Paralelización de la ecuación de calor en 2D

Isis Mociño | Licenciatura en computación matemática

1. Ecuación de calor

Un cuerpo por el cual puede fluir el calor tiene 3 dimensiones. Su temperatura u varía respecto al tiempo t y sus coordenadas en el espacio (x, y, z).

1.1. Caso unidimensional

Para este caso consideraremos un alambre, el cual tiene una dimensión (supondremos x) considerablemente más grande que el resto. Esto nos permite considerar la temperatura u respecto a sólo una coordenada x en el espacio y el tiempo t; es decir la función es $u(x, t)$.

Figure 1: Alambre de longitud L.

Por la **Ley de la Conservación de la Energía**, se tiene que la tasa de cambio del calor almacenado en un punto de un cuerpo respecto al tiempo es igual al flujo neto de calor hacia ese punto. Este proceso está guiado por una función continua $Q = \rho c_\rho u$, la cual representa el calor en cada punto (donde ρ la densidad y c_ρ el calor específico a presión constante) y un vector V el cual representa el flujo del calor. Por lo cual, se puede escribir como

$$\frac{\partial Q}{\partial t} + \nabla V = \rho c_\rho \frac{\partial u}{\partial t} + \nabla V = 0. \tag{1}$$

De acuerdo con la **Segunda Ley de la Termodinámica**, si dos cuerpos idénticos se ponen en contacto térmico y uno es más caliente que el otro, entonces el calor debe fluir del cuerpo más caliente al más frío a una velocidad proporcional a la diferencia de temperatura. Esto quiere decir que V es proporcional al gradiente negativo de la temperatura; es decir $V = -k\nabla u$, donde k es la conductividad térmica. En una dimensión se reduce a

$$V = -k\frac{\partial u}{\partial x} e_1. \tag{2}$$

Por lo que al sustituir (2) en (1), obtenemos

$$\rho c_\rho \frac{\partial u}{\partial t} + \nabla\left(-k\frac{\partial u}{\partial x} e_1\right) = 0,$$

de donde se obtiene la **ecuación de calor unidimensional**

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \tag{3}$$

con $\alpha = \dfrac{k}{\rho c_\rho}$.

Supongamos que el alambre tiene longitud L tal que un extremo de él es en $x = 0$ y el otro en $x = L$, como se muestra en Figure 1. De igual manera, supongamos que la barra está aislada en toda su longitud de tal manera que gana y pierde calor solo por sus extremos. Esto significa que la distribución de la temperatura depende solo de la **condición inicial** y las **condiciones frontera.**

La **condición inicial** está dada por la distribución inicial de temperatura $u(x, 0)$. Las **condiciones frontera**, representan la temperatura del alambre en sus extremos; es decir $u(0, t)$ y $u(L, t)$.

1.2. Caso bidimensional

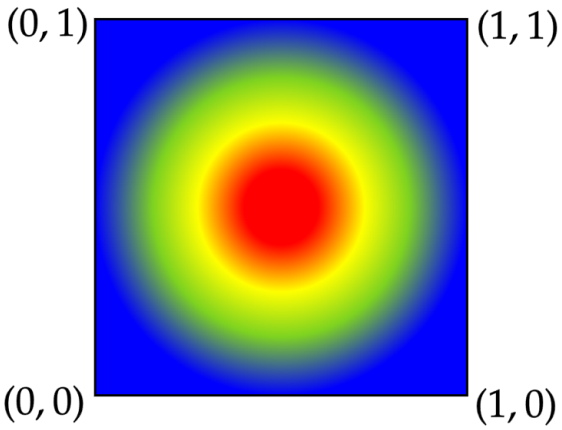


Figure 2: Placa $[0, 1] \times [0, 1]$

Para obtener la ecuación en dos dimensiones basta con considerar un placa, considerar $u(x, y, t)$ y agregar la coordenada y a la ecuación (3). De esta manera, la **ecuación de calor bidimensional** está dada como

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right), \tag{4}$$

a α también se le conoce como coeficiente difusivo.

En este proyecto se trabajará con la ecuación de calor para el caso bidimensional. El dominio Ω a usar es $[0, 1] \times [0, 1]$, como se muestra en Figure 2. Como condición de frontera se tomará la condición de Dirichlet. Además, la condición inicial será arbitraria pero suave.

1.3. Condición de Dirichlet

Se especifica en valor de la función en todos los puntos de la frontera. Si este valor es nulo, la condición es de Dirichlet *homogénea*.

2. Diferencias finitas

Las diferencias finitas son expresiones de la forma $f(x+b) - f(x+a)$. Existen distintas diferencias finitas:

- Progresivas: Hacen uso del punto actual y el posterior.
- Regresivas: Hacen uso del punto actual y el anterior.
- Centradas: Hacen uso del punto posterior y anterior.

