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).

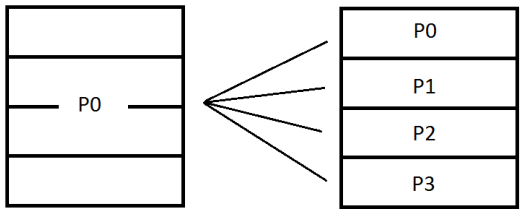
1.2.1 Lectura de los datos:

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.



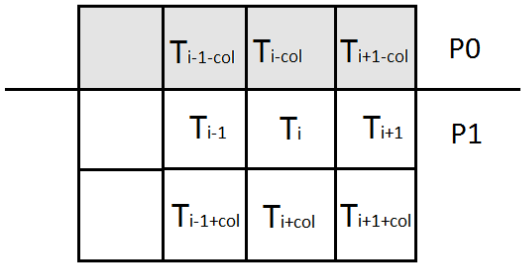
Primero cada proceso calcula los vectores de tamaño y desplazamiento para saber cuantos datos le corresponden y cual es su primer dato. De esta manera, para cada configuración, cuando el proceso líder construya la placa de chips podrá realizar una llamada colectiva Scatter y distribuir cada trozo a su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

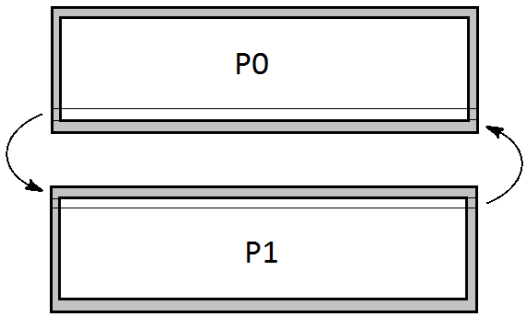
Notese que en la función colectiva, se comienza a enviar desde NCOL, esto es, desde la segunda línea, y equivalentemente cada proceso recoge los datos comenzando desde la segunda línea. En el siguiente apartado se explica el motivo de esto.

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.

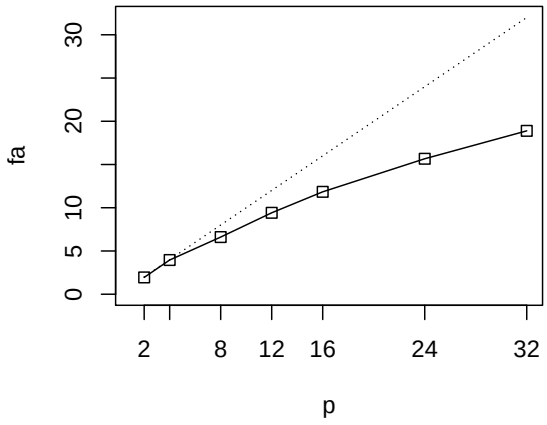
```
// convergence every 10 iterations
if (niter % 10 == 0){

    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;
}
```

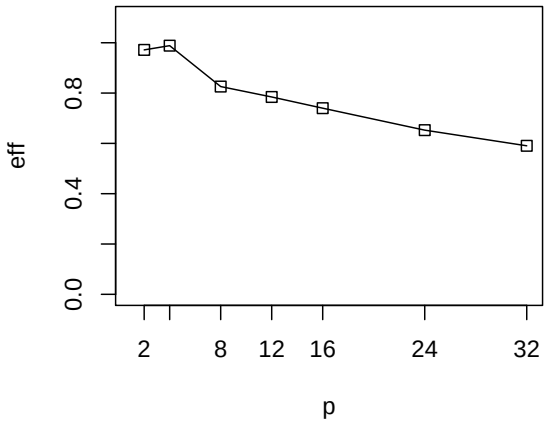
1.2.5 Resultados de la paralelización

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.

