Proyecto Final
Cómputo Paralelo

Andrés Alejandro Suro Aguirre
andres.suro@cimat.mx
Universidad de Guanajuato
20 de Mayo , 2023

---

1) La ecuación de Calor: La ecuación de Calor es una ecuación diferencial parcial usada en el área de la física y las matemáticas, fue desarrollada por primera vez por Joseph Fourier en 1822 para evaluar como el calor se disipa en una superficie respecto al tiempo. Es una de las ecuaciones mas estudiadas en el área de las ecuaciones diferenciales parciales. También se considera de gran interés en el área de la probabilidad, estando conectada con el estudio de caminatas aleatorias.

La ecuación de calor unidimensional está dada por la expresión,

$$\begin{aligned}\frac{\partial u}{\partial t} &= \alpha \nabla^2 u \\ &= \alpha \frac{\partial^2 u}{\partial x^2}.\end{aligned}$$

La interpretación de esta ecuación es la siguiente.

En una dimensión podemos visualizar el dominio como una vara, la cual consiste de un intervalo $[a, b] \in \mathbb{R}$. La temperatura es una función en este intervalo, la cual definimos como $u(x)$, con $x \in [a, b]$, sin embargo, como la temperatura cambia en razón del tiempo, vemos que la función $u : \mathbb{R}^2 \to \mathbb{R}$, toma dos valores $u(x, t)$, siendo t el tiempo.

Como la función u cambia al moverse en el eje x y al avanzar en el tiempo, tendremos que

Podemos interpretar entonces la derivada de u con respecto a x,

$$\frac{\partial u}{\partial x}(x, t),$$

como que tan rápido cambia la temperatura al movernos en el espacio o en la barra unidimensional.

La derivada,

$$\frac{\partial u}{\partial t},$$

representa la razón de cambio de la temperatura en un punto $x \in [a, b]$ conforme avanza el tiempo.

La ecuación de calor esta escrita en termino de estas dos ecuaciones, y nos dice que el cambio de temperatura con respecto al tiempo depende, o es proporcional, al cambio de temperatura con respecto al espacio, i.e, es proporcional a la segunda ecuación diferencial parcial con respecto a x.

Para entender mejor la derivación de la ecuación, evaluemos un caso discreto, en el que tenemos un conjunto finito de valores. Sean x_1, x_2 y x_3 puntos en el intervalo $[a, b]$, con temperaturas u_1, u_2 y u_3 respectivamente. Lo que queremos comparar es el promedio entre u_1 y u_3 con u_2, es decir,

$$D = \frac{u_1 + u_3}{2} - u_2$$

Si $D > 0$, u_2 tenderá a aumentar su valor, y entre mas grande sea la diferencia mas rápido se calentara. Si D es negativa entonces la temperatura disminuirá con una razón proporcional a D. Entonces la derivada de u_2 es proporcional a el valor promedio de sus vecinos y a su propio valor. Escribimos la ecuación en el caso discreto,

$$\begin{aligned}\frac{\partial u}{\partial t} &= \alpha\left(\frac{u_1 + u_2}{2} - u_2\right) \\ &= \frac{\alpha}{2}\left((u_3 - u_2) - (u_2 - u_1)\right) \\ &= \frac{\alpha}{2}\left(\Delta u_2 - \Delta u_1\right) \\ &= \frac{\alpha}{2}\Delta\Delta u_1.\end{aligned}$$

lo cual , al traducirlo al caso continuo, proporciona la ecuación inicial, i.e,

$$\frac{\partial u}{\partial t} = \alpha\frac{\partial^2 u}{\partial x^2}.$$

Se puede pensar que las segundas derivadas miden como se compara el valor de la temperatura en un punto x_2 con el promedio de la temperatura de los puntos x_1 y x_3.

Lo anterior se debe a la Segunda ley de la termodinámica, la cual afirma que el calor fluirá, de cuerpos más calientes a cuerpos adyacentes más fríos , en proporción a la diferencia existente entre los cuerpos. El movimiento del calor en un cuerpo es influenciado por la masa del mismo y por un factor llamado, capacidad calorífica, la cual es especifica para cada materia.

Ahora para que la función u describa correctamente a la temperatura, debe cumplir tres restricciones, u debe ser una ecuación diferencial parcial, se deben cumplir ciertas condiciones de frontera y una condición inicial.

La razón por la que nos interesa la condición de la frontera, es el hecho que la segunda derivada aparece en la ecuación debido a que se requiere que cada punto tienda al promedio de sus dos puntos vecinos, pero en la frontera no hay puntos vecinos en ambos lados por lo que cada punto en la frontera tendera a su único punto vecino.