Principalmente son usadas para aproximar derivadas de la siguiente manera:

$$f'(x) = \frac{f(x+h) - f(x)}{h}. \tag{5}$$

Mediante diferencias finitas aproximaremos $\frac{\partial u}{\partial t}$, $\frac{\partial^2 u}{\partial x^2}$ y $\frac{\partial^2 u}{\partial y^2}$. Con una diferencia finita progresiva y de acuerdo con (5), obtenemos que

$$\frac{\partial u}{\partial t} \approx \frac{u(x,y,t+\Delta t) \; - \; u(x,y,t)}{\Delta t}, \tag{6}$$

para Δt suficientemente pequeño. Además, se cumple que

$$\begin{aligned} \frac{\partial^2 u}{\partial x^2} &\approx \frac{\frac{\partial u}{\partial x}\left(x + \frac{\Delta x}{2}, y, t\right) \; - \; \frac{\partial u}{\partial x}\left(x - \frac{\Delta x}{2}, y, t\right)}{\Delta x} \\ &= \frac{\frac{u(x+\Delta x,y,t) - u(x,y,t)}{\Delta x} - \frac{u(x,y,t) - u(x-\Delta x,y,t)}{\Delta x}}{\Delta x} \\ &= \frac{u(x+\Delta x,y,t) - 2u(x,y,t) + u(x-\Delta x,y,t)}{\Delta x^2}, \end{aligned} \tag{7}$$

y de la misma manera llegamos a que

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u(x,y+\Delta y,t) - 2u(x,y,t) + u(x,y-\Delta y,t)}{\Delta y^2}. \tag{8}$$

2.1. Aplicación a la ecuación de calor

Sea $t = n$, consideremos nuestro dominio $[0,1] \times [0,1]$ discretizado, es decir como una malla y/o matriz como se muestra en Figure 3. Donde u_{ij} corresponde a el punto en la posición $x = i$ y $y = j$. Podemos reescribir (6), (7) y (8) como

$$\frac{\partial u}{\partial t} = \frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t}, \tag{9}$$

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2}, \tag{10}$$

y

$$\frac{\partial^2 u}{\partial y^2} = \frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}}{\Delta y^2}, \tag{11}$$

respectivamente.

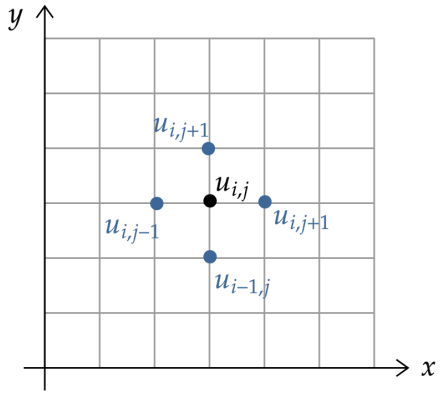


Figure 3: Puntos involucrados en las diferencias finitas.

Así, al sustituir (9), (10) y (11) en la ecuación (4) y tomando $r_x = \frac{\alpha \Delta t}{\Delta x^2}$ y $r_y = \frac{\alpha \Delta t}{\Delta y^2}$, se llega a que

$$\begin{aligned} \frac{u^{n+1}_{i,j} - u^n_{i,j}}{\Delta t} &= \alpha\left(\frac{u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}}{\Delta x^2} + \frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}}{\Delta y^2}\right) \\ \Rightarrow u^{n+1}_{i,j} - u^n_{i,j} &= r_x\left(u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}\right) + r_y\left(u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}\right) \\ \Rightarrow u^{n+1}_{i,j} &= r_x\left(u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}\right) + r_y\left(u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}\right) + u^n_{i,j}. \end{aligned} \tag{12}$$

Es decir, el punto con coordenada (i,j) en el tiempo $t = n + 1$ depende de los valores del punto y alrededor en el tiempo anterior. Esto se puede ver mejor en el esquema en Figure 4.

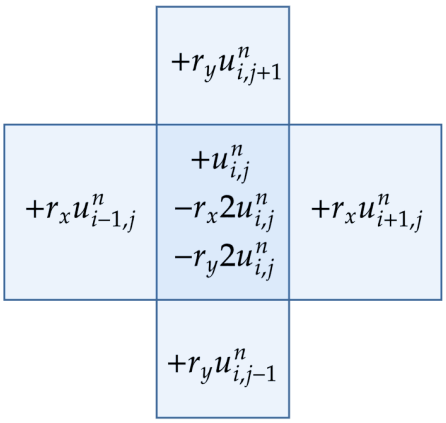


Figure 4: Suma ponderada con distribución visual.

