restricción que las funciones de comunicación punto a punto no tienen es que la cantidad de datos enviados en el bufer tiene que coincidir exactamente con la esperada en el destino.

5. No exiten funciones de comunicación colectivas que no sean bloqueante, todas utilizan la función Barrier para asegurar las sincronización de los datos.

### 2.1.3 Funciones de Comunicación Colectiva en MPI

Podemos dividir las funciones de comunicación colectiva en función del tipo de comunicación que realizan y también en función del tipo de dato que manejan ya que así las distingue MPI.

- *all-to-one*:
  - MPI_REDUCE
  - MPI_GATHER
- *one-to-all*
  - MPI_BCAST
  - MPI_SCATTER
- *all-to-all*
  - MPI_ALLREDUCE
  - MPI_ALLGATHER
  - MPI_ALLTOALL
  - MPI_REDUCE_SCATTER
- *all-to-some*
  - MPI_SCAN

Respecto al tipo de dato que manejan, MPI dispone de las *vector variant* de las funciones vistas; MPI_GATHERV, MPI_SCATTERV, MPI_ALLGATHERV, MPI_ALLTOALLV.

Estas funciones permiten enviar una serie de datos de tamaño variable, se diferencian por tener un argumento extra llamado displs, que es un array de enteros que indica las posiciones de cada dato.

Respecto a las funciones ALL, en estas los parámetros son idénticos que los de sus semejantes con la excepción de que sobra el parámetro *root* ya que todos recibirán el mensaje.

La cabecera de una función colectiva puede tener los siguientes parámetros:

- buffer : Contiene la dirección de comienzo del buffer.
- count : Número de entradas en el buffer.
- datatype : Tipo de datos del buffer.
- root : Identificador del proceso raíz.
- comm : Comunicador MPI, representa a los procesos involucrados en la comunicación.

Las funciones Scatter y Gather utilizan dos buffers distintos.

- send buffer : Contiene la dirección de comienzo del buffer a enviar.
- recv buffer : Contiene la dirección de comienzo del buffer a recibir.
- send count : Número de entradas en send buffer.
- recv count : Número de entradas en recv buffer.
- displs : Vector de enteros con las posiciones de cada dato en el buffer.

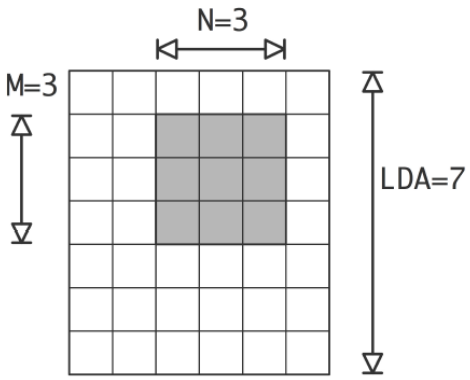
Es de especial interés el parámetro *op* de Reduce, que define el tipo de operación a realizar. Existen los siguientes tipos de operaciones:

| Operation Value | Meaning |
|---|---|
| MPI_MAX | Máximo |
| MPI_MIN | Mínimo |
| MPI_SUM | Suma |
| MPI_PROD | Producto |
| MPI_LAND | And lógico |
| MPI_BAND | And binario |
| MPI_LOR | Or lógico |
| MPI_BOR | Or binario |
| MPI_LXOR | Or exclusivo lógico |
| MPI_BXOR | Or exclusivo binario |
| MPI_MAXLOC | Máximo y su dirección |
| MPI_MINLOC | Mínimo y su dirección |

## 2.2 Tipos de datos derivados y comunicadores

Hasta ahora, antes de conocer el paralelismo de datos, se ha trabajado con tipos de datos primitivos (int, double, char, etc.). Estos son contiguos en memoria.

MPI ofrece un nuevo paradigma. Es posible que el programador desee enviar datos de carácter heterogéneo o no contiguos en memoria, MPI permite realizar este tipo de transferencias. Se puede ver en el siguiente ejemplo:




Las filas de una matriz no son contiguas en memoria, están separadas por un stride igual al número de columnas. Es decir, la segunda fila está en este caso a un stride 6 de la primera fila.

Si se desea acceder a los datos de la matriz coloreados en gris se puede hacer de una manera eficiente un nuevo tipo de dato. Los objetos irregulares son conocidos como tipos de datos derivados.

### 2.2.1 Tipos de datos elementales

MPI tiene un número de tipos de datos elementales, que se corresponden a los tipos simples de los lenguajes de programación. Los nombres se asemejan a los de C y Fortran. Así tenemos los tipos MPI_FLOAT y MPI_DOUBLE, y por otro lado los tipos MPI_REAL y MPI_DOUBLE_PRECISION. Hay que respetar su uso, no se puede usar MPI_FLOAT si programamos en Fortran ni MPI_REAL si trabajamos en C.

Las llamadas de MPI aceptan arrays de elementos, para enviar un único elemento hay que apuntar a su dirección. Hay dos problemas al usar únicamente tipos de datos elementales:

- Las rutinas de comunicación de MPI sólo pueden enviar elementos de un único tipo, homogéneos, aunque estén en posiciones de memoria contiguas. Se puede usar MPI_BYTE, pero no es recomendable.
- Tampoco es posible enviar elementos del mismo tipo si no están contiguos en memoria. Se puede enviar memoria contigua que contenga esos elementos (además de otros), pero supondría un gasto innecesario de ancho de banda.

### 2.2.2 Tipos de datos derivados

MPI permite crear tipos de datos propios, algo similar a definir una estructura en un lenguaje de programación, pero no del todo. Son especialmente útiles cuando se trata de enviar múltiples elementos en un mismo mensaje.

Los tipos de datos derivados permiten solucionar los problemas anteriormente comentados de diferentes formas:

1. Se puede crear un tipo de dato contiguo que consiste en un array de elementos de distintos tipos de datos. No hay diferencia entre enviar un elemento de un tipo o múltiples de distinto tipo.
2. Se puede crear un tipo de datos vector que consista en bloques espaciados de forma regular de elementos de un mismo tipo. Es una solución al problema de no poder enviar datos no contiguos.
3. Para los datos no contiguos espaciados de forma irregular existe el tipo de dato indexado. Se trata de un array que contiene las ubicaciones de los bloques. Los bloques pueden ser de distinto tamaño.
4. El tipo de datos struct puede albergar múltiples tipos de datos.

Todos estos mecanismos se pueden combinar para obtener tipos datos de tipo heterogéneo espaciados de forma irregular.

#### 2.2.2.1 Datatype signatures

Con los tipos de datos primitivos, si el emisor envía un array de enteros, el receptor tiene que declarar el tipo de dato también como enteros. Con los tipos de datos derivados ya no ocurre esto. El emisor y el receptor pueden declarar diferentes tipos de datos

siempre y cuando tengan el mismo "datatype signature".

La firma del tipo de dato es la representación interna del tipo de dato. Por ejemplo, si el emisor declara un tipo consistente en 2 enteros y envía 4 elementos de ese tipo, el receptor puede recibir dos elementos de un tipo que consista en 4 enteros. En ambos casos se envían 8 enteros y se reciben 8 enteros.

#### 2.2.2.2 Basic calls

Los nuevos tipos de datos se crean con:

- MPI_Type_contiguous
- MPI_Type_create_subarray
- MPI_Typevector
- MPI_Type_struct
- MPI_Type_indexed
- MPI_Type_hindexed

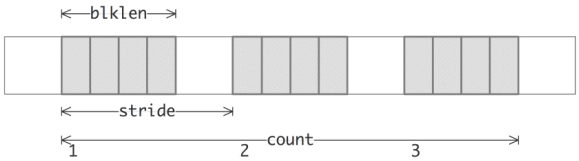
Es necesario llamar a MPI_Type_commit, el cual se encarga de que MPI haga los cálculos de indexación para los tipos de datos. Cuando no se necesite ya más hay que llamar a MPI_Type_free.

#### 2.2.2.3 Tipo contiguo

Es el tipo derivado más simple. Define un array de elementos de un tipo elemental o de otro tipo definido con anterioridad. No hay diferencia entre enviar un elemento de tipo contiguo o múltiples elementos del tipo que lo componen.

#### 2.2.2.4 Tipo vector

Es el tipo de dato no contiguo más simple. Describe una serie de bloques, todos de la misma longitud, y espaciados con un mismo stride, constante.

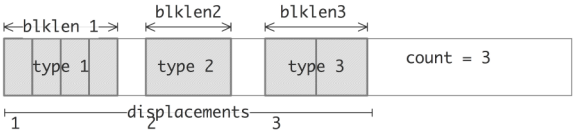



#### 2.2.2.5 Tipo indexado

El tipo indexado puede enviar elementos ubicados arbitrariamente de un array de un tipo único. Para ello hay que proporcionar un array de posiciones, acompañado de un array de longitudes con un array separado con el tamaño de cada bloque.

#### 2.2.2.6 Tipo struct

El tipo struct puede contener múltiples tipos de datos. La especificación contiene un contador que indica cuantos bloques hay en una única estructura.

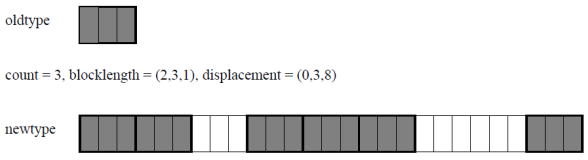



El tipo de estructura es muy similar en funcionalidad al tipo MPI_Type_hindexed, que usa un indexado basado en el Byte. El uso del tipo struct probablemente sea más limpio.

#### 2.2.2.7 Empaquetado

Una de las razones para usar tipos de datos derivados es poder tratar con datos no contiguos. Anteriormente esto sólo se podía hacer empaquetando los datos de su contenedor original en un buffer y desempaquetándolo en su receptor en sus estructuras de datos de destino.

MPI ofrece esa opción de empaquetado, parcialmente por la compatibilidad entre librerías, pero también por flexibilidad. A diferencia de los tipos de datos derivados, que transfieren los datos automáticamente, las rutinas de empaquetado añaden datos de forma secuencial en un buffer, y el empaquetado hay que hacerlo en orden.