Por lo que la función debe cumplir que la pendiente en los puntos frontera sea 0 todo el tiempo, i.e,

$$u(a, t) = u(b, t) = 0,$$

para todo $t > 0$.

Para determinar la condición inicial de u, debemos determinar la temperatura de u en todo x cuando $t = 0$,

$$u(x, 0) = f(x), \quad a \le x \le b.$$

En nuestro caso trabajaremos con una superficie bidimensional, por lo tanto la ecuación de calor se modifica de la siguiente forma.

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + \alpha \frac{\partial^2 u}{\partial y^2} \tag{0.1}$$

Donde nuestro domino estará dado por $[0,1] \times [0,1]$.

Diferencias finitas: Es un método que es ampliamente usado para aproximar EDP's usando computadoras. Se deriva usando series de Taylor y la definición de derivadas y puede aproximar la primera y segunda derivadas de una función.

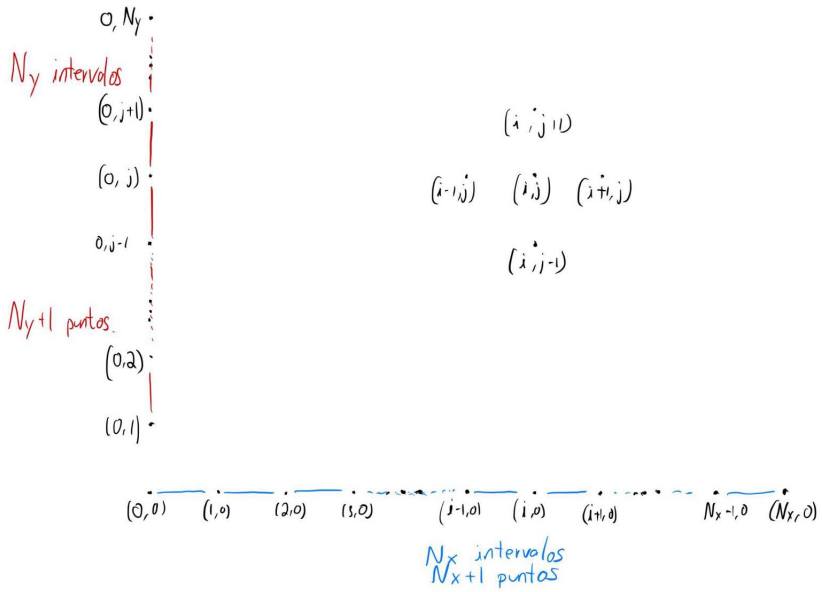
Primero que nada, debemos dividir la región, $0 \leq x \leq 1$ y $0 \leq y \leq 1$ en N_x y N_y sub-regiones respectivamente, cada una de tamaño

$$\Delta x = \frac{b_x}{N_x} = \frac{1}{N_x}$$

y

$$\Delta y = \frac{b_y}{N_y} = \frac{1}{N_y}$$

donde en cada el eje x, habrá $N_x + 1$ puntos y en el eje y habrá $N_y + 1$. La siguiente imagen ilistra esta división.



Recordamos que la derivada de una función se obtiene calculando el límite del cociente,

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x},$$

donde Δx representa un pequeño cambio en x, i.e, $\Delta x = (x + h) - x$.

Ahora, procedemos a reemplazar las derivadas por cocientes de diferencias. Asumiendo que la función puede ser diferenciada varias veces, obtenemos, por la expansión de Taylor y considerando los puntos $x + \Delta x$ y $x - \Delta x$, que

$$f(x + \Delta x) = f(x) + \Delta x f'(x) + \frac{\Delta x^2}{2!} f''(x) + O(\Delta x^3) \tag{0.2}$$

$$f(x - \Delta x) = f(x) - \Delta x f'(x) + \frac{\Delta x^2}{2!} f''(x) + O(\Delta x^3). \tag{0.3}$$

Reordenando (0.2) y dividiendo por Δx, llegamos a

$$f'(x) + O(\Delta x) = \frac{f(x + \Delta x) - f(x)}{\Delta x}. \tag{0.4}$$

Nuevamente, si restamos (0.3) de (0.2), y dividimos por $2\Delta x$, obtenemos

$$\frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x} = f'(x) + O(\Delta x^2). \tag{0.5}$$

Posteriormente, sumamos (0.2) y (0.3) con el fin de obtener la aproximación de $f''(x)$, la cual, al reordenar y dividir por por Δx conduce a

$$f''(x) + O(\Delta x^2) = \frac{f(x-\Delta x)-2f(x)+f(x+\Delta x)}{\Delta x^2}. \tag{0.6}$$

Estas diferencias son llamadas diferencias finitas centrales debido a la simetría de las aproximaciones de diferencias (0.5) y (0.6) en el punto x.