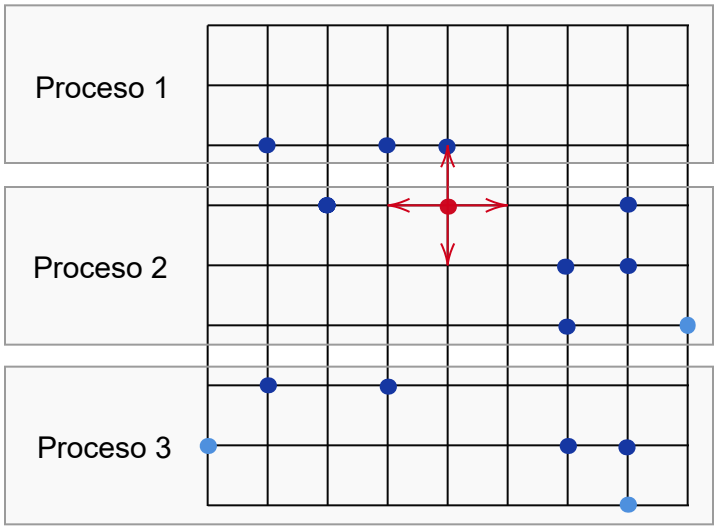
2.2. Convergencia

Este método converge si se satisface el criterio CFL; es decir, si $r_x + r_y < 1/2$.

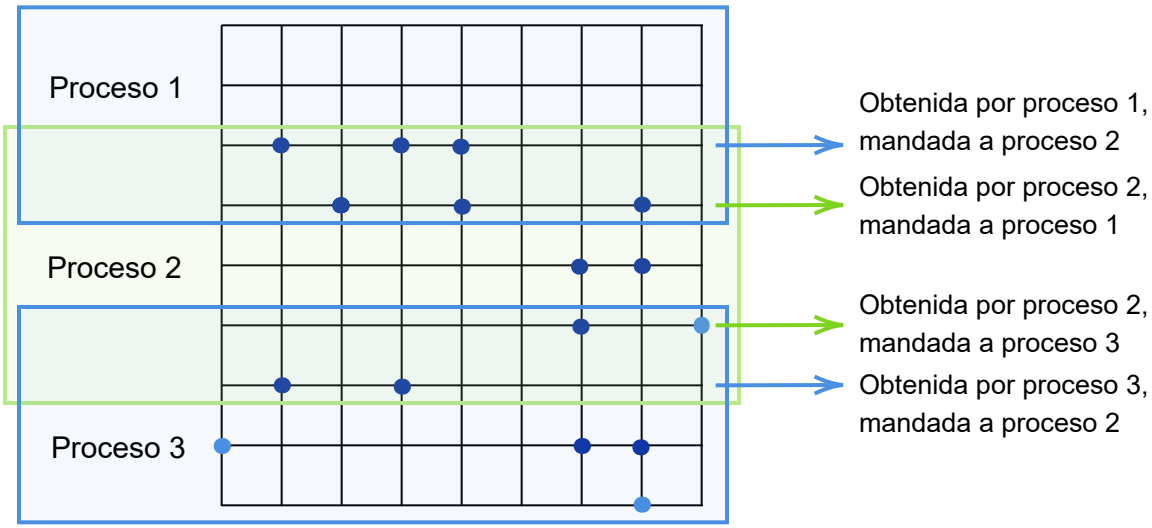
3. Estrategia de cálculo en paralelo

3.1. *Overlapping domain decomposition*

El objetivo de paralelizar es distribuir el trabajo por hacer en distintos procesos. Como contamos con una matriz, a cada proceso le mandaremos una parte de ésta, en este caso un conjunto de filas. Como ejemplo se muestra en el siguiente diagrama.



Sin embargo, al hacerlo de esta manera contamos con un problema. De acuerdo con nuestra fórmula de actualización (ecuación 12), para actualizar el punto rojo necesitamos los puntos adyacentes, entre ellos el superior que pertenece al proceso 1 al cual el proceso 2 no tiene acceso.
Esto se puede solucionar mediante la estrategia *overlapping domain decomposition*. Esto consiste en que cada proceso además inlcuirá la fila vecina de los procesos adyacentes, de esta manera se pueden actualizar todos los puntos que originalmente pertenecían a cada proceso. Un problema es que las filas en la frontera de cada proceso siguen sin poder ser actualizadas; sin embargo, éstas sí han sido actualizadas en los procesos vecinos por lo que basta comunicarlas entre procesos. La idea se visualiza de mejor manera en el siguiente esquema.



3.2. Comandos a usar en MPI

A continuación se describen los comandos principalmente usados en nuestro código con MPI.

