Facultad de Informática

UPV/EHU

Grado en Ingeniería Informática


# Sistemas de Cómputo Paralelo

Ingeniería de Computadores

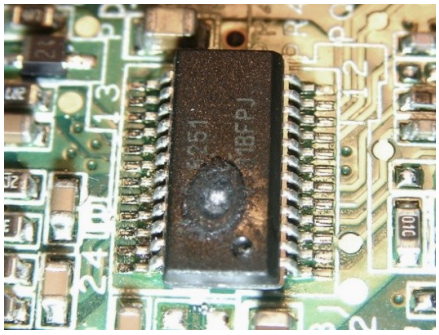
## Proyecto MPI

*La gente quiere explorar, llevar sus capacidades al límite, saber hasta dónde puede llegar. Pero en nuestra sociedad el gozo de la creación es un lujo reservado a unos pocos, los artistas, los artesanos, los científicos. Quien lo ha experimentado sabe que no es una cosa cualquiera. No hace falta descubrir la Teoría de la Relatividad ni nada parecido: todo el mundo puede disfrutar de esa experiencia. A veces basta con ver qué hace el prójimo. Enfrentarse a una demostración matemática sencilla como el Teorema de Pitágoras, que se estudia a los quince años, y entender de qué va el asunto, es algo apasionante. "Vaya, eso no se me había ocurrido nunca". Pues bien, eso también es un proceso creativo aunque el teorema tenga más de dos mil años.*

*Uno no deja de sorprenderse de las maravillas que va descubriendo, porque uno las "descubre" aunque alguien ya lo haya hecho antes. Y si además uno consigue aportar su granito de arena, eso es aún más apasionante.*

Noam Chomsky

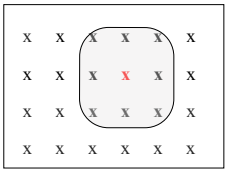
# ¡Cuidado, que quema!



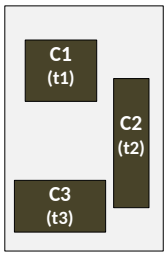
La empresa TXIPSA se dedica a la fabricación de placas de circuitos impresos para diversos usos. Estas placas contienen varios chips que se calientan a diferentes temperaturas en su uso normal, por lo que deben ser colocados de forma estratégica en la placa para reducir la temperatura global del sistema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

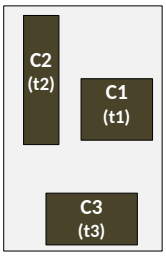
Hasta ahora, la simulación se hace en serie analizando el comportamiento de diferentes configuraciones, hasta obtener la mejor solución. Para aplicar el algoritmo de difusión del calor, se divide la tarjeta en una rejilla bidimensional de puntos, y se calcula la temperatura de cada punto teniendo en cuenta la **temperatura de los puntos vecinos.** Se repite la operación, iteración tras iteración, hasta que el sistema converge a una determinada temperatura media, que depende de la posición de los chips de la tarjeta (las fuentes de calor).



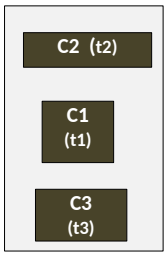
La siguiente figura muestra tres posibles configuraciones de una tarjeta de circuito impreso con tres chips, en las que, tras el proceso de difusión del calor, se alcanzan diferentes temperaturas medias en la tarjeta.




Config. 1 – $T_m1$




Config. 2 – $T_m2$



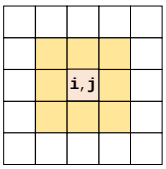

Config. 3 – $T_m3$

Recientemente TXIPSA ha adquirido un *cluster* de **memoria distribuida**, con los nodos conectados mediante Infiniband, 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.**

## Descripción de la aplicación

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




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

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

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.

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.

**>> calculate_Tmean**

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

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

  // thermal diffusion
  Tfull = thermal_diffusion (param, grid, grid_aux);

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

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`.