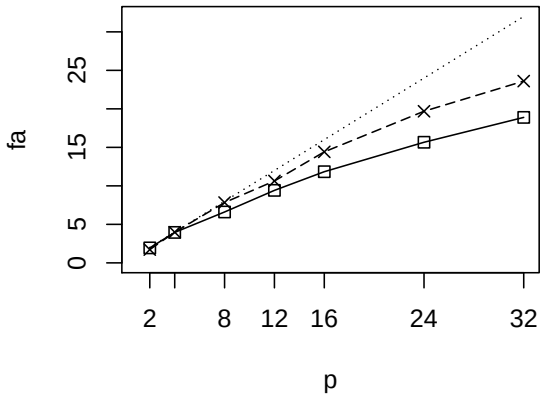
Ahora se procede a realizar el cálculo directamente, obviando la primera y última líneas, que se calcularán al final del todo, tras comprobar que la comunicación está realizada.

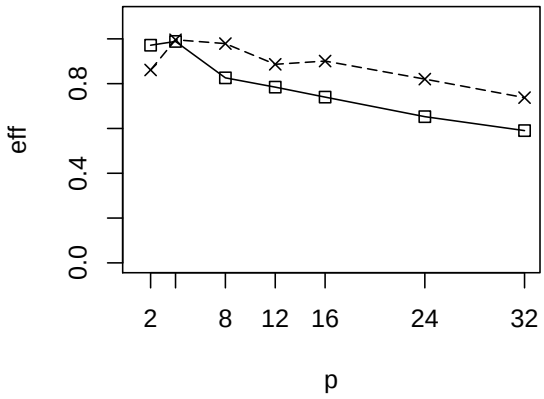
El modelo de paso de mensajes implica que el emisor inicia la comunicación. Tal y como se indica en [4], las comunicaciones tendrán en general menores costos si las funciones Receive ya han sido publicadas en el momento en que el emisor inicia la comunicación, los datos pueden ser movidos directamente 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.

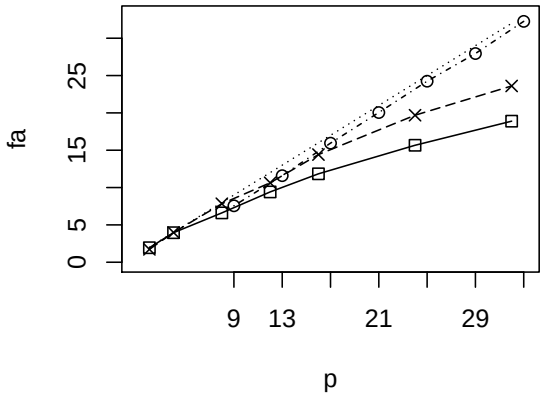
Proc.	**T**(s)	**speed-up**	**efficiency**
serial	2400	-	-
9	317.61 +- 0	7.556	0.84
13	207.03 +- 0	11.593	0.89
17	150.49 +- 0	15.948	0.94
21	119.71+- 0	20.049	0.95
25	99 +- 1	24.243	0.97
29	85.87 +- 0	27.948	0.96
33	74.4 +- 0	32.258	0.98

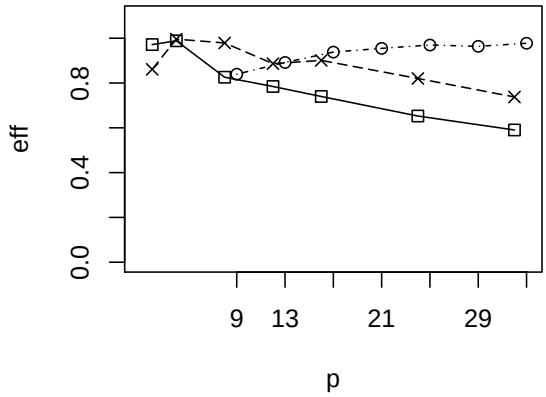
Los resultados de esta última muestran que el nuevo modelo es muy superior al anterior ya que su eficiencia aumenta con P y alcanza factores de aceleración casi óptimos. Además podemos intuir de los valores de la eficiencia que alcanza su máximo o se acerca a el con los 33 procesadores(4 grupos de 8), 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/dissipation 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;
        }
    }
}
```