`MPI_Init` | Inicializa la estructura de comunicación de MPI entre los procesos.
Parámetros de entrada: número de argumentos `argc` y el puntero al vector de argumentos `argv`.

```
MPI_Init(&argc, &argv);
```

`MPI_Finalize` | Finaliza la comunicación paralela entre los procesos.

```
MPI_Finalize();
```

`MPI_Comm_size` | Determina el número de procesos que están actualmente asociados al comunicador y lo almacena en una variable dada.
Parámetros de entrada: comunicador sobre el que se quiere conocer el tamaño `comm`.
Parámetros de salida: tamaño del comunicador `size`.

```
MPI_Comm_size(comm, &size);
```

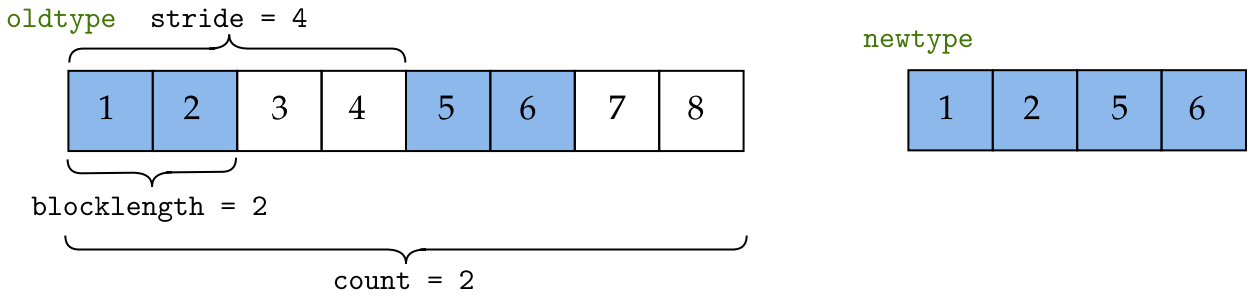
`MPI_Comm_rank` | Determina el identificador del proceso asociado con el comunicador y lo almacena en una variable dada.
Parámetros de entrada: comunicador sobre el que se quiere conocer el tamaño `comm`.
Parámetros de salida: identificador del proceso `rank`.

```
MPI_Comm_rank (comm, &rank);
```

`MPI_Type_vector` | Define un nuevo tipo de dato.
Parámetros de entrada: número de bloques `count`, número de elementos por bloque `blocklength`, número de elementos entre el inicio de cada bloque `stride`, antiguo tipo de dato `oldtype`.
Parámetros de salida: nuevo tipo de dato `newtype`.

```
MPI_Type_vector(count, blocklength, stride, oldtype, &newtype);
```

El siguiente diagrama ilustra mejor el funcionamiento.



`MPI_Type_commit` | Confirma el tipo de dato.

Parámetros de entrada: tipo de dato `datatype`.

```
MPI_Type_commit(&datatype);
```

`MPI_Type_free` | Libera el tipo de dato.

```
MPI_Type_free(&datatype);
```

`MPI_Sendrecv` | Envía y recibe un mensaje en la misma operación.
Parámetros de entrada: posición inicial del buffer de salida `sendbuf`, número de elementos a enviar `sendcount`, tipo de los datos que contiene el buffer de salida `sendtype`, rango del destino `dest`, etiqueta del mensaje de salida `sendtag`, número de elementos a recibir `recvcount`, tipo de los datos que contiene el buffer de recepción `recvtype`, rango del origen `source`, etiqueta del mensaje de recepción `recvtag`, comunicador `comm`.
Parámetros de salida: dirección inicial del buffer de entrada `recvbuf`, confirmación del estado de operación `status`.

```
MPI_Sendrecv(&sendbuf, sendcount, sendtype, dest, sendtag, &recvbuf, recvcount,
recvtype, source, recvtag, comm, &status);
```

`MPI_Allreduce` | Reduce un valor de un grupo de procesos y lo redistribuye entre todos.
Parámetros de entrada: dirección del buffer en envío `sendbuf`, número de elementos a enviar `count`, tipo de los datos a enviar `datatype`, operación de reducción `op`, comunicador `comm`.
Parámetros de salida: dirección inicial del buffer de recepción `recvbuf.`

```
MPI_Allreduce(&sendbuf, &recvbuf, count, datatype, op, comm);
```

`MPI_Cart_create` | Crea un nuevo comunicador con topología cartesiana.
Parámetros de entrada: comunicador de entrada `comm_old`, número de dimensiones `ndims`, arreglo de enteros de tamaño `ndims` que especifica el numero de procesos de cada dimensión, variable `periods` que especifica si la topología es periódica, variable `reorder` que indica si los rangos son reordenados.
Parámetros de salida: cominucador con la topología cartesiana especificada `reorder.`

```
MPI_Cart_create(Mcomm_old, ndims, &dims, &periods, reorder, &comm_cart);
```

`MPI_Cart_coords` | Determina las coordenadas de un procesos en una topología cartesiana dado su rango.
Parámetros de entrada: comunicador con topología cartesiana `comm`, rango del proceso dentro del comunicador `rank`, máximo de dimensiones llamadas `maxdims`.
Parámetros de salida: arreglo de enteros que contiene las coordenadas del proceso `coords.`

```
MPI_Cart_coords(comm, rank, maxdims, &coords);
```

`MPI_Cart_shifts` | Devuelve el rango del proceso fuente y destino para una operación de movimiento en una topología cartesiana.