**>> 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
a = 0.44*(NCOL-2) + 1;
b = 0.56*(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);
```

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.

**>> thermal_diffusion**

```
for (i=1; i<NROW-1; i++)
for (j=1; j<NCOL-1; j++){
  T = grid[i*NCOL+j] + 0.10 * (
      grid[(i-1)*NCOL + j-1] + grid[(i-1)*NCOL + j] + grid[(i-1)*NCOL + j+1] +
      grid[    i*NCOL + j-1] + grid[    i*NCOL + j+1] +
      grid[(i+1)*NCOL + j-1] + grid[(i+1)*NCOL + j] + grid[(i+1)*NCOL + j+1]
      - 8*grid[i*NCOL + j] );
  grid_aux[i*NCOL+j] = T;
  Tfull += 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];
return (Tfull);
```

**El programa principal** es sencillo:

```
{ ...
  read_data (argv[1], &param, &chips, &chip_coord);
  clock_gettime (CLOCK_REALTIME, &t0);
  ...
  // loop to process chip configurations
  for (conf=0; conf<param.conf_kop; conf++){
    // inintial values for grids
    init_grid_chips (conf, param, chips, chip_coord, grid_chips);
    init_grids (param, grid, grid_aux);
    // main loop: thermal injection/disipation until convergence (t_delta or max_iter)
    Tmean = calculate_Tmean (param, grid, grid_chips, grid_aux);
    printf ("  Config: %2d    Tmean: %1.2f\n", conf + 1, Tmean);
    // processing configuration results
    results_conf (conf, Tmean, param, grid, grid_chips, &BT);
  }
  clock_gettime (CLOCK_REALTIME, &t1);
  tej = (t1.tv_sec - t0.tv_sec) + (t1.tv_nsec – t0.tv_nsec)/(double)1e9;
 printf ("\n\n >>> Best configuration: %2d   Tmean: %1.2f\n", BT.conf + 1, BT.Tmean);
 printf ("   > Time (serial): %1.3f s \n\n", tej);
 // writing best configuration results
 results (param, &BT, argv[1]);
  ...
}
```

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

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

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

`nconf`: número de configuraciones diferentes que hay que simular; cada configuración está compuesta por los mismos chips, pero en diferentes posiciones de la tarjeta.

`nchip`: número de chips que tiene la tarjeta.

`max_iter`: criterio de finalización de la simulación: número máximo de iteraciones.

`scale`: factor de escala de la simulación (de 1 a 12); el valor 1 representa una rejilla de 200×100 puntos, que usaremos para las pruebas durante el desarrollo de la aplicación; un valor 10 representa una rejilla de 2000×1000 puntos, y es el que utilizaremos para la obtención de resultados, una vez programada la aplicación.

`t_ext`: temperatura ambiente a la que se inicializa la rejilla de puntos.

`tmax_chip`: temperatura máxima de los chips.

`t_delta`: criterio de finalización de la simulación: diferencias de temperatura media menores que ese valor.

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

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

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

### ▪ Definición de la tarjeta

El **fichero de entrada** que define la tarjeta, y de donde se leen los datos de partida, está dividido en tres bloques (lo que sigue es un ejemplo):

```
12  3   4   20.0  160.0  0.01  10000
```

**1er bloque de datos** (`param`), parámetros generales de la simulación: factor de escala (10); nº de configuraciones a simular (3); nº de chips (4); temperatura ambiente (20.0); temperatura máxima de un chip (160.0); criterios de convergencia: temperatura (0.01) y nº máximo de iteraciones (10000).

```
40  40  100.0
50  20  160.0
30  60  120.0
20  20  135.0
```

**2º bloque de datos** (`chips`), definición de los chips de la tarjeta, una línea por cada chip; por ejemplo: 4 chips, el primero de tamaño 40×40 y temperatura máxima 100.0; etc.