Ahora, para usar estas aproximaciones, debemos reemplazar las derivadas con los cocientes de diferencias. Para el caso de la derivada con respecto al tiempo tenemos

$$u(x,y,t\Delta t) = u(x,y,t) + \Delta t\frac{\partial u}{\partial t}(x,y,y) + \frac{\Delta t}{2!}\frac{\partial^2 u}{\partial t^2} + \cdots \tag{0.7}$$

Después de reordenar y dividir por Δt, llegamos a

$$\frac{\partial u}{\partial t} = \frac{u(x,y,t+\Delta t)-u(x,y,t)}{\Delta t} + O(\Delta t) \tag{0.8}$$

En el caso de las derivadas con respecto al espacio tenemos que, sin perdida de generalidad, tomando el eje x,

$$u(x+\Delta x,y,t) = u(x,y,t) + \Delta x\frac{\partial u}{\partial x}(x,y,t) + \frac{\Delta x^2}{2!}\frac{\partial^2 u}{\partial x^2} + \cdots \tag{0.9}$$

$$u(x-\Delta x,y,t) = u(x,y,t) - \Delta x\frac{\partial u}{\partial x}(x,y,t) + \frac{\Delta x^2}{2!}\frac{\partial^2 u}{\partial x^2} - \cdots. \tag{0.10}$$

Sumando y reordenando se llega a

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x+\Delta x,y,t)-2u(x,y,t)+u(x-\Delta x,y,t)}{\Delta x^2} + O(\Delta x^2), \tag{0.11}$$

análogamente,

$$\frac{\partial^2 u}{\partial y^2} = \frac{u(x,y+\Delta y,t)-2u(x,y,t)+u(x,y-\Delta y,t)}{\Delta y^2} + O(\Delta y^2). \tag{0.12}$$

Sustituyendo, (0.8), (0.11) y (0.12) en (0.1), obtenemos

$$\begin{aligned}\frac{u(x,y,t+\Delta t)-u(x,y,t)}{\Delta t} &= \alpha\left(\frac{u(x+\Delta x,y,t)-2u(x,y,t)+u(x-\Delta x,y,t)}{\Delta x^2}\right)\\ &+ \alpha\left(\frac{u(x,y+\Delta y,t)-2u(x,y,t)+u(x,y-\Delta y,t)}{\Delta y^2}\right)\\ &+ O(\Delta t,\Delta x^2,\Delta y^2).\end{aligned} \tag{0.13}$$

Finalmente, obtenemos la siguiente aproximación,

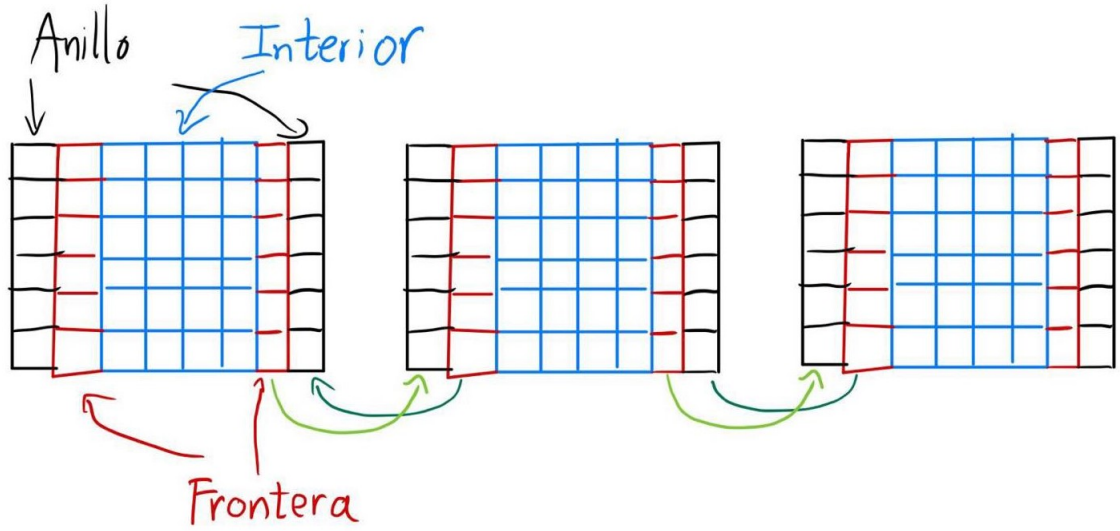
$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} = \alpha \frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} + \alpha \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^2}. \tag{0.13}$$