## 2.3 Reparto dinámico de carga

### 2.3.1 Partición de los datos

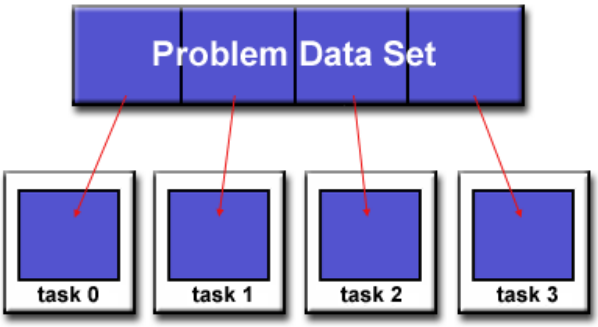
Uno de los primeros pasos a la hora de diseñar un programa paralelo es el de dividir el problema en bloques de trabajo discretos que puedan ser distribuidos

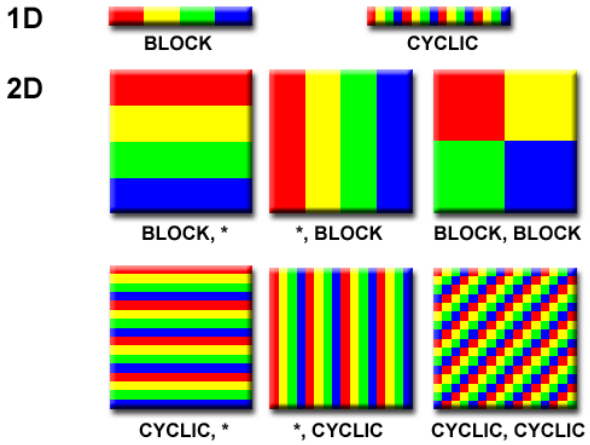
entre múltiples procesos.

Existen dos formas básicas de dividir los datos, la descomposición funcional y la descomposición del dominio.

### 2.3.2 Descomposición del dominio del problema

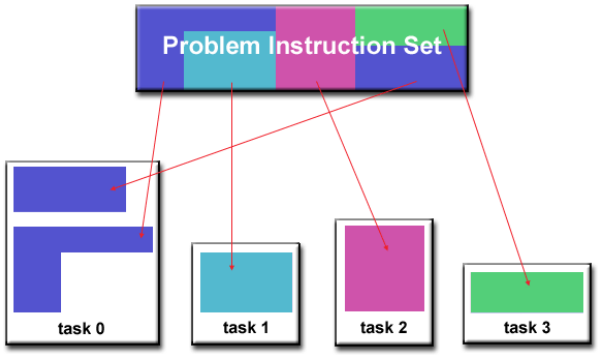
En este tipo de partición los datos se dividen entre los procesos y luego estos trabajan con ellos.




### 2.3.3 Descomposición funcional

La descomposición funcional no se centra tanto en los datos como en las cargas de trabajo del problema, de manera que cada proceso realiza un proporción equivalente del trabajo total.




### 2.3.4 Intercambio de las fronteras

En ciertos problemas, se requiere de datos que al dividir tocaron a otro procesador. El problema que se forma entonces es cuando un computador posee el valor p pero no el valor p+1. MPI nos ofrece dos métodos para gestionar dicho problema:

- Un procesador envía algo a otro procesador.
- Un procesador recibe algo de un origen.

### 2.3.5 Ejemplo de envío: Ping-Pong

Un procesador A envía algo a un procesador B y este se lo devuelve, el código que se ejecuta en A sería el siguiente:

```
MPI_Send( /* to: */ B     .);
MPI_Recv(/* from: */ B     .);
```

Mientras que en B se ejecutaría el siguiente:

```
MPI_Recv(/* from: */ A     .);
MPI_Send( /* to: */ A     .);
```

Si estamos trabajando en el modo SPMD el programa se mostraría así:

```
if ( /* I am process A */ ) {
    MPI_Send( /* to: */ B     .);
    MPI_Recv(/* from: */ B     .);
}
else if ( /* I am process B */ ) {
    MPI_Recv (/* from: */ A     .);
    MPI_Send ( /* to: */ A     .);
}
```

### 2.3.6 Comunicación bloqueante

El uso de MPI_Send y MPI_Recv bloqueará durante unos momentos la comunicación. Recv bloqueará al computador hasta recibir el dato deseado del MPI_Send.

El problema que se puede dar en la comunicación

bloqueante es el interbloqueo. Cuando dos computadores esperan que el otro les envíe algo pero jamás llega ya que el otro está bloqueado también en el MPI_Recv.

Una manera de evitar los interbloqueos es crear un grafo usando los vértices como procesadores y las aristas como las comunicaciones de entrada y salida.

Otro posible caso es que un procesador necesita un dato de su predecesor y ha de enviar un dato a su sucesor, en este caso el algoritmo a utilizar será el siguiente:

```
Sucesor = mythid+1;
Predecesor = mythid +1;
If ( /* I am not the first processor */ )
    Send (target=successor);
If ( /* I am not the first processor */ )
    Receive (source=predecesor)
```

# Bibliografía

[1] Pacheco P.:*An Introduction to Parallel Programming.* Morgan Kaufmann,2011. Capítulo 3, apartado 4.

[2] 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.

[3] Victor Eijkhout, *Parallel Computing for Science and Engineering*, 1st Edition 2015.

[4] Blaise Barney, Lawrence Livermore National Laboratory, *Introduction to Parallel Computing.*

## 2.4 Ejercicios

### 2.4.1 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 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

---

**a.** Inicialmente calculo Nloc y remainder.

```
//Compute Nloc and remainder
Nloc = floor((double)N/(double)npr);
remainder = N - Nloc*npr;
```

Este caso es sencillo y se resuelve con el siguiente bucle y las asignaciones finales.

```
//We distribute the work among the
    procesesses
for(i=0; i<npr-1; i++){
    size[i] = Nloc;
    shift[i] = i*Nloc;
}
//Finally we charge the last process with
    the remainder
size[npr-1] = Nloc + remainder;
shift[npr-1] = (npr-1)*Nloc;
```

**b.** En este segundo caso he comenzado distribuyendo la carga del resto entre los primeros procesadores, luego, las cargas distintas entre procesos no permiten el cálculo de shift usado anteriormente, por lo que tomo las referencias de tamaño y shift calculadas en la anterior iteración para calcular el shift, sabemos donde empezamos porque sabemos dónde termina el anterior.

```
//Value asignment to first process
size2[0] = Nloc;
shift2[0] = 0;

//Distribution of the remainder among the
    first processes
i = 0;
while(remainder){
    size2[i] += 1;
    remainder -= 1;
    i++;
}

//Distribute the rest of the vector among
    the processes
for(i=1; i<npr; i++){
    size2[i] += Nloc;
    shift2[i] = shift2[(i-1)] + size2[(i-1)
        ];
}
```

### 2.4.2 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 una función. A mayor valor de n, más preciso el resultado.

Completa el programa MPI *P12-inteser.c* para realizar esa misma función entre P procesos, utilizando funciones de comunicación colectiva. Compara el resultado con el de la versión serie.

---

Para resolver el problema es necesario modificar la función Read_data, encargada de leer por pantalla los límites superior, inferior y número de evaluaciones de la función. Dado que todos los nodos no pueden utilizar la entrada estándar al mismo tiempo, limito la lectura a un único nodo y luego distribuyo los datos leídos entre el resto de procesos utilizando la función **MPI_Bcast**.

```
void Read_data(double* a_ptr, double* b_ptr,
     int* n_ptr, int pid){

  float a, b;
  float buf[3];

  if (pid == 0){
    printf("\n  Introduce a, b (limits) and
        n (num. of trap.)  \n");
    scanf("%f %f %d", &a, &b, n_ptr);
    buf[0] = a;
    buf[1] = b;
    buf[2] = (float)*n_ptr;
  }
  //Distribute read values
  MPI_Bcast(&buf,3,MPI_INT,0,MPI_COMM_WORLD)

  (*a_ptr)= (double)(buf[0]);
  (*b_ptr)= (double)(buf[1]);
  (*n_ptr)= (double)(buf[2]);
```

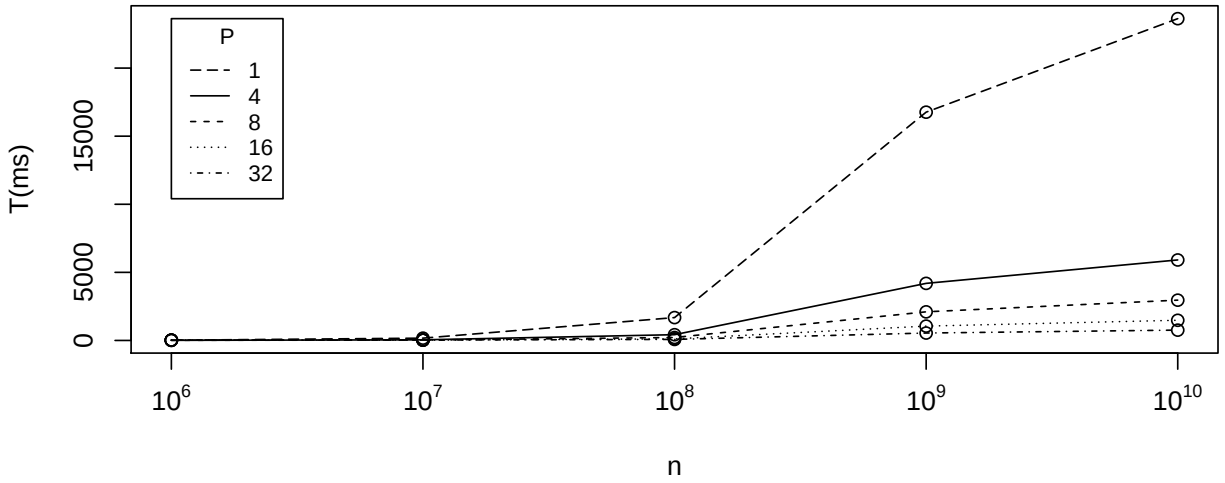
Tras realizar el cálculo del área correspondiente en cada proceso, es necesario sumarlas todas para obtener la integral en todo el intervalo. Esto puede realizarse en una única línea llamando a la función **MPI_Reduce** con los siguientes parámetros.

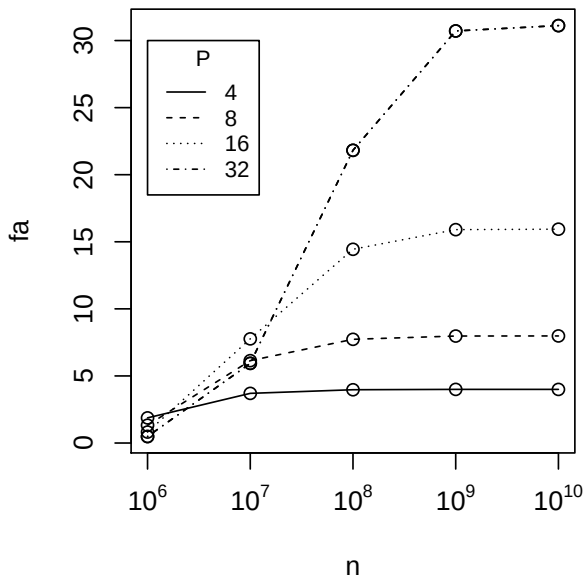
```
/*
Adding the partial results,

  Description of parameters:
  sendbuf  - local result of integral
```

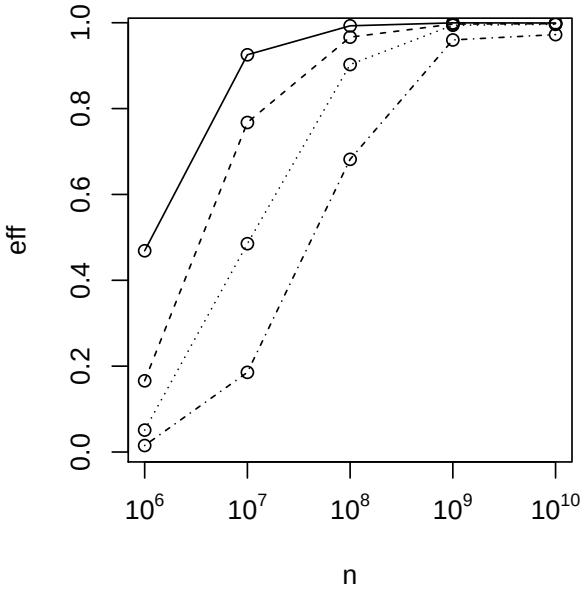
```
  recvbuf   - total result of integral
  count     - 1 element in send buffer per
      porcess
  datatype  - We are using double precission
  op        - We will compute a sume
  root      - the process 0
  comm      - All the processes active
*/
```

```
MPI_Reduce(&resul_loc, &resul, 1,
    MPI_DOUBLE   , MPI_SUM, 0,
    MPI_COMM_WORLD);
```

Las siguientes imágenes muestran los resultados comparando la ejecución para distintos números de procesadores. Los tiempos, los factores de aceleración y la eficiencia de cada uno.