```
86  15
135  49
21  27
90  59
```

**3er bloque de datos** (`chip_coord`): coordenadas (`x`, `y`) del vértice superior izquierdo de cada chip de una configuración. P. ej., primer chip: 86, 15; al ser de tamaño 40×40 ocupará las posiciones 86-125 y 15-54 en la rejilla básica de 200×100 puntos.

```
126  40
26  72
168  29
62  23
```

Segunda configuración.

El fichero `card` contiene la descripción de las configuraciones que hay que simular. Se trata de 20 configuraciones diferentes de 4 chips, con una rejilla de 2400×1200 puntos (factor de escala 12).

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, 200×100 puntos.

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

```
~$ heat_s card0
```

### ▪ Resultados

Como resultado de la simulación, se guardan 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 200×100 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`). Por supuesto, puedes utilizar cualquier otro software para representar la matriz gráficamente.

[ Nota: para abrir una terminal gráfica desde linux, conectate mediante el comando `ssh -X` ; desde Windows hay que ejecutar un cliente previamente, `Xming` por ejemplo. ]

### ▪ Estructura del programa

El programa serie está dividido en tres módulos: `heat_s.c` (programa principal), `diffusion.c` (que contiene las rutinas `calculate_Tmean, thermal_diffusion y thermal_update`), y `faux.c` (con las rutinas auxiliares de lectura de datos `read_data` y generación de resultados). El fichero `defines.h` contiene algunas definiciones de constantes y estructuras de datos. Todos los ficheros (módulos fuente `.c`, ficheros de cabecera `.h`, y definición de tarjetas) los tienes en el directorio `SCP/proyecto`.

Para generar el ejecutable hay que compilar los tres programas; por ejemplo:

```
> icc -o heat_s heat_s.c diffusion.c faux.c      (en paralelo, mpicc)
```

Cambia el nombre de las versiones paralelas de los ficheros —por ejemplo, `heat_p.c`— así como el de los ficheros que se generan (en `faux`), ya que tendrás que comparar los resultados obtenidos en serie y en paralelo.

## Tareas a realizar

Hay que paralelizar el programa serie para poder ejecutarlo en el *cluster*. Se pide 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, en cada iteración cada proceso va a necesitar datos, los correspondientes a las fronteras, que se encuentran en los procesos `pid+1` y `pid-1`. Para esa operación, utiliza en una primera versión, funciones de comunicación síncrona (`Ssend` y `Recv`); luego, antes de pasar a la segunda fase, optimiza los resultados utilizando funciones de comunicación inmediata (`Isend` y `Irecv`).

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, 12, 16, 20, 24, 32 y 48 procesos. Obtén los tiempos de ejecución, y calcula los *speed-ups* y eficiencias que consigues. Representa gráficamente esos datos y extrae las conclusiones pertinentes. Basándote en los resultados, estima el número de procesos (*P*) óptimo para este problema en este *cluster*.

**Nota**. Los procesadores de la máquina `dif-cluster` y de los nodos del *cluster* no son iguales; por lo tanto, para que la comparación sea "correcta", conectate mediante `ssh` a un nodo del *cluster* y ejecuta ahí la versión en serie.

### ▪ Fase 2

Una vez completada la primera, hay que realizar una segunda versión del programa paralelo 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 mismo valor *P* estimado en la primera fase), y efectuar una planificación dinámica de las tareas (tal y como hemos hecho en un ejercicio anterior).

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* procesos, 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. Fechas y evaluación.

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

La **primera fase del proyecto es obligatoria**, y la **fecha límite es el 8 de mayo.** Para ese día, tienes que conseguir un resumen de los resultados principales de la fase 1 y discutirlo con la profesora, antes de empezar con la 2ª fase. **Si no**, se tiene que seguir con la primera fase los días que queden.

El informe técnico final se debe entregar para el **24 de mayo** (eGela).

El proyecto total se evaluará de esta forma:

> Grupo
> - aplicación: % 75
> - informe técnico: % 25

## ¿Quieres mejorar?

Si tienes interés y tiempo (ten en cuenta que el resto de asignaturas están ahí, y que seguramente son igual de interesantes), o continuación se muestran propuestas para hacer vuestro proyecto más atractivo, y, de paso, conseguir puntos extra (dependiendo de lo hecho).

1. La arquitectura del *cluster* is híbrida: ciertos nodos de memoria repartida (52 nodos), en los cuales hay 2 procesadores de 4 nodos. Por lo tanto, se puede combinar MPI y OpenMP.

   Crea una versión de tu aplicación que convine las dos API-s. Verifica que el resultado es correcto, y compara lo siguiente: tiempo de ejecución en 32 procesos, utilizando solo MPI; y el tiempo de ejecución en 32 procesos: 8 procesos MPI x 4 procesos OpenMP.

2. Para hacer la evolución de la temperatura, se utilizan las matrizes `grid` y `grid_aux`: las nuevas temperaturas se cargan en la matriz `grid_aux`, y al final, se cargan los datos en la matriz `grid`. Evita hacer las copias alternando dichas matrices de iteración a iteración: en una iteración se parte de `grid` y se genera `grid_aux`, y en la siguiente al revés. Compara los tiempos de ejecución de esta nueva versión tanto en serie como en paralelo.

3. Se puede hacer un vídeo con la evolución de la temperatura. Por ejemplo, se puede guardar la matriz de temperaturas cada 10 iteraciones y, luego, crear un video utilizando los colores y aplicaciones que quieras.

4. Cualquier propuesta que se te ocurra acerca del proyecto.

## ▪ Código de la versión serie

