Proyecto Final
Cómputo Paralelo

Josué Alexis Campos Negrón Universidad de Guanajuato
josue.campos@cimat.mx 22 de mayo del 2023

Introducción

En este proyecto trabajaremos con la ***ecuación de calor*** la cual considera que un cuerpo por el cual puede fluir el calor tiene 3 dimensiones, u es la temperatura que varía respecto al tiempo t y las coordenadas en el espacio $(x,\ y,\ z)$.

Por la ***Ley de la Conservación de la Energía*** tenemos que la tasa de cambio del calor almacenado en un cuerpo respecto su tiempo es igual al flujo neto de calor hacia ese punto. Para realizar este proceso tenemos la siguiente función continua

$$Q = \rho c_\rho u,$$

que representa el calor en un punto, donde ρ es la densidad y c_ρ el calor específico a presión constante. Considerando el vector V cuya representación es el flujo del calor tenemos que se puede reescribir de la siguiente forma

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

Por la ***Segunda Ley de Termodinámica*** sabemos que si dos cuerpos idénticos están en contacto térmico y uno tiene mayor temperatura que otro, entonces el calor fluye del cuerpo más caliente al más frio a una velocidad proporcional a la diferencia de temperatura. Con lo anterior tenemos que V es proporcional al gradiente negativo de la temperatura, si c es la conductividad térmica tenemos que

$$V \ = \ -c\nabla u. \tag{2}$$

Considerando el caso unidimensional tenemos que el cuerpo se encuentra únicamente sobre el eje x y por lo tanto la temperatura es una función con parámetrso $u(x,\ t)$ y al utilizar los resultados de (1) y (2) tenemos que la ***ecuación de calor unidimensional*** está dado por

$$\rho c_\rho \frac{\partial u}{\partial t} + \nabla\left(-c\frac{\partial u}{\partial x}e_1\right) = 0 \Longrightarrow \frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2}, \tag{3}$$

donde $k = \dfrac{c}{\rho c_\rho}$.

Si para el caso unidimensional consideramos un alambre de longitud L y suponemos que este se encuentra aislada la única fuente de calor proviene por sus extremos, entonces la distribución de la temperatura depende de una **condición inicial** y **condiciones de frontera**. La condición inicial está dada por la temperatura en el timepo cero, es decir $u(x,\ 0)$, y la condiciones de frontera por el valor de x en los extremos, es decir $u(0,\ t)$ y $u(L,\ t)$.

Para el caso bidimensional consideramos la función u con los parámetros $u(x,\ y, t)$ y de manera similar que el caso unidimensional por la ecuación de calor está dada por

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

donde k es también conocido como el coeficiente difusivo.

En nuestro caso estaremos trabajando con el dominio $\Omega = [0,1] \times [0,1]$, condición de frontera dada por la condición de Dirichlet en donde se especifica el valor de la función en todos los puntos de la frontera en donde si es nulo entonces deicmos que Dirichlet es homogénea y la condición inicial será arbitraria pero suave.

Método de diferencias finitas

Las diferencias finitas son expresiones de la forma $f(x+b) - f(x+a)$ las cuales pueden ser progresivas, regresivas y centradas. Una de las principales aplicaciones es para aproximar derivadas de la siguiente manera

$$f'(x) = \frac{f(x+h) - f(x)}{h}. \tag{4}$$

Utilizaremos diferencias finitas progresivas el cual considera el punto actual y posterior para el aproximar los valores de $\frac{\partial u}{\partial t}$, $\frac{\partial^2 u}{\partial x^2}$ y $\frac{\partial^2 u}{\partial y^2}$. Dada la expresión (4), para Δt suficientemente pequeño tenemos que