### 2.4.3 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[12], B[13], B[14], B[15]. Tras ello, cada proceso suma 100 a los elementos recibido, y, finalmente, se recopilan los datos en P2, en las mismas posiciones iniciales del vector B.

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

---

Como los segmentos a entregar son de tamaños distintos he utilizado la variante de vector de las funciones Scatter y Gather. Primero creo los vectores de displacement, y sendcounts, desde que posiciones quiero que reciban datos y la cantidad de los mismos.

```
int  displacements[4];
int  sendcounts[4];
displacements[0] = 3;  sendcounts[0] = 3;
displacements[1] = 7;  sendcounts[1] = 2;
displacements[2] = 10;  sendcounts[2] = 1;
displacements[3] = 12;  sendcounts[3] = 4;

// Scattering of B from pid=2
MPI_Scatterv(&B, &sendcounts[0], &
    displacements[0], MPI_INT, &Bloc,
    sendcounts[pid], MPI_INT, 2,
    MPI_COMM_WORLD);

// Local calculation
for(i=0; i < sizeof(&Bloc); i++){ Bloc[i
    ]+=100; }

// Gathering of Bloc in pid=2
MPI_Gatherv(&Bloc, sendcounts[pid], MPI_INT,
     &B, &sendcounts[0], &displacements[0],
    MPI_INT, 2, MPI_COMM_WORLD);
```

### 2.4.4 P2.1

En un programa MPI, el proceso P3 tiene una matriz MAT de 10x10 enteros, de la que tiene que enviar la diagonal al resto de procesos. Completa el programa *P21-diagonal0.c* para que ejecute esa 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.

---

Lo primero que se ha hecho es definir el tipo *diagonal*.

```
// Defining the diagonal type
MPI_Datatype diagonal;
MPI_Type_vector(N, 1, N, MPI_INT, &diagonal)
    ;
MPI_Type_commit(&diagonal);
```

A continuación, se desea enviar la diagonal desde el nodo P3 a los nodos P0, P1 y P2. El nodo P3 tendrá que rellenar su tipo diagonal. En este caso, suponiendo que se trabaje con cuatro nodos, todos los nodos ejecutan **broadcast**. El nodo P3 es el emisor y el resto los receptores Se podría haber hecho con un tres *sends* a los respectivos nodos.

```
// 1. Sending the diagonal of the matrix MAT
     in P3 to P0, P1 and P2
// It is received as a vector in a buffer
if (pid==3){
    printf("Soy nodo %d.Voy a enviar mi
        diagonal:\n",pid);
    for (i=0; i<N; i++){
        buf[i]=MAT[i][i];
        printf ("%d",buf[i]);
        printf(",");
    }
    printf("\n");
}
//Sended data by broadcast
MPI_Bcast(&buf, N, MPI_INT, 3,
    MPI_COMM_WORLD);
if (pid !=3){
    for (i=0; i<N; i++){
        printf ("Nodo: %d Buf: [%d][%d] %d\n
            ",pid,i,i,buf[i]);
    }
}
```

### 2.4.5 P2.2

Completa el programa *P22-pack0.c* para que P1 envíe a P2 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, P1 empaqueta los datos y envía el paquete a P2; por su parte, P2 recibe el mensaje y desempaqueta los datos. Ejecuta el programa con 4 procesos.

---

Lo primero que se hace es inicializar los datos en P1. A continuación, se empaquetan los datos de interés y se envían al nodo de destino mediante MPI_Send.

```
if (pid==1){
    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
    MPI_Pack(&A[0][0], sizeA*sizeA, MPI_INT,
         buf, sizebuf, &pos, MPI_COMM_WORLD
        );
    MPI_Pack(&B[0], sizeB, MPI_FLOAT, buf,
        sizebuf, &pos, MPI_COMM_WORLD);
    MPI_Pack(&C, 1, MPI_DOUBLE, buf, sizebuf
        , &pos, MPI_COMM_WORLD);
    MPI_Send(buf, pos, MPI_PACKED, 2, 0,
        MPI_COMM_WORLD);
}
```

Ahora hay que recibir los datos en el nodo P2. Hay que tener en cuenta que hay que desempaquetar en el mismo orden que en el que se envía.

```
if (pid==2){
    MPI_Recv(buf, sizebuf, MPI_PACKED, 1, 0,
         MPI_COMM_WORLD, &info);
//Unpacking
    pos=0;
    MPI_Unpack(buf, sizebuf, &pos, &A[0][0],
         sizeA*sizeA , MPI_INT,
        MPI_COMM_WORLD);
    MPI_Unpack(buf, sizebuf, &pos, &B[0],
        sizeB, MPI_FLOAT, MPI_COMM_WORLD);
    MPI_Unpack(buf, sizebuf, &pos, &C, 1,
        MPI_DOUBLE, MPI_COMM_WORLD);
}
```

### 2.4.6 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:

*I am pid:0 and pid2: 0 and W(0,5,10,15) are 0 1 2 3*

*I am pid:0 and pid2: 0 and W(0,5,10,15) are 0 1 2 3*

### 2.4.7 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 de ese programa. En la primera, se reparten las tareas entre todos los proceso de modo estático consecutivo, procesando cada uno de ellos 320/npr números consecutivos.

En la segunda, el reparto de tareas debe ser dinámico, bajo demanda. Uno de los procesos (P0, por ejemplo) funciona como manager y reparte a cada uno de los restantes procesos (workers) números a procesar, uno a uno, cuando 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.

---

### Versión estática

Una vez declaradas las funciones y variables necesarias para llevar a cabo nuestro objetivo podemos empezar con la distribución de elementos a analizar:

```
//Distribution of tasks
reparto = floor(NUMBER/npr);
resto = NUMBER - reparto*npr;
int i = 0;
first[0]= 1;
for(i=1; i<npr+1; i++){
    if(resto > 0){
        first[i] = first[i-1] + reparto + 1;
        resto--;
    }else{
        first[i] = first[i-1] + reparto;
    }
}
```

De esta manera conseguimos que los elementos que forman el problema a solventar se reparten de manera equitativa y en orden con el número de procesadores involucrados en el desarrollo.

```
Ej: npr = 4
pid = 0 => 1..80 pid = 1 => 81..160
pid = 2 => 161..240 pid = 3 => 241..320
```

Una vez distribuidos los elementos cada procesador ejecuta el algoritmo de Collatz en el rango de valores asignado:

```
//Execution of work
for (n=first[pid]; n<first[pid+1]; n++){
    steps=collatz(n);
    work(steps);
    total_steps+=steps;
    if (steps > max_steps) {n_max_steps = n;
        max_steps = steps;}
}
```

Una vez calculados los distintos valores de la ejecución en paralelo recogemos los resultados obtenidos:

```
//Gather of results
int all_max_steps, all_steps;

MPI_Reduce( &total_steps,&all_steps, 1,
    MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce( &max_steps,&all_max_steps, 1,
    MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
```

### Versión dinámica

En este caso el reparto de tareas se hace en tiempo de ejecución, El algoritmo utilizado es el siguiente:

**Manager:**
Inicialmente enviamos una tarea a cada Worker, por cada envío esperamos una respuesta por lo que llevamos la cuenta en la variable answer.

```
int n = 1,answer = 0;

//Send initial work to each worker
for(i=1; i < npr; i++ ){
    if(n < NUMBER+1){
        MPI_Send(&n, 1, MPI_INT, i, 0,
            MPI_COMM_WORLD);
        n++;
        answer++;
    //In case npr is greater than NUMBER,
        the remaining processes wont work
    }else{
        MPI_Send(&n, 1, MPI_INT, info.
            MPI_SOURCE, 1, MPI_COMM_WORLD);
    }
}
```

Mientras esperemos respuestas, esto es, que answer sea mayor que cero, recibimos el resultado de alguno de los Workers, answer decrece, en caso de quedar trabajo, se envía otro dato junto con el Tag 0, como esperamos una respuesta, answer aumenta, sino, se envía un mensaje junto con el Tag 1 y no esperamos respuesta.

```
//MANAGER LOOP
while(answer > 0){
    MPI_Recv(&f, 1, MPI_INT, MPI_ANY_SOURCE,
         MPI_ANY_TAG, MPI_COMM_WORLD, &info)
        ;
    answer--;
    total_steps+=f;
    if (f > max_steps) {n_max_steps = n;
        max_steps = f;}

    if(n < NUMBER + 1){
        MPI_Send(&n, 1, MPI_INT, info.
            MPI_SOURCE, 0, MPI_COMM_WORLD);
        answer++;
        n++;
    }else{
        MPI_Send(&n, 1, MPI_INT, info.
            MPI_SOURCE, 1, MPI_COMM_WORLD);
    }
}
```

**Worker:**
El bucle de Worker es mucho más simple y consiste en una recepción inicial, si se trata de trabajo (TAG 0), se ejecuta la tarea y se envían los resultados. Esperamos la respuesta del Manager con más trabajo o indicando el final de este.

```
//WORKER LOOP
MPI_Recv(&b, 1, MPI_INT, 0, MPI_ANY_TAG,
    MPI_COMM_WORLD, &info);//Recibe trabajo
while(info.MPI_TAG != 1){
```