Idea para paralelizar : Dado nuestro dominio D, el cual es una malla $2D$, dividiremos D en subintervalos, donde cada sub intervalo le corresponde a un proceso. El beneficio se alcanza debido a que cada iteración de cada sub-dominio se realiza en paralelo. Si bien podríamos dividir el dominio en intervalos tanto en el eje x como en el eje y, para esta implementación solo usaremos una partición en el eje x, i.e, el plano $2D$ se dividirá en bloques y cada elemnto del bloque se actualiza en cada iteración del tiempo, usando los valores anteriores de la función de calor evaluada en el elemento actual y sus dos vecinos (en caso de los bloques en la frontera se usara a su único vecino).

El problema principal es el como se actualizara cada bloque si los puntos (nodos) de sus fronteras laterales necesitan información que se encuentra en otros bloques.

La solución es usar comunicación entre proceso vecinos, usando un proceso topológico cartesiano, debido a que los sub dominios serán rectángulos. Esta idea consiste en que cada bloque tendrá 3 partes, el anillo, la frontera y el interior. El interior se define como todos los nodos del bloque que tienen a todos sus vecinos dentro del mismo bloque. La frontera son aquellos elementos del bloque que tienen un vecino de otro bloque.

Finalmente, el anillo es una capa vertical de nodos del bloque vecino, el anillo se encuentra anidado a la izquierda o derecha de la frontera, dependiendo de donde este posicionado el bloque al que pertenece; los nodos de el anillo pertenecen a la frontera del bloque vecino. A continuación se muestra un dibujo del funcionamiento.



El código funciona separando la frontera del interior de tal forma que el interior se actualiza mientras que el anillo es copiado del otro bloque/procesos. La frontera del proceso actual se convierte en el anillo de otro proceso. Después de completar el calculo del anillo, es posible computar la frontera, pues esta es dependiente del interior y del anillo. Para logra esto, utilizamos un punto de sincronización antes de terminar la iteración actual de tiempo.

Los comandos principales que usaremos para poder paralelizar el código son los siguientes

a)`MPI_Comm_size(MPI_COMM_WORLD, p)`: Regresa el numero total de procesos en el comunicador especificado en la variable `p`.

b)`MPI_CART_CREATE(MPI_Comm comm, int ndimd, int *dims, int*perdios, int reorder, MPI_COMM *new)`: Se utiliza en el caso que un algoritmo realiza cálculos y comunicación en un a malla 2 o 3 dimensional, donde los puntos de la malla son asignados a diferentes procesos, y estos intercambian datos con sus vecinos. MPI hace esto definiendo topologías virtuales para intercomunicación, la cual puede ser usada para comunicar procesos dentro del grupo asociado.

`comm` es el comunicador inicial sin topología, `ndims` nos dice el numero de dimensiones en la malla que sera generada, `dims` es un arreglo de enteros de tamaño `ndims` tal que `dims[i]` es el numero de procesos en la dimension `i`. El producto de todas las entradas de `dim` es igual a el numero de procesos contenidos en el nuevo comunicador, `new_comm`.

El arreglo booleano `periods` de tamaño `ndims` dice, para cada dimensión, si la malla es periódica ó no. Si `reorder = true`, se le permite a el sistema de tiempo de ejecución reordenar procesos.

c)`MPI_CART_COORDS(MPI_COMM comm, int rank, int ndims, int *coords)`: Cuando se define una topología virtual, cada proceso tiene un rango, pero también una posición en al topología de la malla virtual , la cual se puede expresar por sus coordenadas cartesianas. MPI utiliza esta función para la traducción entre rangos de grupo y coordenadas cartesianas. Mas en especifico, `MPI_CART_COORDS`: traduce el rango de grupo dado por `rank` en coordenadas cartesianas en `coords` de arreglo de enteros. `ndims` es el numero de dimensiones de la malla virtual definida por el comunicador `com`.