```
/*************************************************************
 * defines.h
 * definitions for the thermal simulation of the card
*************************************************************/

// minimal card and maximum size
#define RSIZE 200
#define CSIZE 100

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


struct info_param {
  int    nconf, nchip, max_iter, scale;  // num. of configurations to test, num. of chips,
max. num. iterations, card size scale
  float  t_ext, tmax_chip, t_delta;      // external temp., max. temp. of a chip, temp.
incr. for convergence
};

struct info_chips {
  int    h, w;    //  size (h, w)
  float  tchip;   // temperature
};


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

```
/*   heat_s.c
     Difusion del calor en 2 dimensiones      Version en serie
*****************************************************************************************/

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

#include "defines.h"
#include "faux_s.h"
#include "diffusion_s.h"

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

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

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

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

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

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

  float *grid, *grid_chips, *grid_aux;
  struct info_results BT;

  int    conf, i;
  struct timespec t0, t1;
  double tej, Tmean;

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

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

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

  clock_gettime (CLOCK_REALTIME, &t0);

  grid = malloc(NROW*NCOL * sizeof(float));
  grid_chips = malloc(NROW*NCOL * sizeof(float));
  grid_aux = malloc(NROW*NCOL * sizeof(float));

  BT.bgrid = malloc(NROW*NCOL * sizeof(float));
  BT.cgrid = malloc(NROW*NCOL * sizeof(float));
  BT.Tmean = MAXDOUBLE;

  // loop to process chip configurations
  // ===================================

  for (conf=0; conf<param.nconf; conf++)
  {
    // inintial values for grids
    init_grid_chips (conf, param, chips, chip_coord, grid_chips);
    init_grids (param, grid, grid_aux);

    // main loop: thermal injection/disipation until convergence (t_delta or max_iter)
    Tmean = calculate_Tmean (param, grid, grid_chips, grid_aux);
    printf ("  Config: %2d    Tmean: %1.2f\n", conf + 1, Tmean);

    // processing configuration results
    results_conf (conf, Tmean, param, grid, grid_chips, &BT);
  }

  clock_gettime (CLOCK_REALTIME, &t1);
  tej = (t1.tv_sec - t0.tv_sec) + (t1.tv_nsec - t0.tv_nsec)/(double)1e9;
  printf ("\n\n >>> Best configuration: %2d    Tmean: %1.2f\n", BT.conf + 1, BT.Tmean);
  printf ("   > Time (serial): %1.3f s \n\n", tej);
  // writing best configuration results
  results (param, &BT, argv[1]);


  free (grid);free (grid_chips);free (grid_aux); free (BT.bgrid);free (BT.cgrid); free (chips);
  for (i=0; i<param.nconf; i++) free (chip_coord[i]);
  free (chip_coord);

  return (0);
}
```

```
/* diffusion_s.c
*********************************************************************************************/
#include "defines.h"

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

  // 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
  a = 0.44*(NCOL-2) + 1;