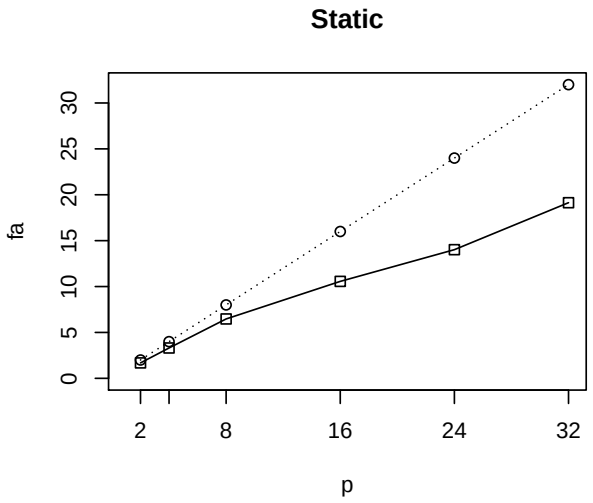
```
    steps=collatz(b);
    work(steps);
    MPI_Send(&steps, 1, MPI_INT, 0, 1,
        MPI_COMM_WORLD);//Env a resultados
    MPI_Recv(&b, 1, MPI_INT, 0, MPI_ANY_TAG,
         MPI_COMM_WORLD, &info);//Recibe
        trabajo
}
```

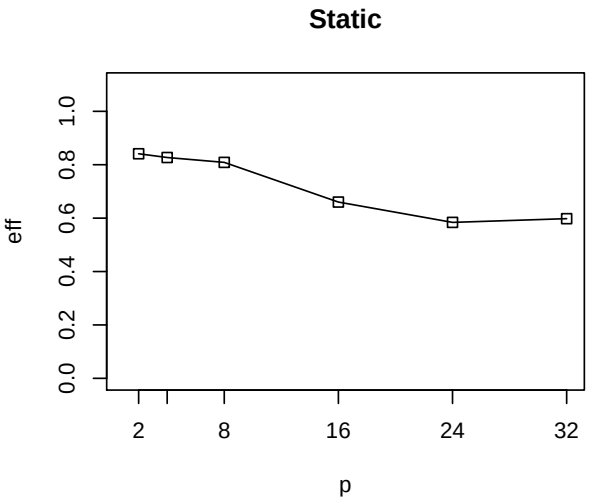
En los siguientes gráficos pueden apreciarse las diferencias en los factores de aceleración y eficiencias en cada caso.

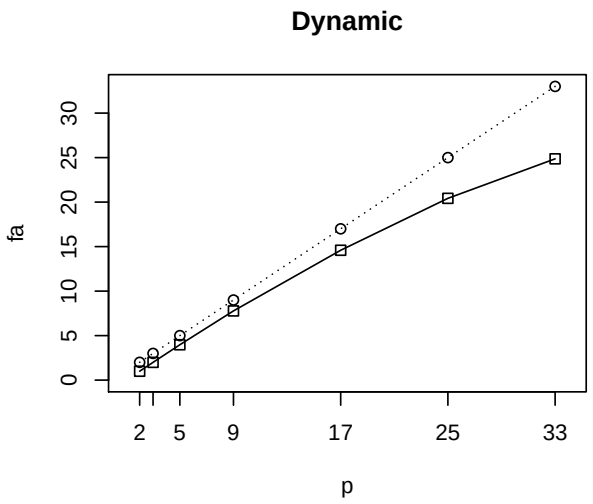
Como era de esperar el reparto dinámico consigue mejores factores de aceleración para todos los números de procesadores exceptuando el primer caso, dónde el modelo manager worker carecería de sentido. Del mismo modo se aprecia como la eficiencia aumenta con el número de procesadores, los beneficios del reparto dinámico van superando los costes de la comunicación entre el Manager y los Worker, eficiencia que alcanza su máximo con 9 procesadores y va bajando suavemente hasta los 33, probablemente esto se deba a que los Workers terminan su trabajo y hacen cola esperando la respuesta del Manager.

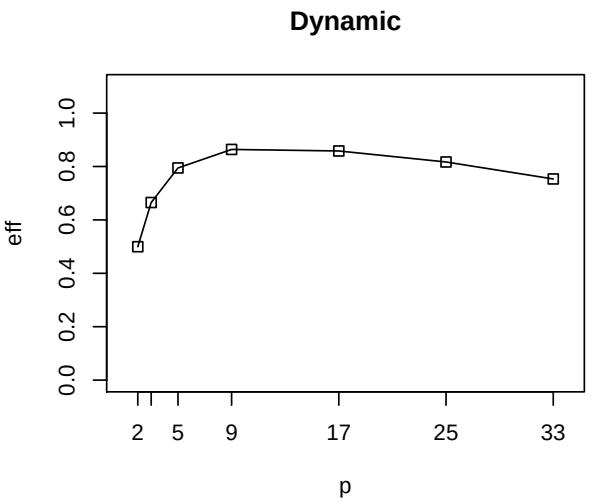
Por otro lado, las cargas de trabajo no están homogéneamente distribuidas y el reparto estático lo ignora, esto explica los resultados para este caso y la mala eficiencia.

| **Proc.** | **Tstatic**(ms) | **Proc.** | **Tdynamic**(ms) |
|---|---|---|---|
| 1 | 30082 | 1+1 | 30107 |
| 2 | 17883 | 1+2 | 15076 |
| 4 | 9093 | 1+4 | 7572 |
| 8 | 4649 | 1+8 | 3869 |
| 16 | 2848 | 1+16 | 2062 |
| 24 | 2146 | 1+24 | 1473 |
| 32 | 1572 | 1+32 | 1210 |


Static
fa
0
5
10
15
20
25
30
2
8
16
24
32
p


Static
eff
0.0
0.2
0.4
0.6
0.8
1.0
2
8
16
24
32
p


Dynamic
fa
0
5
10
15
20
25
30
2
5
9
17
25
33
p


Dynamic
eff
0.0
0.2
0.4
0.6
0.8
1.0
2
5
9
17
25
33
p

### 2.4.8 P3.2

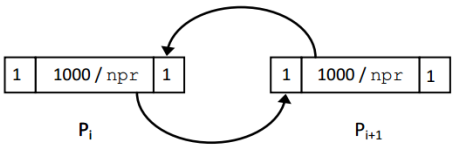
En una determinada aplicación (*P32-convoser.c*) se procesa un vector de $N = 1002$ elementos al que se aplica una operación de convolución de manera iterativa

```
for (i=1; i<N-1; i++) Aux[i] = (A[i-2] + 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. Como puedes ver, el primer y el último elemento del vector no se procesan.

El programa se va a ejecutar con un número de procesos divisor de 1000. Escribe una versión paralela del programa en el que **(a)** *P0* reparte los 1000 elementos del vector que hay que procesar, en trozos de $1000/npr$ elementos: el primero lo procesará el mismo, el segundo *P1*...; **(b)** cada proceso efectúa la operación sobre el trozo de vector que le ha tocado y calcula el máximo local;**(c)** los procesos envían su máximo local a *P0*, el que calcula el máximo global y, tras ello, avisa a todos los procesos si hay que efectuar una nueva convolución o si la operación ha terminado;**(d)** al acabar, los procesos envían a *P0* su trozo de vector S procesado, para que éste los reconstruya.

Ten en cuenta que cada procesador va a necesitar para procesar su trozo los datos que le han correspondido y 2 más, el anterior al primero 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(que van a ir cambiando iteración a iteración).




---

Tal y como indica el enunciado, hay que repartir los trozos de vector entre los procesadores. Primero calculamos en todos los procesos los tamaños y desplazamientos. Comprobamos que efectivamente el número de procesadores es divisor de 1000 y creamos los array locales donde trabajará cada procesador.

```
reparto = floor((NELEM-2)/npr);
resto = (NELEM-2) - reparto*npr;
if(resto != 0){
    printf ("\n\n ERROR: 1000 must be
        multiple of npr\n\n");
    exit (-1);
}
int size[npr],displacement[npr];
    for(i=0; i<npr; i++){
    size[i] = reparto;
    displacement[i] = i*reparto;
}

float A[NELEM], Aux[NELEM];
float A_loc[reparto+2], Aux_loc[reparto+2];
```

Tras llenar aleatoriamente en el proceso 0 el vector A, procedemos a su reparto utilizando la función Scatterv con los vectores size y displacement previamente calculados. El primer y último elemento de A, no se reparten ni modifican ni tampoco se comunican como frontera, pero si son necesarios para el cálculo por lo que los asigno a mano.

```
//Reparto de los datos iniciales
 MPI_Scatterv(&A[1], &size[0], &displacement
     [0],  MPI_FLOAT,
                    &A_loc[1], size[pid],
                        MPI_FLOAT, 0,
                        MPI_COMM_WORLD);
if(pid == 0){
    A_loc[0] = 150;
}
if(pid == npr-1){
    A_loc[reparto+1] = 150;
}
```

Después del reparto inicial, comienza el bucle de cálculo hasta la convergencia, pero cada iteración, antes de realizar la convolución hay que actualizar los datos que son frontera entre procesadores, para ello realizamos dos envíos en cascada uno desde 0 hasta el último proceso, enviando cada proceso su "último" elemento al sucesor. Y otro envío desde el último proceso hasta 0 enviando cada proceso su "primer" elemento a su predecesor.

```
if(pid > 0){
    //Recibimos en la primera posicion el
        ante ultimo elemento del predecesor
    MPI_Recv(&A_loc[0], 1, MPI_FLOAT, pid-1,
         0, MPI_COMM_WORLD, &info);
    //Enviamos ante ultimo elemento al
        sucesor en caso de que exista
    if(pid < npr-1){
        MPI_Send(&A_loc[reparto], 1,
            MPI_FLOAT, pid+1 , 0,
            MPI_COMM_WORLD);
    }
}else{
    //Enviamos ante ultimo elemento al
        sucesor
    MPI_Send(&A_loc[reparto], 1, MPI_FLOAT,
        pid+1 , 0, MPI_COMM_WORLD);
```

```
}

if(pid < npr-1){
    //Recibimos en la ultima posicion el
        segundo elemento de los sucesores
    MPI_Recv(&A_loc[reparto+1], 1, MPI_FLOAT
        , pid+1, 0, MPI_COMM_WORLD, &info );
    //Enviamos segundo elemento al
        predecesor en caso de que exista
    if(pid > 0){
        MPI_Send(&A_loc[1], 1, MPI_FLOAT,
            pid-1 , 0, MPI_COMM_WORLD);
    }
}else{
    //Enviamos segundo elemento al
        predecesor
    MPI_Send(&A_loc[1], 1, MPI_FLOAT, pid-1
        , 0, MPI_COMM_WORLD);
}
```

Para determinar el final del cálculo, se recoge el máximo global en 0 y se envía el resultado del control de convergencia.

```
//Recogemos el maximo global
MPI_Reduce(&max_loc, &max, 1, MPI_FLOAT,
    MPI_MAX, 0, MPI_COMM_WORLD);
// convergence control
if(pid == 0){
    steps++;
    if (max < lim) end=1;
}
//Send convergence result
MPI_Bcast(&end, 1, MPI_INT, 0,
    MPI_COMM_WORLD);
```

Finalmente, una vez fuera del bucle, recogemos en 0 el vector entero utilizando la función gatherv.