Parámetros de entrada: comunicador con topología cartesiana `comm`, dimensión de la coordenada que cambia `direction`, desplazamiento `disp`.
Parámetros de salida: rango del proceso de origen `rank_source`, rango del proceso de destino `rank_dest`.

```
MPI_Cart_shift(comm, direction, disp, &rank_source, &rank_dest)
```

4. Código

Para el objetivo de este proyecto se consideran $(x, y) \in [0, 1] \times [0, 1]$ y $t \in [0, 0.5]$. Los intervalos $[0, 1]$ se discretizarán en 200 puntos y el intervalo $[0, 0.5]$ en 100,000 puntos. Además, consideraremos un coeficiente de difusión $\alpha = 1$.
En ambos códigos cuando se menciona a x nos referimos a los renglones y cuando se menciona y a las columnas. A continuación se presentan las versiones secuencial y paralela de este proceso.

4.1. Secuencial

Archivo: `SecuencialCalor.cpp`
Compilación: `g++ SecuencialCalor.cpp -o run`

A continuación se detallan las secciones principales.

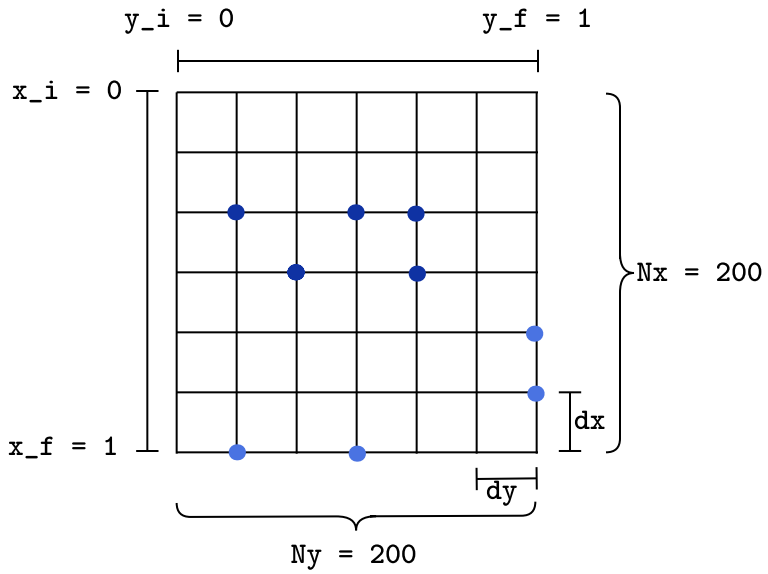
4.1.1. Declaración de variables

En esta sección se detalla qué representan las variables usadas.

- `Nx, Ny, Nt`: cantidad de puntos a considerar en los intervalos a los que pertenecen x, y y t.
- `x_i, x_f, y_i, y_f, t_i, t_f`: extremos de los intervalos a los que pertenecen x, y y t.
- `dx, dy, dt`: distancia entre los puntos de cada intervalo; es decir Δx, Δy y Δt.
- `*x, *y`: vectores de puntos en los intervalos de x y y.
- `**u`: matriz de la iteración actual.
- `**u_old`: matriz de la iteración anterior.
- `alp`: coeficiente de difusión α.
- `rx, ry`: coeficientes de actualización r_x y r_y.
- `sum`: suma de las entradas de la matriz resultante.

4.1.2. Inicialización

Se inicializan las variables de acuerdo con los valores que se mencionaron al inicio de esta sección (*Código*). Esto se presenta una representación visual. Además, se imprime `rx + ry` para verificar que se cumpla el criterio CFL.



4.1.3. Reserva y liberación de memoria

Al ejecutar ciclos, que nuestro contador comience en 1 nos facilitará las cosas. Como en C++, los arreglos se indexan en 0, si deseamos trabajar con N elementos, reservaremos el espacio para N+1, para así contar de 1 a N e ignorar el 0. En consecuencia, así se ha reservado la memoria en esta sección. Dicha memoria se libera al final del programa.

4.1.4. Puntos iniciales

Los vectores `x` y `y` se inicializan de la siguiente manera.

```
1  // Puntos en x
2  for(int i = 1; i <= Nx; i++)
3      x[i] = x_i + (i - 1) * dx;
4  // Puntos en y
5  for(int i = 1; i <= Ny; i++)
6      y[i] = y_i + (i - 1) * dy;
```

Esto para representar a los puntos que parten al intervalo $[0, 1]$.

Además, contamos con dos matrices `u` y `u_old`. Ambas se llenan con el siguiente comando.

```
for(int i = 1; i <= Nx; i++)
  for(int j = 1; j<= Ny; j++)
    u_old[i][j] = u[i][j] = sin(x[i] + y[j]) * sin(x[i] + y[j]);
```

Sin embargo, la frontera de `u` se inicializa con el valor `1.0`, como se muestra a continuación.

```
for(int i = 1; i <= Ny; i++)
  u[1][i] = u[Nx][i] = 1.0;
for(int i = 1; i <= Nx; i++)
  u[i][1] = u[i][Ny] = 1.0;
```

4.1.5. Método

El código correspondiente se muestra a continuación. En esta implementación no copiamos el contenido de la matruz `u` a la matriz `u_old`, simplemente se hace un cambio de dirección (línea 20) para ahorra tiempo y memoria. Por esto, en la iteración 1 respecto al tiempo t, después de calcular la matruz `u`, la frontera de `u_old` se iguala a `1.0` (líneas 13-18).