d)`MPI_CART_SHIFT(MPI_COMM comm, int dir, int displ, int *rank_source, int *rank_dest)`: Las topologías virtuales se defines para facilitar la comunicación entre procesos asociados. En este caso entre un proceso y su proceso vecino en una dimensión especifica. Para determinar estos proceso vecinos, MPI usa a la función actual, donde `dir` especifica la dimensión para la que el proceso vecino debe ser determinada. `displ` determina la distancia al vecino, si `displ = -1`, se demanda a el vecino que precede al proceso, si `displ = 1` entonces al que le sigue. `rank_dest` contiene el rango de grupo de el proceso vecino en la dimensión y distancia especificada. El rango del proceso para el que, el proceso que hizo la llamada, es el proceso vecino en la dimensión y distancia especificada es regresado en `rank_source`.

e)`MPI_SENDRECV(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`: La forma mas básica de intercambio de comunicación entre dos proceso se provee por comunicación punto a punto. Dos proceso participan en este tipo de interacción, el proceso que envía la información ejecuta la función `MPI_Send` y el proceso receptor ejecuta la operación `MPI_Recv`. Sin embargo, en varias situaciones, un proceso puede enviar y recibir datos, ejecutando la función `MPI_SENDRECV`. Esta operación combina las operaciones de enviar y recibir en una llamada.

`sendbuf` especifida un buffer de envío en el que los elementos de los datos a enviar son guardados.

`sendcount` es el número de elementos a enviar.

`sendtype` es el tipo de datos de los elementos en el buffer de envío.

`dest` es el rango del proceso objetivo al que los datos son enviados.

`sendtag` es el tag para el mensaje que será enviado.

`recvbuf` es el buffer para los mensajes que serán recibidos.

`recvcount` es máximo numero de elementos a ser recibidos.

`recvtype` es el tipo de dato de los elementos que serán recibidos.

`source` es el rango del proceso del que el mensaje es esperado

`recvtag` es el tag del mensaje a ser recibido.

`comm` es el comunicador

`status` nos dice la estructura de datos que será guardada en el mensaje recibido.

Código Secuencial: La implementación secuencial fue como sigue,

Se determinaron los términos difusivos en cada eje, seguido de especificar los dominios, el lapso de tiempo a evaluar, el número de divisiones del tiempo y del eje x y y.

```
int threads = 4;
double kx = 1.0;// termino difusivo en x
double ky = 1.0;// termino difusivo en y

//////////////

double xI = 0; // Dominios en X y Y
double xF = 1;
double yI = 0;
double yF = 1;

/////////////////

double tI = 0; // tiempo inicial
double tF= 0.5; // tiempo final

////////////////

int Nt = 100000; // Num de divisiones en el tiempo
int Nx = 200; // Num de puntos en x, Razon : si se divide en n, hay n+1
    puntos
int Ny = 200; // Num de puntos en y, Razon : si se divide en n, hay n+1
    puntos
```

Posteriormente calculamos los "pequeños cambios" en cada eje, que no será mas que la distancia entre puntos vecinos en el eje x, dado por Δx, en el eje y, dado por Δy y en el tiempo, dado por Δt.

```
//Discretizacion

double Dx = (xF- xI)/(Nx - 1); //Dividimos entre el num de intervalos que
    habra, que es N-1 (N es el num de pts)
double Dy = (yF - yI)/ (Ny-1);// Mismo pero para el intervalo y
double Dt = (tF- tI)/ (Nt - 1);
double rx = kx*( Dt / pow(Dx,2) );// coeficientes
double ry = ky*( Dt / pow(Dy,2) );
```

En seguida se calculan dos vectores que contienen los puntos (coordenadas) de cada eje de la malla, donde la distancia entre cada punto esta dado por Δx o por Δy.

```
double *x = (double *)malloc(Nx * sizeof(double));
double *y = (double *)malloc(Ny * sizeof(double));

////// Condiciones iniciales
for (int i = 0; i < Nx; i++){
```

```
    x[i] = xI + (i-1)*Dx;
}

for (int j = 0; j < Ny; j++){
    y[j] = yI + (j-1)*Dy;
}
```

A continuación, declaramos las condiciones iniciales del modelo, es decir, la temperatura de cada nodo de la malla en el tiempo $t = 0$. En este caso, distribuimos la temperatura de la malla usando la función $\sin(x + y)^2$.

```
for (int i = 0; i < Nx; i++){
    for(int j = 0; j < Ny; j++){
        Uold[i][j] = ( sin( x(i) + y(j) ) ) * ( sin( x(i) + y(j) ) );
    }
}
Unew = Uold;
clone(Uold, Unew, Nx, Ny, threads);
```