```
//Recogida del vector final
MPI_Gatherv(&A_loc[1], size[pid], MPI_FLOAT,
     &A[1], &size[0],&displacement[0],
    MPI_FLOAT, 0, MPI_COMM_WORLD);
```

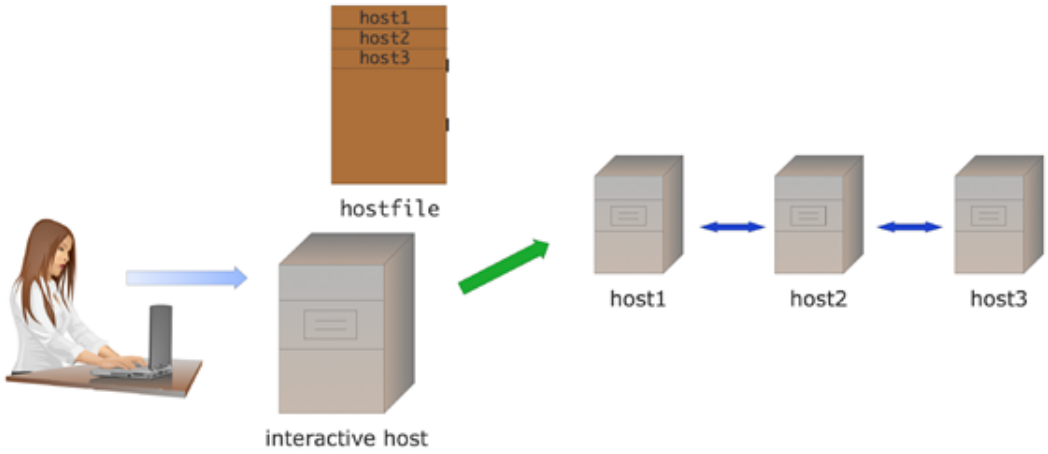
# 3 Anexos

## 3.1 El Hardware

En su forma más simple, una máquina de memoria distribuida es un conjunto de ordenadores conectados a través de cables de red, a esto se le denomina "Beowulf Cluster". Cada procesador puede ejecutar de forma independiente una tarea, y tiene su propia memoria, sin acceso directo a la del resto de nodos. MPI es el que permite ejecutar múltiples instanciaciones de un mismo ejecutable e intercambiar información a través de la red.

## 3.2 Modelo Básico: Configuración MPI Interactiva

Este es el modelo con el que se trabaja en el laboratorio:

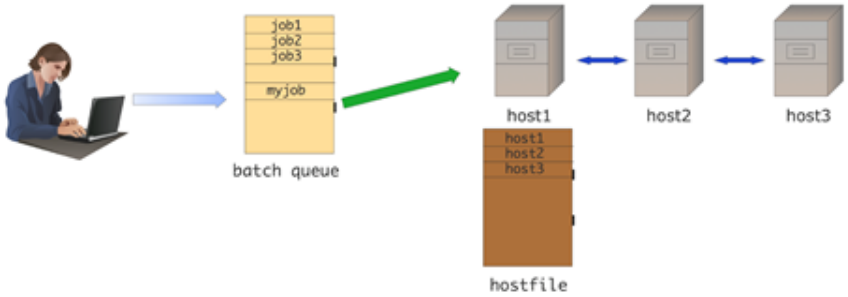



El usuario trabaja con una máquina interactiva, la cual tiene acceso a un número de hosts, normalmente formados por una red de "Workstations" o estaciones de trabajo.

El usuario tecleará el comando "mpiexec" y deberá proporcionar el número de nodos con los que quiere trabajar, sus nombres (habitualmente en un archivo llamado "hostfile"), y algunos otros parámetros seguidos del nombre de la aplicación y sus argumentos.

El programa "mpirun" se conectará mediante ssh a cada host, dándole la información necesaria para que se puedan encontrar entre ellos. Toda salida de los nodos irá encaminada a través del mpirun, y aparecerá en la consola interactiva.

Existe otro segundo escenario llamado Configuración MPI Batch. El usuario crea un script con la tarea a ejecutar y es controlado por un "scheduler" que envía la tarea a los hosts. El script contiene el comando "mpirun", y el "hostfile" se genera dinámicamente cuando empieza el trabajo. Como la tarea puede comenzar cuando el usuario no esté activo, la salida se guarda en un fichero.



### 3.3 Características de las Máquinas

A la hora de ejecutar un programa conviene conocer el entorno de pruebas con el que se está trabajando. Para este proyecto se está utilizando un cluster de tipo "Beowulf" que consiste en 33 máquinas de las mismas características. Más abajo se expondrán sus detalles técnicos. El entorno de trabajo es bajo SSH. Mediante el comando "lshw" se obtiene información sobre el hardware.

- Procesador: Intel Core 2 Duo E6320 1.86GHz (Lanzado en Q2 del año 2007).
  - Núcleos: 2
  - Frecuencia: 1860MHz
  - Caché L2: 4MB
- Chipset: Intel P965 Express
- Memoria RAM: 1977MiB DDR2 ($\approx$ 2GB)
- Adaptador de Red 1: 82566DC Gigabit Network Connection (Integrado en placa base)
  - Size: 100Mbit/s
  - Capacity: 1Gbit/s
- Adaptador de Red 2: DGE-528T Gigabit Ethernet Adapter
  - Size: 1Gbit/s
  - Capacity: 1Gbit/s
  - 2000Mbps Gigabit full duplex support

Pese a que existan dos adaptadores de red, lo más probable es que todos los equipos vayan conectados a un mismo switch a través del segundo adaptador, es decir, con una conexión de Gigabit Ethernet.

A partir del comando "lsblk" se obtiene información sobre los dispositivos de almacenamiento del sistema:

```
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
    sda 8:0 0 149,1G 0 Disk
    sda1 8:1 0 146,1G 0 Part /
    sda2 8:2 0 1K 0 Part
    sda5 8:5 0 3G 0 Part
```

Para conocer la información sobre el Sistema Operativo que está en ejecución se utiliza el comando "lsb_release –a"
No LSB modules are available.
Distributor ID: Ubuntu
Description: Ubuntu 12.04.1 LTS
Release: 12.04
Codename: precise

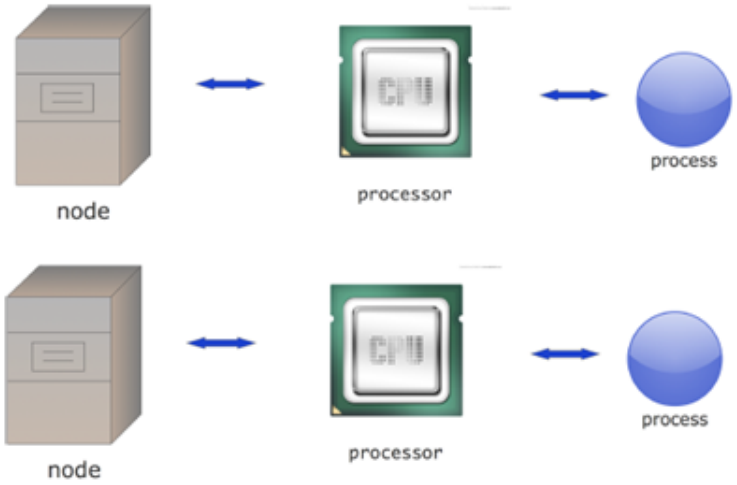
Para conocer con qué compiladores se va a trabajar, se introduce en la consola de Linux el siguiente comando, "mpicc –v". Éste muestra la versión del mpicc y la del icc, de modo que no hay necesidad de introducir "icc –v". El resultado es el siguiente.
mpicc for MPICH2 version 1.5
icc version 13.0.1 (gcc version 4.6.0 compatibility)
A la hora de compilar se usa el nivel 2 de optimización.

### 3.3.1 Posibles mejoras de hardware

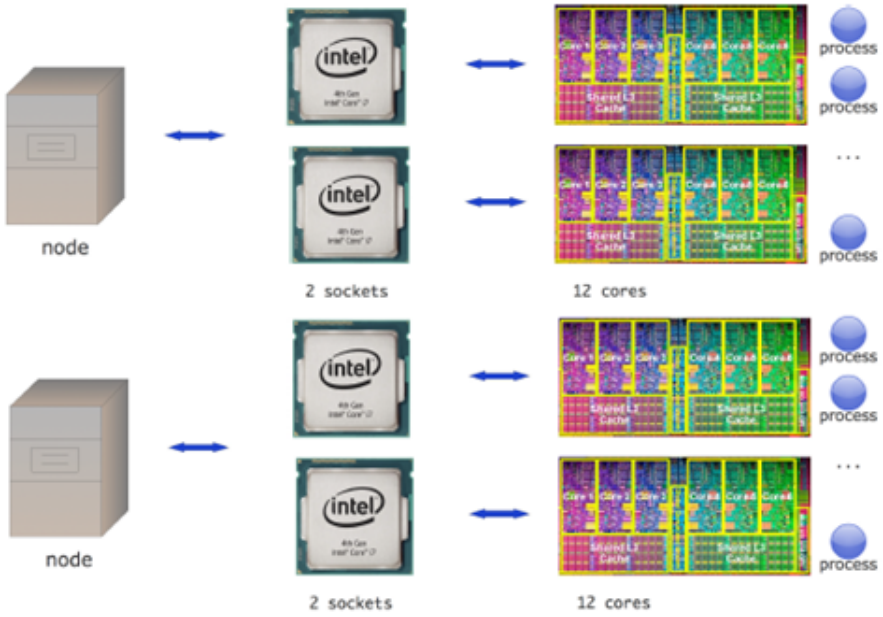
Con estos datos que se han obtenido del hardware actual de los equipos, se pueden considerar ciertas mejoras que disminuirían los tiempos de ejecución de los programas. Algunas son las siguientes.

#### 3.3.1.1 Combinar MPI con Multi-threading y OpenMP

Esta mejora se encuentra entre el hardware y la programación. Cuando empezó MPI hace veinte años, los procesadores sólo tenían un único núcleo. En caso de que se estuviera trabajando con un cluster, a los procesadores se les denominaba nodos. Cuando se ejecutaba un programa en MPI cada procesador tendría un único proceso.




Actualmente puede haber más de un núcleo en cada procesador, de hecho es ya algo más que habitual. En las placas base destinadas a servidores es también común que existan más de un socket, por lo que puede haber dos procesadores físicos con sus correspondientes núcleos. Por ejemplo, en la siguiente imagen aparece el ejemplo de dos nodos, cada uno de ellos con dos procesadores que a su vez contienen 6 núcleos. Por lo tanto, cada nodo tiene 12 núcleos para procesar información. Ahora un mismo nodo podría tener más de un proceso en ejecución al mismo tiempo, ya que cada núcleo se comporta como un procesador independiente, con la ventaja de tener acceso a memoria compartida.




El modelo básico de MPI, en un principio, ignora la memoria compartida y trata a cada núcleo como un nodo más de la red de comunicación. No se puede ver de forma inmediata qué núcleos están en la misma máquina. Esto es ineficiente, ya que los mensajes a través de la red son más lentos y además consumen ancho de banda. Sin embargo se puede configurar para que esto no ocurra y se aprovechen de la cercanía y la memoria compartida.

Un usuario puede hacer una petición de multi-threading con "MPI_Init_thread", y el sistema responderá con el mayor nivel soportado.

En cuanto a la programación híbrida (MPI + OpenMP) pretende aprovechar las ventajas de los dos mundos. Como OpenMP trabaja con un espacio de memoria compartida, es posible reducir el coste de memoria asociado a las tareas de MPI y la necesidad de duplicar datos. Otras ventajas son el balanceo de carga y la posibilidad de trabajar con dos niveles de paralelismo al mismo tiempo. Sin embargo, el principal problema de mezclar MPI con OpenMP es que hay muchas dificultades

#### 3.3.1.2 Mejora de las comunicaciones

Se ha comprobado que las máquinas trabajan con dos interfaces de red, una integrada en placa y otra a través de PCI. Hay que recordar que PCI tiene un ancho de banda de 133,3MB/s, y que 1Gbit son 125MB. Si se desea trabajar con Gigabit Ethernet puede ser una interfaz de comunicación bastante acertada.

Si se van a usar las dos interfaces para la comunicación, la velocidad máxima de la interfaz más lenta va a ser un condicionante.

En tal caso, una mejora sería utilizar otra interfaz de red. Se puede utilizar otra con PCI o PCI-e.

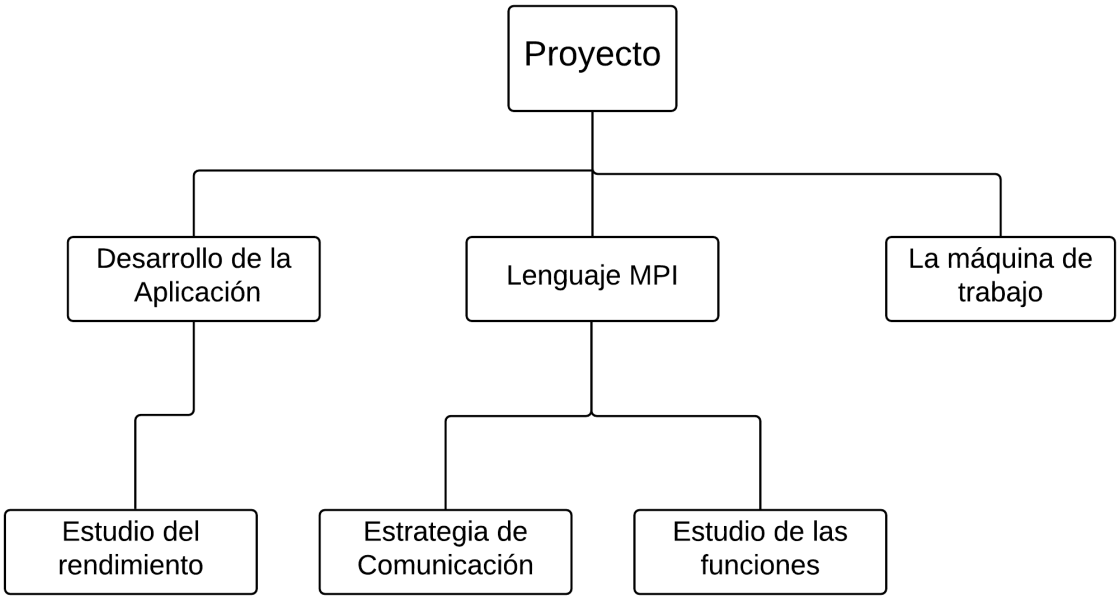
#### 3.3.1.3 Procesadores

El procesador es un elemento clave a la hora de procesar datos. En este caso se ha visto que se está usando un Socket 775 con un Intel Core2 Duo E6320 1.86GHz.

Para el socket 775 existe una técnica que se denomina “775 mod”. Consiste en, mediante una pequeña modificación, hacer compatible éste socket con los procesadores del socket 771, que son los Intel Xeon, destinados a servidores.

Aunque en el momento de su aparición llegaron a costar los 1000€, actualmente se pueden encontrar por 20€. Un ejemplo podría ser un Xeon X5450, aunque hay mejores opciones. Los únicos inconvenientes que tiene el realizar esta técnica es el tener que manipular parte del hardware y que el procesador E6320 que se está utilizando actualmente tiene un costo energético bastante bajo. Aunque el procesador del X5450 sea bastante mejor en muchos aspectos (Frecuencia de 3Ghz, 4 nucleos, mejor rendimiento, instrucciones SSE4.1...), éste consume significativamente más, por lo que para el uso que se le da al cluster, es muy probable que no merezca la pena esa modificación. No obstante, si fuera un cluster muy activo y en el que el tiempo es algo realmente significativo, sí que se podría plantear esta opción.

### 3.4 Poster




### 3.5 Tabla de dedicación

| Tarea | Mikel | Jose Ángel | Christian |
|---|---|---|---|
| Estudio Programación MPI | 1h 30’ | 1h | 1h |
| Puzle 1 | 13h 45’ | 15h 45’ | 8h 30’ |
| -Estudio | 7h | 8h | 3h |
| -Documentación | 3h | 2h | 2h |
| -Problemas | 3h | 5h | 3h |
| -Presentación | 45’ | 45’ | 30’ |
| Parte 1 Aplicación | 12h | 6h | - |
| -Adaptación del código | 8h | 4h | - |
| -Documentación | 4h | 2h | - |
| Parte 2 Aplicación | 16h | 11h | - |
| -Adaptación del código | 8h | 5h | - |
| -Documentación | 8h | 5h | - |
| Presentación | 2h 30’ | 2h 30’ | - |
| Total | 46h | 35h 45’ | 9h 30’ |

## 3.6 Actas

**Informatika Fakultatea, UPV/EHU**

**S**istemas de **C**ómputo **P**aralelo 15/16

Aprendizaje Basado en Proyectos: Programación paralela

**ACTA DE CONSTITUCIÓN DE GRUPO - DOCUMENTO DE COMPROMISOS**

▪ **Participantes**

| | Nombre y apellidos | Firma |
|---|---|---|
| 1. | Mikel Dalman Cherino | |
| 2. | Jose Ángel Gamiel Quintana | J.A. Gamiel |
| 3. | Christian Merino Arenas | Christian |

▪ **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

**Informatika Fakultatea, UPV/EHU**

**S**istemas de **C**ómputo **P**aralelo 15/16

Aprendizaje Basado en Proyectos: Programación paralela

## ACTA DE SESIÓN DE TRABAJO DE GRUPO

- **Asistentes**

| | Nombre y apellidos | Firma de conformidad |
|---|---|---|
| 1. | Jose Ángel Gumiel Quintana | J.A. Gumiel |
| 2. | Mikel Dalman Cherino | |
| 3. | Christian Merino Arenas | Christian |

- **Temas tratados y decisiones tomadas**

1.- Exposición del trabajo realizado.
2.- Decisión de usar Google Drive como sistema de información.
3.- Decisión del uso de Latex para los informes y la documentación.

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

1.- Terminar los informes correspondientes a cada una de las partes.
2.- Continuar con los ejercicios.

| 14/03/16 | 17:00 - 17:30 |
|---|---|
| Fecha | Hora comienzo — Hora final |

**Informatika Fakultatea, UPV/EHU**

**S**istemas de **C**ómputo **P**aralelo 15/16

Aprendizaje Basado en Proyectos: Programación paralela

## ACTA DE SESIÓN DE TRABAJO DE GRUPO

- **Asistentes**

| | Nombre y apellidos | Firma de conformidad |
|---|---|---|
| 1. | Jose Ángel Gumiel Quintana | J.A. Gumiel |
| 2. | Mikel Dalman Cherino | |
| 3. | Christian Merino Arenas | Christian |

- **Temas tratados y decisiones tomadas**

1.- Puesta en común de las tres partes del puzzle.
2.- Realización de la presentación.

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

1.- Estudiar presentación. Especialmente las partes de los compañeros.

| 04-04-16 | 18:00 - 19:30 |
|---|---|
| Fecha | Hora comienzo — Hora final |

## 3.7 Ejercicios de representación de datos

**Ejercicio 1:** Se ejecuta 8 veces un determinado programa y se mide su tiempo de ejecución. Los resultados obtenidos aparecen en la siguiente tabla. Hacer una estimación del tiempo de ejecución del programa.

| Tiempo (ms) |
|---|
| 23,256 |
| 24,128 |
| 23,872 |
| 24,190 |
| 25,876 |
| 144,637 |
| 22,987 |
| 23,386 |

La media de los datos de la tabla es: 39.0415. Parece que no es acorde a los datos, esto se debe a que la media no es un estadístico robusto y se ve influenciada con facilidad por los valores extremos, no así la moda, que sí lo es y tiene un valor: 24 refleja mucho mejor el tiempo de ejecución del programa.

De todas formas no podemos desechar un valor a la ligera, para ello voy a utilizar un modelo en el que considero como extremos todos los datos que no se encuentren entre los cuartiles Q1 y Q3 de la distribución. Para no hacer un modelo tan restrictivo multiplico el rango intercuartil por 1.5.