Notemos que para cada punto de la partición del intervalo del tiempo se calculan las entradas de la matriz `u` de acuerdo con la ecuación (12), esto corresponde a las líneas 2 a 11.

```
1   for(int k = 1; k <= Nt; k++){
2     for(int i = 2; i < Nx; i++){
3       for(int j = 2; j < Ny; j++){
4         double c = 1.0 - (2.0 * (rx + ry));
5         u[i][j] = (c * u_old[i][j])
6                 + (rx * u_old[i - 1][j])
7                 + (rx * u_old[i + 1][j])
8                 + (ry * u_old[i][j - 1])
9                 + (ry * u_old[i][j + 1]);
10      }
11    }
12
13    if(k == 1){
14      for(int i = 1; i <= Ny; i++)
15        u_old[1][i] = u_old[Nx][i] = 1.0;
16      for(int i = 1; i <= Nx; i++)
17        u_old[i][1] = u_old[i][Ny] = 1.0;
18    }
19
20    swap(u, u_old);
21  }
```

4.1.6. Verificación de resultados

En la variable `sum` se almacena la suma de las entradas de la matriz final `u`. Con los parámetros y valores descritos se debe obtener que `sum = 39999.6`.

4.2. Paralelo (MPI)

Archivo: `ParaleloCalor.cpp`
Compilación: `mpic++.openmpi ParaleloCalor.cpp -O2 -o run`
Ejecución: `sbatch slurm_calor_MPI.unknown`

A continuación se detallan las secciones del código en paralelo usando MPI, de igual manera se señalan las similitudes y diferencias con la versión secuencial.

4.2.1. Declaración de variables

Esta sección es completamente igual que en la versión secuencial.

4.2.2. Declaración de variables MPI

Se declaran variables necesarias para identificar procesos, repartir la matriz a cada proceso, crear la topología cartesiana, comunicar, así como variables de refencia a la matriz global y variables auxiliares. La descripción de cada una se encuentra a su derecha en el código.

4.2.3. Inicialización

Esta sección es casi igual que en la versión secuencial, excepto que en lugar de usar `Nx` y `Ny`, se usa `NxG` y `NyG`, pues se busca hacer referencia a variables globales y aún no comienza la región paralela. Además, será necesario liberar la memoria del nuevo tipo de dato declarado.

4.2.4. Inicialización componentes MPI

Esta es la primer diferencia notable y consta de distintas subsecciones.

- **Iniciamos la región paralela** (línea 1), además de obtener el número de procesos (línea 2) y su identificador (línea 3).

```
1 MPI_Init(&argc, &argv); // Inicio region paralela
2 MPI_Comm_size(MPI_COMM_WORLD, &ntasks); // No. de procesos
3 MPI_Comm_rank(MPI_COMM_WORLD, &taskid); // Ident. de procesos
```

- **Topología cartesiana**. Se declaran las variables para generar una topología cartesiana 2D (líneas 1-4) y se crea un comunicador comm2D con dicha topología (líneas 6-7). Dado esto, obtenemos las coordenadas cartesianas correspondientes a cada proceso (línea 9), así como sus vecinos (líneas 11-14).

```
1  dims[0] = ntasks;
2  dims[1] = 1;
3  periods[0] = periods[1] = 0;
4  reorder = 0;
5
6  MPI_Comm comm2D;
7  MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder, &comm2D);
8
```

```
9   MPI_Cart_coords(comm2D, taskid, ndims, coords);
10
11  vecino[0] = vecino[1] = MPI_PROC_NULL;
12  direction = 0;
13  displasment = 1;
14  MPI_Cart_shift(comm2D, direction, displasment, &vecino[0], &vecino[1]);
```

4.2.5. División dominio

Esta parte corresponde a repartir la matriz siguiendo la estrategia *Overlapping domain decomposition*. Como esto se hace respecto a los renglones, las columnas no se dividen (línea 16). La manera de dividir los renglones es dividir la cantidad de renglones `NxG` entre la cantidad de procesos `ntasks` obteniendo `NN` (línea 2), Dado lo anterior consideramos lo siguiente:

- Si contamos con 1 proceso, entonces se hace como la versión secuencial (líneas 4-5).
- En otro caso, si estamos en el proceso 0 y `ntasks-1`, entonces corresponden los primeros y últimos `NN+1` renglones, respectivamente (líneas 7,8,10,11).
- En otro caso, si estamos en el proceso distintos al 0 y `ntasks-1`, entonces corresponden los `NN+2` renglones (líneas 12-13).

```
1   // EJE X (renglon):
2   NN = floor(1.0 * NxG / ntasks);
3
4   if(ntasks == 1) // Caso serial
5     Nx = NxG;
6     else
7       if(taskid == 0) // Proceso 0
8         Nx = NN;
9       else
10        if(taskid == ntasks - 1) // Proceso n - 1
11          Nx = NxG - NN * taskid + 2;
12        else // Proceso 0 < i < ntasks - 1
13          Nx = NN + 2;
14
15  // EJE Y (columnas):
16  Ny = NyG;
```

4.2.6. Reserva y liberación de memoria

Se hace igual que en la versión secuencial, solamente que ahora también se consideran las variables `xG`, `yG` e `index_global`.