Recordamos, que la temperatura en los puntos fronteras es invariante al cambio del tiempo, por lo tanto, antes de iniciar el ciclo principal, asignamos estos valores,

```
for (int j = 0; j < Ny; j++){
    Unew[0][j] = 0.0; // Oeste
    Unew[Nx - 1][j] = 0.0; // Este
}
for (int i = 0; i < Nx; i++){
    Unew[i][0] = 0.0; // Sur
    Unew[i][Ny] = 2.0; // Norte
}
```

Finalmente, realizamos el calculo del cambio de temperatura con respecto al tiempo en cada punto. Usamos la formula dada por (0.13). Donde en la matriz `Unew`, guardamos los valores de la temperatura en el tiempo actual y en `Uold` guardamos la temperatura de cada punto, obtenida en la iteración anterior.

```
// Ciclo principal
for (int k = 0; k < Nt; k++){
    for (int i = 1; i < (Nx - 1) - 1; i++ ){
        for (int j = 1; j < (Ny - 1) - 1; j++){
            double aC = 1 - 2 * (rx + ry);

            double aE = rx;
            double aW = rx;
            double aS = ry;
```

```
            double aN = ry;

            Unew[i][j] = aC * Uold[i][j] + aW*Uold[i-1][j]
            + aE * Uold[i+1][j] + aS * Uold[i][j-1]
            + aN * Uold[i][j+1];
        }
      }
      clone(Uold, Unew, Nx, Ny, threads);
  }
```

Donde el calculo de `Unew[i][j]` se deriva al despejar u^{n+1} en la ecuación (0.13). Recordemos que las condiciones de frontera no cambian, por lo que el ciclo correrá desde 2 hasta $N_x - 1$ en el eje x y desde 2 hasta $N_y - 1$ en el eje y.

Código Paralelo: El código paralelizado es muy similar a el secuencial, sin embargo los cambios mas importantes son aquellos en los que se establece el proceso de comunicación entre los procesos. Primero, debemos inicializar la región paralelo, para ellos usamos

```
    call MPI_INIT(ierr)
    call MPI_COMM_SIZE(MPI_COMM_WORLD,numtasks,ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD,taskid,ierr)
```

Seguido de esto, como deseamos utilizar una malla bidimensional, donde, después de dividirla en bloques en el eje x, cada bloque este asignado a un proceso diferente e intercambie información con sus vecinos, usamos la función

```
    .-------------------------------------------------.
!     |           MPI: Topologia (cartesiana)         |
!     .-------------------------------------------------.
      call MPI_CART_CREATE(MPI_COMM_WORLD,ndims,dims_vec, &
                           periodicite,reorganisation, &
                           comm2D,ierr)
```

Posteriormente, como estamos trabajando en el plano cartesiano, por lo expresado en la sección anterior, usamos la función

```
    call MPI_CART_COORDS(comm2D,taskid,ndims,coords,ierr)
```

para traducir el rango de cada proceso en coordenadas cartesianas.

El siguiente comando que debemos usar es

```
    call MPI_CART_SHIFT(comm2D,direction,displasment,&
                        vecino(1),vecino(2),ierr)
```

Esto para poder determinar los dos vecinos del proceso actual y así poder calcular en

anillo del proceso actual (los datos que recibe) y a que procesos se el enviara la frontera (elementos que se pegaran en el anillo de los vecinos).

A continuación, realizamos una partición de la malla, solamente en el eje x donde cada bloque corresponderá a un proceso. En esta sección también se determina si se le debe agregar una ó dos columnas a cada sección, la cual representara el anillo. Lo anterior se determina si el bloque está en la frontera, i.e, si está en la extrema izquierda o derecha.

```
     .-------------------------------------------------.
!    |            MPI: Divisin del dominio            |
!    .-------------------------------------------------.

!    ========================================
!     Divisin del dominio en x
     NN = floor(1.0d0*NxG/numtasks)
!    -------------
!    IMPRIMIR
     print*,'taskid=',taskid,', NN =',NN

     if (numtasks.eq.1) then
        Nx = NxG
     else
        if (taskid.eq.0) then
            Nx = NN
        elseif (taskid.eq.numtasks-1) then
            Nx = NxG - NN*taskid + 2
        else
            Nx = NN + 2
        endif
     endif
!    ========================================
!     Divisin del dominio en y
     Ny = NyG
```

Ahora, definimos un arreglo de índices globales, los cuales usaremos para comunicar los bloques.

```
    allocate(index_global(Nx))

     if (numtasks.eq.1) then
        do i=1,Nx
           index_global(i) = i
        enddo
     else
        if (taskid.eq.0) then
           do i=1,Nx
              index_global(i) = i
           enddo
```

```
      else
         do i=1,Nx
           index_global(i) = taskid*(NN)-1 + (i-1)
         enddo
      endif
   endif
```