  b = 0.56*(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);
}

double thermal_diffusion (struct info_param param, float *grid, float *grid_aux)
{
  int     i, j;
  float   T;
  double  Tfull = 0.0;

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

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

  return (Tfull);
}

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

  end = 0; niter = 0;
  while (end == 0)
  {
    niter ++;
    Tmean = 0.0;
    // heat injection and air cooling
    thermal_update (param, grid, grid_chips);
    // thermal diffusion
    Tfull = thermal_diffusion (param, grid, grid_aux);
    // convergence every 10 iterations
    if (niter % 10 == 0) {
      Tmean = Tfull / ((NCOL-2)*(NROW-2));
      if ((fabs(Tmean - Tmean0) < param.t_delta) || (niter > param.max_iter))
           end = 1;
      else Tmean0 = Tmean;
    }
  }
  printf ("Iter: %d\t", niter);
  return (Tmean);
}
```

```
/*  faux_s.c
*****************************************************************************************/

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

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

  fdin = fopen (file_name, "r");
  if (fdin == NULL) {
    printf ("\n\nERROR: input file \n\n");
    exit (-1);
  }
  // simulation parameters
  fscanf (fdin, "%d %d %d %f %f %f %d", &param->scale, &param->nconf, &param→nchip,
                       &param→t_ext, &param->tmax_chip, &param->t_delta, &param->max_iter);
  if (param->scale > 12) {
    printf ("\n\nERROR: maximum scale factor is 12 \n\n");
    exit (-1);
  }
  // chip sizes and temperatures
  *chips = (struct info_chips *) malloc (param->nchip * sizeof(struct info_chips));
  for (i=0; i<param->nchip; i++) {
    fscanf (fdin, "%d %d %f", &h, &w, &tchip);
    (*chips)[i].h = h;
    (*chips)[i].w = w;
    (*chips)[i].tchip = tchip;
  }
  // chip positions
  *chip_coord = malloc (param->nconf * sizeof(int*));
  for (i=0; i<param->nconf; i++) (*chip_coord)[i] = malloc (2 * param->nchip * sizeof(int));
  for (i=0; i<param->nconf; i++)
  for (j=0; j<param->nchip; j++)
    fscanf (fdin, "%d %d", &(*chip_coord)[i][2*j], &(*chip_coord)[i][2*j+1]);
  fclose (fdin);
}

/*****************************************************************************************/
void results_conf (int conf, double Tmean, struct info_param param, float *grid,
                      float *grid_chips, struct info_results *BT)
{
  int  i, j;
  if (BT->Tmean > Tmean) {
    BT->Tmean = Tmean;
    BT->conf = conf;
    for (i=1; i<NROW-1; i++)
    for (j=1; j<NCOL-1; j++) {
      BT->bgrid[i*NCOL+j] = grid[i*NCOL+j];
      BT->cgrid[i*NCOL+j] = grid_chips[i*NCOL+j];
    }
  }
}

/*****************************************************************/
void fprint_grid (FILE *fd, float *grid, struct info_param param)
{
  int  i, j;
  // j - i order for better visualitation
  for (j=NCOL-2; j>0; j--) {
    for (i=1; i<NROW-1; i++) fprintf (fd, "%1.2f ", grid[i*NCOL+j]);
    fprintf (fd, "\n");
  }
  fprintf (fd, "\n");
}
```

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

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

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

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

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

  sprintf (name, "%s_ser.chips", file_name);
  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);
 }
```