4.2.7. Índices

Recordemos que estamos dividiendo (repartiendo) los renglones, y éstos se enumeran de `1` a `NN` por

proceso y de `1` a `Nx` de manera global. Además, cada punto del renglón representa una coordenada en $[0,1] \times [0,1]$, para saber por proceso qué valor representa dicho puntos, es necesario hacer un mapeo de índices de renglón proceso a renglón global.

Esto se realiza de la siguiente manera, donde es necesario considerar el caso serial (1 proceso), el proceso 0 y el resto de los procesos por separado.

```
if(ntasks == 1) // Caso serial
  for(int i = 1; i <= Nx; i++)
    index_global[i] = i;
else
  if(taskid == 0) // Proceso 0
    for(int i = 1; i <= Nx; i++)
       index_global[i] = i;
  else // Proceso 0 < i <= ntasks
    for(int i = 1; i <= Nx; i++)
       index_global[i] = (taskid * NN) - 1 + (i - 1);
```

4.2.8. Definición tipo de dato

Como vamos a separar por renglones, nos definimos un nuevo de dato `tipo_row`, mediante las siguientes líneas. Lo que hace es tomar `Ny` bloques de tamaño 1, un espacio después del anterior; es decir, un renglón pues la matriz original cuenta con `Ny` columnas de nuestro interés.

```
MPI_Datatype tipo_row;
MPI_Type_vector(Ny, 1, 1, MPI_DOUBLE, &tipo_row);
MPI_Type_commit(&tipo_row);
```

4.2.9. Puntos iniciales

Los vectores xG y yG se inicializan como se hace con x y y en la versión secuencial, respectivamente. Como ahora estamos repartiendo los renglones por proceso, x se inicializa de acuerdo con el índice que le corresponde de manera global, como se muestra a continuación.

```
for(int i = 1; i <= Nx; i++)
        x[i] = xG[index_global[i]];
```

Ya que las columnas no se reparten `y = yG`.

4.2.10. Método

El método funciona casi igual que en la versión serial. Solamente se añade el proceso de comunicación, pues recordemos que los procesos comparten un renglón que no puede actualizar por sí mismo pero su vecino sí. La comunicación se realiza justo después de actualizar cada punto mediante las siguientes líneas.

```
1 if(ntasks > 1){
2   MPI_Sendrecv(&u[2][1], 1, tipo_row, vecino[0], etiqueta, &u[Nx][1], 1,
  tipo_row, vecino[1], etiqueta, comm2D, MPI_STATUS_IGNORE);
3
4   MPI_Sendrecv(&u[Nx - 1][1], 1, tipo_row, vecino[1], etiqueta, &u[1][1], 1,
  tipo_row, vecino[0], etiqueta, comm2D, MPI_STATUS_IGNORE);
5 }
```

La línea 2 indica mandar el renglón debajo del superior al renglón inferior del vecino superior. Por otro lado, la línea 4 indica mandar el renglón previo al inferior al renglón superior del vecino inferior.

4.2.11. Verificación de resultados

Finalmente cada proceso debe sumar los elementos de su matriz resultante. Sin embargo, recordemos que se comparten los renglones, lo cual puede llevar a sumar elementos de más. Esto se soluciona definiendo los rangos de suma mediante las líneas de a continuación.

```
1  if(ntasks == 1){ // Caso serial
2      it_i = 1;
3      it_f = Nx;
4  }else{
5      if(taskid == 0){ // Proceso 0
6          it_i = 1;
7          it_f = Nx - 1;
8      }else if(taskid == ntasks - 1){ // Proceso n - 1
9          it_i = 2;
10         it_f = Nx;
11     }else{ // Proceso 0 < i <= ntasks
12         it_i = 2;
13         it_f = Nx - 1;
14     }
15 }
```

Una vez que cada proceso haya realizado su suma correspondiente, se suman todas para obtener la suma completa de la matriz, esto mediante el siguiente comando.

```
MPI_Allreduce(&sum, &suma_global, 1, MPI_DOUBLE, MPI_SUM, comm2D);
sum = suma_global;
```

5. Simulaciones

Se utilizó el servidor INSURGENTE del Laboratorio de Supercómputo del Bajío, la partición usada es C1Mitad1. Más información sobre el Laboratorio se encuentra en este enlace.