A continuación, definimos el vector columna (la frontera) que estaremos compartiendo con otros procesos como un vector de tipo MPI, i.e,

```
call MPI_TYPE_VECTOR(Ny,1,Nx,MPI_DOUBLE_PRECISION,&
                     tipo_col,ierr)
  call MPI_TYPE_COMMIT(tipo_col,ierr)
```

En vista de lo anterior, nos resta calcular la malla. En este caso tendremos una malla global, que es calculada de igual forma que en la implementación secuencial.

```
Malla (GLOBAL)

  allocate(xG(NxG))
  allocate(yG(NyG))

  do i=1,NxG
     xG(i) = xI + (i-1)*Dx
  enddo
  do j=1,NyG
     yG(j) = yI + (j-1)*Dy
  enddo
```

Por otra parte, como cada proceso trabajara con una parte del dominio, debemos calcular una malla local. Recordemos que el vector de índices globales tiene los índices de cada bloque de la partición. Con él, podemos tomar valores de la malla global y definir mallas locales para cada proceso.

```
Malla (LOCAL)
  allocate(x(Nx))
  allocate(y(Ny))

  do i=1,Nx
     index = index_global(i)
     x(i) = xG(index)
  enddo
  do j=1,Ny
     y(j) = yG(j)
  enddo
```

Finalmente, el programa principal funciona de manera equivalente a su contra parte secuencial, sin embargo si existe un ligero cambio. En cada iteración temporal, debemos

comunicar los procesos, haciendo que los mismos reciban información relevante para realizar su calculo propio, y envíen información necesaria a otros procesos en sus cálculos correspondientes.

La función de MPI que puede lograr esto es

```
call MPI_SENDRECV(Unew(2,1) ,1,tipo_col,vecino(1),etiqueta, &
                  Unew(Nx,1),1,tipo_col,vecino(2),etiqueta, &
                  comm2D,statut,ierr)

call MPI_SENDRECV(Unew(Nx-1,1),1,tipo_col,vecino(2),etiqueta,&
                  Unew(1,1)  ,1,tipo_col,vecino(1),etiqueta,&
                  comm2D,statut,ierr)
```

Es con estas funciones que logramos enviar la frontera de cada proceso a el anillo de otro y viceversa.

Lo ultimo que hacemos es liberar memoria, al igual que en la implementación secuencial.

Simulaciones numéricas: Primero se realizó una simulación secuencial de el problema, esto usando un código en C. El tiempo obtenido fue de 43.002 segundos.

Seguido de esto, se realizaron una serie de simulaciones variando el número de procesos. En total se hizo la simulación con $3, 4, 5$ y 8 hilos. Para cada caso se corrieron 5 simulaciones. Al medir el tiempo de cada una y calcular el promedio se obtuvo la siguiente tabla.

	3 **Hilos**	4 **Hilos**	5 **Hilos**	8 **Hilos**
1	15.2957106	13.9024725	11.7323284	8.16030693
2	15.6538496	14.0356884	11.8461409	8.12507820
3	15.6497660	13.9949131	11.8447075	8.13540649
4	15.4616165	13.9265671	11.9039974	8.09011078
5	15.6729774	14.0930729	11.9097795	8.15560436
Promedios	15.546784	13.99054	11.847	8.133301

Como el Speed Up se define como

$$S_p = \frac{T_1}{T_p},$$

donde p es el número de proceso, T_p es el tiempo de ejecución del algoritmo usando p hilos y T_1 es el tiempo del algoritmo secuencial. Se sigue que la tabla de Speed Up es

Número de hilos	3 **Hilos**	4 **Hilos**	5 **Hilos**	8 **Hilos**
SpeedUp	2.7659739	3.07376	3.62977	5.28734