```
Q1 <- quantile(x,probs=0.25)
Q3 <- quantile(x,probs=0.75)
step<- 2.5 * (Q3-Q1)
res <- sapply(x,FUN=function(x){
        if( x < Q3-step | x > Q1+step ){
            x <- NA
        }else{ x }})
res

## [1] 23.256 24.128 23.872 24.190 25.876     NA 22.987 23.386
```

En definitiva, el programa tendría un tiempo de ejecución de 24 ms, la media del resto de valores no excluidos.

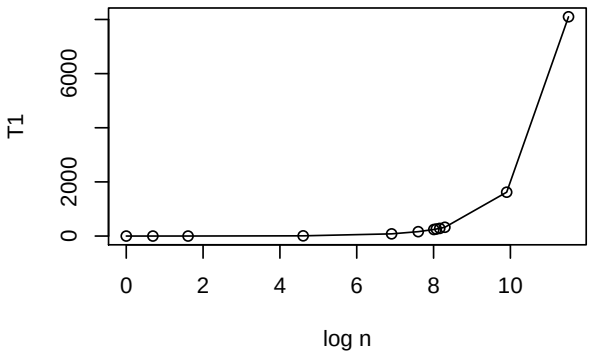
**Ejercicio 2:** En un determinado experimento se ha medido el tiempo de ejecución de dos alternativas de un programa en función del tamaño de los vectores procesados (y, en consecuencia, en función de la carga de cálculo), y se han obtenido los datos de la tabla. Representar los datos en un gráfico de la manera más adecuada posible y sacar conclusiones.

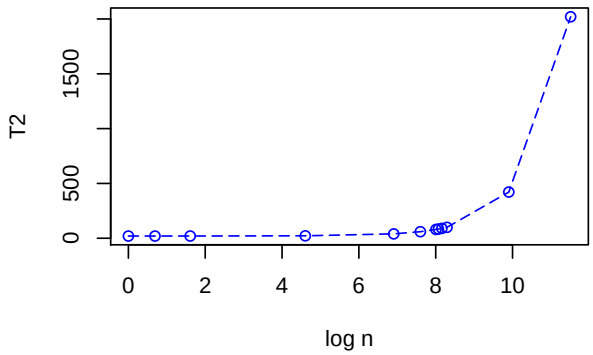
| N(Bytes) | T1(ms) | T2(ms) |
|---|---|---|
| 1 | 0,081 | 20,020 |
| 2 | 0,162 | 20,053 |
| 5 | 0,405 | 20,215 |
| 100 | 8,104 | 22,167 |
| 1000 | 81,062 | 39,893 |
| 2000 | 162,010 | 59,756 |
| 3000 | 242,013 | 80,176 |
| 3200 | 259,197 | 83,830 |
| 3500 | 283,518 | 89,875 |
| 4000 | 323,998 | 99,518 |
| 20000 | 1620,012 | 421,532 |
| 100000 | 8099,996 | 2020,225 |

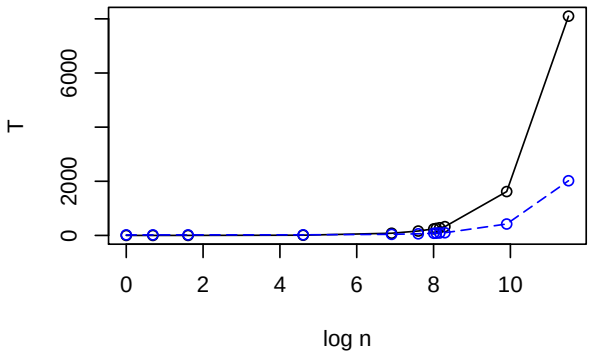
Los datos de la siguiente tabla abarcan un gran número de valores y N incrementa diferentemente. Sabemos que ambos algoritmos son lineales mirando a los datos, el tiempo crece con N = 100 de 8ms para T1 y 22ms para T2 a 8000ms y 20000ms con N=100*1000.También sabemos que hay un punto de corte entre 100 y 1000.

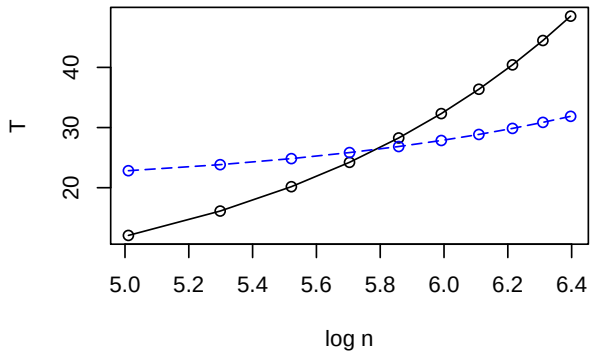
Los siguientes gráficos muestran ambas rectas y su punto de corte, que se ha realizando creando puntos entre 150 y 600 con las funciones ajustadas de T1 y T2.

Podemos concluir que el programa 1 es más rápido con vectores pequeños, de 350 Bytes o menos aproximadamente, a partir de dicho tamaño el programa 2 es mejor ya que crece con una pendiente de 2% y el programa 1 del 8%.



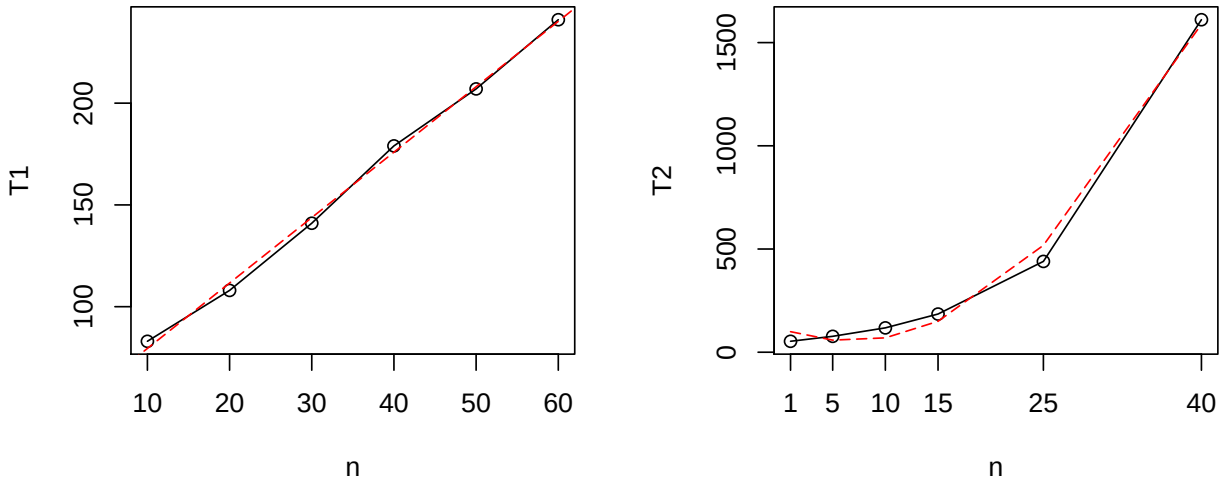



**Ejercicio 3:** Las siguientes tablas muestran los tiempos de ejecución de dos programas diferentes en función del parámetro N. Representa los datos y obtén una expresión matemática que indique cómo varía T en función de N. A partir de ahí calcula el tiempo de ejecución esperado en cada caso para N=100.

| N | T1 |
|---|---|
| 10 | 83 |
| 20 | 108 |
| 30 | 141 |
| 40 | 179 |
| 50 | 207 |
| 60 | 241 |

| N | T2 |
|---|---|
| 1 | 53 |
| 5 | 77 |
| 10 | 118 |
| 15 | 185 |
| 25 | 440 |
| 40 | 1611 |

En las siguientes gráficas pueden verse las representaciones de los datos en negro y sus respectivos ajustes en rojo.




Se ha ajustado las tablas con las siguientes funciones:

Lineal : $y = 47.3333333 + 3.2142857\ x$

Cuadrático : $y = 116.7037178\ -18.5565452\ x + 1.3840086\ x^2$

El tiempo de ejecución esperado a cada caso sería aproximado al los siguientes:

Lineal : 368.7619048 ms.

Cuadrático : $1.2101135 \times 10^4$ ms.

## 3.8 Escenario y Puzle

**Informatika Fakultatea**
**UPV/EHU**

Grado en Ingeniería Informática

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea

**S**istemas de **C**ómputo **P**aralelo

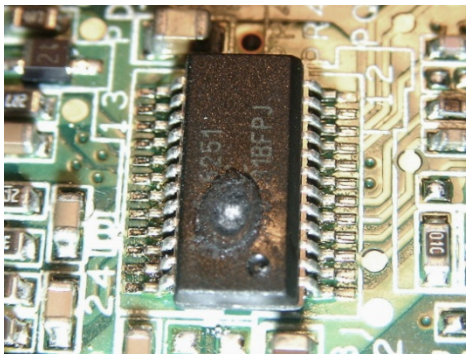
Ingeniería de Computadores

**Aprendizaje Basado en Proyectos** (segunda parte)

**3.** Programación paralela: MPI

## ▪ Pregunta motriz

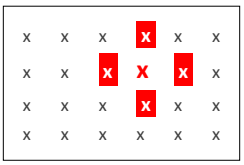
**¡Cuidado, que quema!**



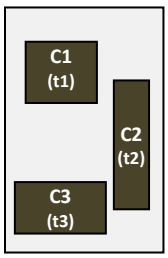
## ▪ Escenario

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.

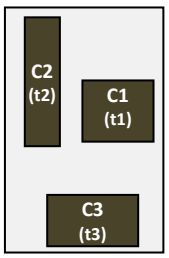
Antes de fabricar el circuito final, se simula un algoritmo de difusión del calor usando diferentes localizaciones de los chips en la tarjeta, para elegir aquella configuración que minimiza la temperatura media global de la tarjeta. Para aplicar el algoritmo de difusión del calor, se divide la tarjeta 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.



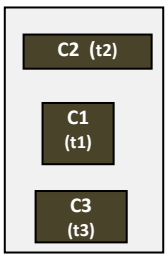
La simulación se hace en un equipo monoprocesador analizando el comportamiento de diferentes configuraciones, hasta obtener la mejor solución. La siguiente figura muestra tres posibles configuraciones de una tarjeta de circuito impreso con tres chips, en las que, tras el proceso de difusión del calor,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

## ▪ Temario a cubrir en el proyecto

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.

## ▪ Resultados de aprendizaje

Al término de esta tarea deberás ser capaz de:

- Programar de forma eficiente pequeñas aplicaciones en multiprocesadores 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”).

## ▪ 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 profesor).

**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 serán cooperativos, grupales, salvo el entregable E5 (examen de conocimientos mínimos), que será un entregable individual.

## ▪ 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 profesor y estudiantes, 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** | **Profesor/a** | | **Estudiantes** | |
|---|---|---|---|---|---|
| **Individual** | 2 puntos | Examen (E5) | 2 p. | | |
| **Grupo** | 4 puntos | Presentaciones (E3.2/E6.3) | 0,5 p. | Presentaciones (E4.1/E4.2) | 0,5 p. |
| | | Informe técnico (E6.2) | 3 p. | | |
| | | Portafolio final (E7) | filtro | | |
| **Nota total** | **6 puntos** | | 5,5 p. | | 0,5 p. |

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.

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 (por confirmar).

**PLANIFICACIÓN DEL TRABAJO**

| Sem. | Clase | Actividades presenciales | t/día | Actividades no presenciales (por semana) | t/sem | Entregables | Evaluación |
|---|---|---|---|---|---|---|---|
| 29 -4/03 | 1 | Entrega del escenario del proyecto y reflexión por grupos | 30 m | | | Acta: constitución de grupo (E1.1) | |
| | | Discusión PBL, póster final | 30 m | | | Póster (din-A4) (E2) | |
| | | Planificación del proyecto | 30 m | | | | |
| | 2 | Programación en MPI | 90 m | | 3 h | | |
| | 3 | Programación en MPI | 70 m | Estudio programación en MPI | | | |
| | | Delimitación de tareas concretas: puzle | 20 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 de informe y de la presentación del puzle | | | |
| | 9 | Reunión de grupo: puesta en común | 80 m | Ejercicios de representación gráfica | | | |
| | | Enunciado de los ejercicios de representación gráfica | 10 m | | | | |
| 21-23/03 | | | | reparación de informe y de la presentación del puzle | 8 h | | |
| | | | | Ejercicios de representación gráfica | | | |
| 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), presenta- | |
| | 11 | Puzle: presentación | 90 m | | | ción (E3.2), acta de eval. (E4.1) | [ 1 p. (E3.2, E4.1) ] |
| | 12 | Puzle: debate aclaratorio | 70 m | Implementación de la aplicación (Fase 1) | | Portafolio para revisión (E7) | |
| | | Enunciado de la aplicación (Fase 1) | 20 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 | | | | Desarrollo de la aplicación | 6 h | | |
| 25-29/04 | 16 | Trabajo: desarrollo de la aplicación | 90 m | Desarrollo de la aplicación | 4 h | | |
| | 17 | Teoría: com. punto a punto, deadlock | 60 m | | | Resultados preliminares: Fase 1 | |
| | | Teoría: Jumpshot | 30 m | | | de la aplicación (E6.1) | |
| | 18 | Debate de resultados preliminares | 15 m | Desarrollo de la aplicación (Fase 2) | | | |
| | | Enunciado de la aplicación (Fase 2) | 15 m | | | | |
| | | Teoría: topologías / anillo. Ejercicio demo: MPI+OpenMP. | 60 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 | Trabajo: desarrollo de la aplicación, documentación | 90 m | Desarrollo / documentación | 4 h | | |
| | 23 | | 90 m | | | | |
| | 24 | | 90 m | | | | |
| 16-20/05 | | | | Preparación de la presentación / Estudio | 8 h | Informe técnico: Fases 1 y 2 (E6.2) | 3 p. |
| 24/05 | 25 | Defensa de la aplicación desarrollada | 60 m | | 3 h | Acta: evaluación de la pres. (E4.2) | [ 1 p. (E6.3, E4.2) ] |
| | | Examen de conocimientos mínimos | 120 m | | | Examen (E5) - Portaf. (E7, filtro) | 2 p. (E.5) |

## Fase previa al puzle: el *cluster* y conceptos básicos de MPI

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

### El *cluster* de trabajo: máquinas y directorios

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 las mismas usando `ssh` pero sin que se pida *password*. Para ello, hay que haber entrado una primera vez en cada máquina, para que nos reconozca como "usuarios autorizados". Para ello,

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 activas (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 optimiz.; 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 nodo 00 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`)

- **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

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.

## 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.

### MPI: Puzle (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 del vector y el desplazamiento desde el origen del vector al comienzo de cada trozo, en estos dos casos:

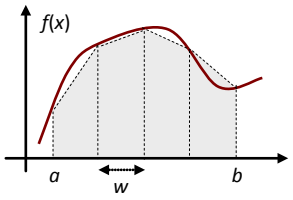
**a.** los posibles restos se añaden al último trozo

**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 MPI `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 (= ln x+1 + atan x), 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 MPI `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
```