Las simulaciones realizadas fueron:

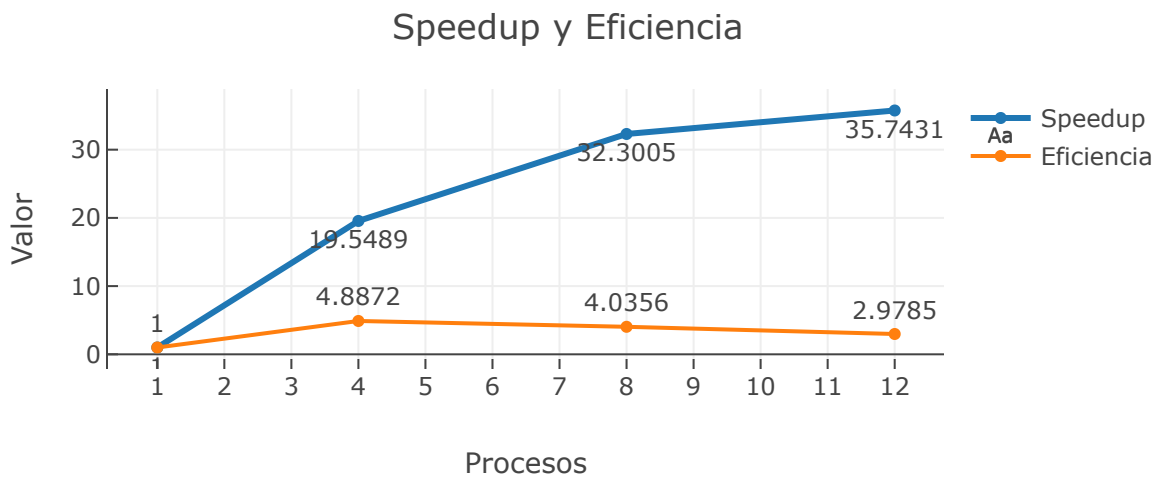
- Código secuencial, 5 veces.
- Código paralelo MPI con 4 procesos, 5 veces.
- Código paralelo MPI con 8 procesos, 5 veces.
- Código paralelo MPI con 12 procesos, 5 veces.

Los resultados se muestran en la tabla Table 1.

Versión		Tiempo promedio (s)
Secuencial		54.48504
Paralelo	*4 procesos*	2.787108
	8 procesos	1.686816
	12 procesos	1.524348

Table 1: Tiempos resultantes.

En la siguiente gráfica se muestran los resultados en términos de speedup y eficiencia.



6. Conclusión

La primera observación que tengo sobre el proyecto es la cantidad de conocimientos que se debieron unir para dar solución a un problema: física, ecuaciones diferenciales, métodos numéricos y cómputo paralelo.

La física nos permitió definir una fórmula para el problema: la *ecuación de calor*. Por otro lado, ecuaciones diferenciales y métodos numéricos brindaron una solución. Con esto el problema estaba solucionado.

Sin embargo, como se notó en la sección *Resultados*, esta solución tardaba aproximadamente 54

segundos en ser ejecutada. Esto puede representar un tiempo razonable comparado con actividades cotidianas. Sin embargo, si esta tarea se tuviera que realizar 100 veces, tardaría una hora y media. Aquí es donde el cómputo paralelo nos muestra sus beneficios, pues realizar la tarea 100 veces se puede reducir a 4.63, 2.8 o 2.5 minutos.

Para lograr esto se utilizó MPI el cual cuenta con dos ventajas principales: no hace copia de memoria y permite realizar cambios mínimos al código secuencial. Además, fue necesaria la estrategia *Overlapping domain decomposition*.

Finalemente es importante notar que un mayor número de procesos, a pesar de ser más rápido, no implica mayor eficiencia, pues la mayor eficiencia se alcanzó con 4 procesos. Así que el tiempo deseado depende del objetivo del usuario y los procesos disponibles.

7. Bibliografía

7.1. Ecuación de calor

[1] Martínez, C. V. (s/f). La ecuación del calor. Usc.es. Recuperado el 19 de mayo de 2023, de referencia 1.
[2] Connor, N. (2020, January 7). Qué es la ecuación de calor - Ecuación de conducción de calor - Definición. Thermal Engineering. Referencia 2
[3] Panda the Red. (2019, June 30). The heat equation, explained. Cantor's Paradise. Referencia 3

7.2. Diferencias finitas

[4] Rosas, C., Javier, J., Cárdenas, G., Pinilla, M. E., Damián, M. V., Moreno, S., Tovar Pérez, A., & Hugo, V. (n.d.). Solución numérica de ecuaciones diferenciales ordinarias con valores en la frontera. Unam.Mx. Recuperado el 19 de mayo de 2023, de Referencia 4.

7.3. MPI

[5] Web de Programación Paralela. (n.d.). Ugr.es. Recuperado el 19 de mayo de 2023, de Referencia 5.