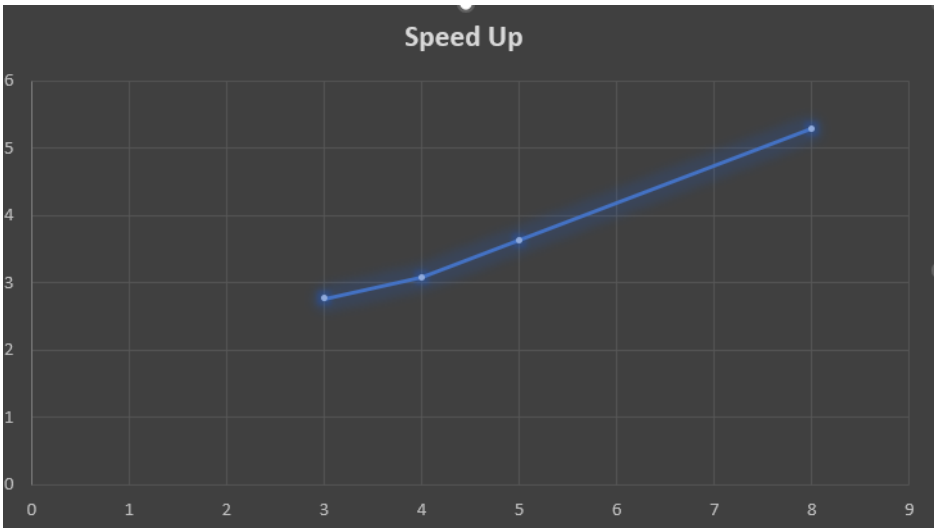
Finalmente, como la eficiencia se define como

$$E_p = \frac{S_p}{p}$$

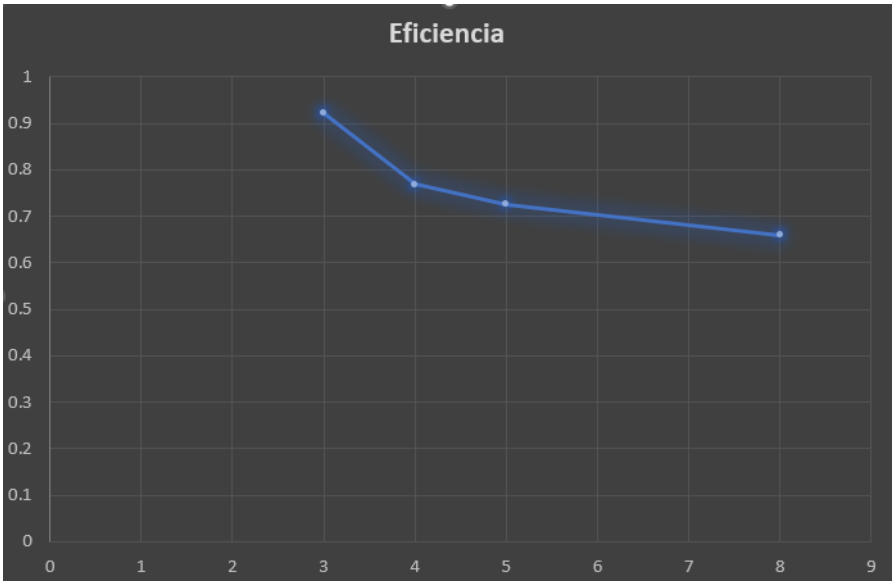
tenemos que la tabla de la eficiencia es

Número de hilos	3 **Hilos**	4 **Hilos**	5 **Hilos**	8 **Hilos**
Efficiency	0.921	0.76844	0.725954	0.66

Al graficar el Speed Up con los hilos utilizados obtenemos la siguiente gráfica,



Y para la Eficiencia se tiene la gráfica,



Conclusiones: La implementación paralela resultó ser mucho mas rápida que la secuencial. Además, vemos que si bien el aceleramiento pareció aumentar con el número de hilos, la eficiencia fue en declive, por lo que no siempre es ideal aumentar la cantidad de procesos.

Bibliografía

But what is a partial differential equation? — DE2 :https://youtu.be/ly4S0oi3Yz8

Solving the heat equation — DE3 :https://youtu.be/ToIXSwZ1pJU

Solving the Heat, Laplace and Wave equations using finite difference methods :https://personal.math.ubc.ca/~peirce/M257_316_2012_Lecture_8.pdf

12.1: The Heat Equation :https://math.libretexts.org/Bookshelves/Differential_Equations/Elementary_Differential_Equations_with_Boundary_Value_Problems_(Trench)/12%3A_Fourier_Solutions_of_Partial_Differential_Equations/12.01%3A_The_Heat_Equation

Solve a 2D Heat Equation Using Data Parallel C++ :https://www.intel.com/content/www/us/en/developer/articles/technical/solve-a-2d-heat-equation-using-data-parallel-c.html

MPI Parallelization for numerically solving the 2D Heat equation :https://dournac.org/info/parallel_heat2d#mpi-implementation

Gropp, William. Using MPI : Portable Parallel Programming with the Message-Passing Interface (third edition)