# MPI: Puzle (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 P1 envíe a P2 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, P1 empaqueta los datos y envía el paquete a P2; por su parte, P2 recibe el mensaje y desempaqueta los datos.

Ejecuta el programa con 4 procesos. Debe imprimir:

```
Received in P2: 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.

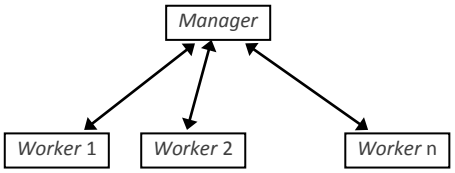
Completa el programa `P23-groups0.c` para que realice esa función. Ejecuta el programa con 8 procesos; debe imprimir:

```
I am pid: 0 and pid2: 0 and W(0,5,10,15) are 0 1 2 3
I am pid: 4 and pid2: 0 and W(0,5,10,15) are 4 5 6 7
```

# MPI: Puzle (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.

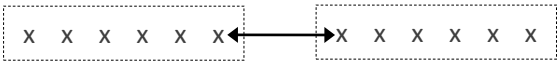



En esos casos, es habitual utilizar el conocido modelo de programación "*manager/worker*", en el que uno de los procesos, el *manager*, se encarga, por un lado, de repartir tareas bajo demanda y, por otro, de recoger resultados. El resto de procesos, los *workers*, solicitan tareas, las ejecutan, y devuelven 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 de ese programa. En la primera, se reparten las tareas (procesar los números) entre todos los procesos de modo estático consecutivo, procesando cada uno de ellos 320/`npr` números consecutivos.

En la segunda, el reparto de tareas debe ser dinámico, bajo demanda. Uno de los procesos (P0, por ejemplo) funciona como *manager* y reparte a cada uno de 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 versión 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 el comportamiento de ambos parámetros en función del número de procesos y explica los resultados.

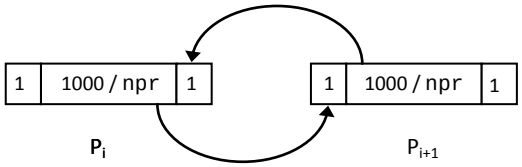
**P3.2** En una determinada aplicación (`P32-convoser.c`) se procesa un vector de `N` = 1.002 elementos al que se aplica una operación de convolución de manera iterativa

```
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. Como puedes ver, 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 un versión paralela del programa en el que **(a)** P0 reparte los 1.000 elementos el vector que hay que procesar, en trozos de 1000/`npr` elementos: el primero lo procesará él mismo, el segundo P1…; **(b)** cada proceso efectúa la operación sobre 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 A procesado, para que éste los reconstruya.

Ten en cuenta que 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 (que van a ir cambiando iteración a iteración).



## Ejercicios complementarios

Aquí tienes unos ejercicios por si quieres hacer alguna prueba 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:

a. hay que recibir y procesar los mensajes en orden estricto de `pid` (P1, luego P2…).
b. 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:

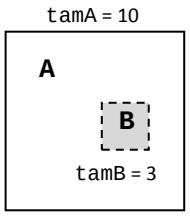
```
double A[N][N], B[N], C[N], D[N], PE

C[N] = A[N][N] * B[N]
D[N] = A[N][N] * C[N]
PE = ∑(C[i]· D[i])
```

Al principio se pide el tamaño de los vectores, N, tras lo cual reserva 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?

## 3.9 Aplicación a paralelizar

### Aplicación a paralelizar usando MPI

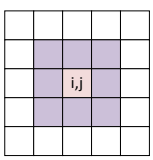
Tras la fase de aprendizaje desarrollada en formato "puzle" disponemos ya de las herramientas y estrategias necesarias para afrontar con éxito la paralelización de una determinada aplicación. Como vimos en la presentación del proyecto, se trata de una aplicación que simula el comportamiento térmico de 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.

#### ▪ Descripción de la aplicación

El programa `heats.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:

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



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`

**>> difussion**

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

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

  // thermal difussion
  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
```

**>> 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);
```

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);
  ...

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

    // initial values for grids
    init_grid_chips (conf, param, 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 la variable `param`, un *struct* con los siguientes campos:

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

```
struct info_chip {
  int    x, y, h, w;
  float  tchip;
};
```

| | |
|---|---|
| `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. |
| `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 de la simulación: diferencias de temperatura media menores que ese valor. |
| `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.

`chips`: un *struct* con la información de cada chip: posición en la tarjeta base (x,y), tamaño (h,w), y temperatura (`tchip`).

## ▪ 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
```
Núm. de configuraciones a simular (3); núm. de chips (4); temp. ambiente (20.0); temp. máxima de un chip (160.0); criterios de convergencia: temperatura (0.01) y núm. máximo de iteraciones (10000); factor de escala (10).

```
40  40  100.0
50  20  160.0
30  60  120.0
20  20  80.0
```
Una línea con la definición de cada chip de la tarjeta: por ejemplo, el primero, tamaño (40x40) y temperatura máxima (100.0)

```
86  15
135  49
21  27
90  59
```
Primera configuración a simular: coordenadas (x,y) del vértice superior izquierdo de cada chip. P.e., 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
26  72
168  29
62  23
```
Segunda configuración a simular: coordenadas (x,y) del vértice superior izquierdo de cada chip.

```
67  35
129   2
22  11
119  84
```
Tercera configuración a simular: coordenadas (x,y) del vértice superior izquierdo de cada chip.

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:

```
> heats 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 (que puedes comparar con la inicial, que se encuentra en el fichero `card0_ser.chips`).

## ▪ Estructura del programa

El programa serie está dividido en tres módulos: `heats.c` (programa principal), `difussion.c` (que contiene las rutinas `thermal_update` y `difussion`), y `faux.c` (con las rutinas auxiliares de lectura de datos y generación de resultados).

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

## ▪ 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 graficas, 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.

**>> Entregables**

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 realizado, y entrega de la carpeta final. Fecha: **24 de mayo**.

## 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)
#define NCOL (CSIZE*param.scale + 2)

struct info_chip {
  int    x, y, h, w;
  float  tchip;
};

struct info_param {
  int    nconf, nchip;
  float  t_ext, tmax_chip, t_delta;
  int    max_iter, scale;
  struct info_chip **chips;
};

struct temp {
  double Tmean;
  int    conf;
  float  bgrid[MAX_GRID_POINTS];
  float  cgrid[MAX_GRID_POINTS];
};
```

```
/* File: heats.c */

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

#include "defines.h"
#include "faux.h"
#include "difussion.h"


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


/*****************************************************************************/
void init_grid_chips (int conf, struct info_param param, 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=param.chips[conf][n].x*param.scale; i<(param.chips[conf][n].x+param.chips[conf][n].h)*param.scale; i++)
  for (j=param.chips[conf][n].y*param.scale; j<(param.chips[conf][n].y+param.chips[conf][n].w)*param.scale; j++)
    grid_chips[(i+1)*NCOL+(j+1)] = param.chips[conf][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[])
{
  int    conf;
  struct info_param param;

  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);

  printf ("\n  ======================================================================");
  printf ("\n    Thermal difussion - 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, 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);
}
```

```
/* 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 difussion
    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)
{
  int i, j, h, w;
  float tchip;
  FILE *fdin;

  fdin = fopen (file_name, "r");

  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);
  }

  param->chips = (struct info_chip **) malloc(param->nconf*sizeof(struct info_chip*));
  for (i=0; i<param->nconf; i++)
    param->chips[i] = (struct info_chip *) malloc(param->nchip * sizeof(struct info_chip));

  // chip sizes and temperatures
  for (j=0; j<param->nchip; j++)
  {
    fscanf (fdin, "%d %d %f", &h, &w, &tchip);
    for (i=0; i<param->nconf; i++)
    {
      param->chips[i][j].h = h;
      param->chips[i][j].w = w;
      param->chips[i][j].tchip = tchip;
    }
  }

  // chip positions
  for (i=0; i<param->nconf; i++)
  for (j=0; j<param->nchip; j++)
    fscanf (fdin, "%d %d", &param->chips[i][j].x, &param->chips[i][j].y);

  fclose (fdin);
}


/*****************************************************************************/
void results_conf (int conf, struct info_param param, float *grid, float *grid_chips, struct temp *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+1;
    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 temp *BT, char *finput)
{
  FILE  *fd;
  char  name[100];

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