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

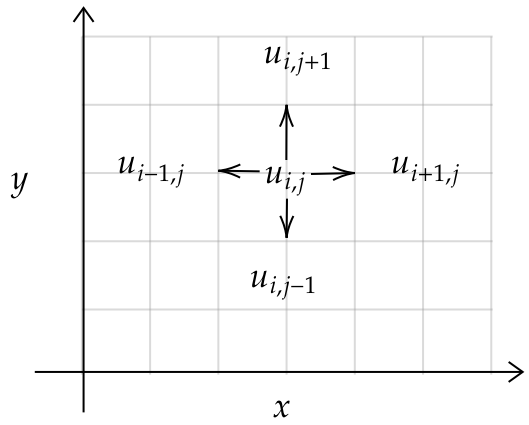
De manera similar, para $\frac{\partial^2 u}{\partial x^2}$ obtenemos que

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

utilizando el mismo procesimiento podemos concluir que $\frac{\partial^2 u}{\partial y^2}$ está dado por

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

Ahora bien, consideremos $u_{i,j}^n = u(i,\ j,\ n)$ y tomemos como dominio el intervalo $[0,\ 1] \times [0,\ 1]$ discretizado. De esta manera tenemos la siguiente representación para nuestros valores $u_{i,j}$.



Podemos reescribir nuestra ecuación de calor en terminos de $u_{i,j}$ de la siguiente manera.

$$\frac{\partial u}{\partial t} = \frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t}, \quad \frac{\partial u^2}{\partial x^2} = \frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} \quad \text{y} \quad \frac{\partial u^2}{\partial y^2} = \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^2}$$

Definamos las constantes $r_x = k\dfrac{\Delta t}{\Delta x^2}$ y $r_y = k\dfrac{\Delta t}{\Delta x^2}$, es de nuestro interés conocer el valor explícito de la expresión $u_{i,j}^{n+1}$, por lo tanto, utilizando las expresiones anteriores y sustituyendo en la ecuación de calor vemos que

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

De esta manera, observamos que el tiempo $n+1$ depende de resultados del tiempo n, entonces podemos realizar el cálculo de esta expresión de manera iterativa.

Implementación en paralelo

Para la implementación en paralelo utilizamos Open MPI, por lo tanto, primero daremos una introducción de las funciones que utilizamos para realizar la paralelización, seguido de ellos tendremos el bloque de código que se utilizó en nuestra implementación y finalmente explicaremos la lógica de la paralelización.

La función `MPI_Init` inicializa la comunicación de MPI entre los procesos. Tiene como parámetros:

- `argc:` Puntero al número de argumentos.
- `argv:` Vector al número de argumentos

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

La función `MPI_Comm_size` almacena en una variable dada el número de procesos asociados al comunicador. Tiene como parámetros:

- `comm:` Comunicador sobre el que se quiere conocer el tamaño.

- `size:` Tamaño del comunicador.

```
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
```

La función `MPI_Comm_rank` almacena en una variable dada el identificador del proceso en el comunicador seleccionado. Tiene como parámetros:

- `comm:` Comunicador sobre el que se quiere conocer el tamaño.
- `rank:` Identificador del proceso.

```
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
```

La función `MPI_Cart_create` crea un comunicador con topología cartesiana. Tiene como parámetros:

- `comm_old:` Comunicador de entrada.
- `ndims:` Número de dimensiones en la rejilla cartesiana.
- `dims:` Vector de tamaño `ndims` en donde cada entrada representa el número de procesos por dimensión.
- `periods:` Vector de tamaño `ndims` que indica por entrada si la dimensión es periodica (true) o no (false).
- `reorder:` Si la clasificación puede reordenarse.
- `comm:` Variable en donde se almacenará el comunicador.

```
MPI_Cart_create(MPI_COMM_WORLD,
                ndims,
                dims_vec,
                periodicite,
                reorganisation,
                comm2D)
```

La función `MPI_Cart_coords` determina las coordenadas de un proceso en la topología cartesiana dado un identificador de proceso. Tiene como parámetros:

- `comm:` Comunicador con topológica cartesiana.
- `rank:` Identificador de proceso.
- `maxdims:` Tamaño del vector coord.
- `coord:` Vector de tamaño `ndims` donde se almacenarán las coordenadas correspondientes al proceso.

```
MPI_Cart_coords(comm2D,
                taskid,
                ndims,
                coords)
```

La función `MPI_Cart_shift` devuelve el identificador del proceso fuente y destino de una operación de

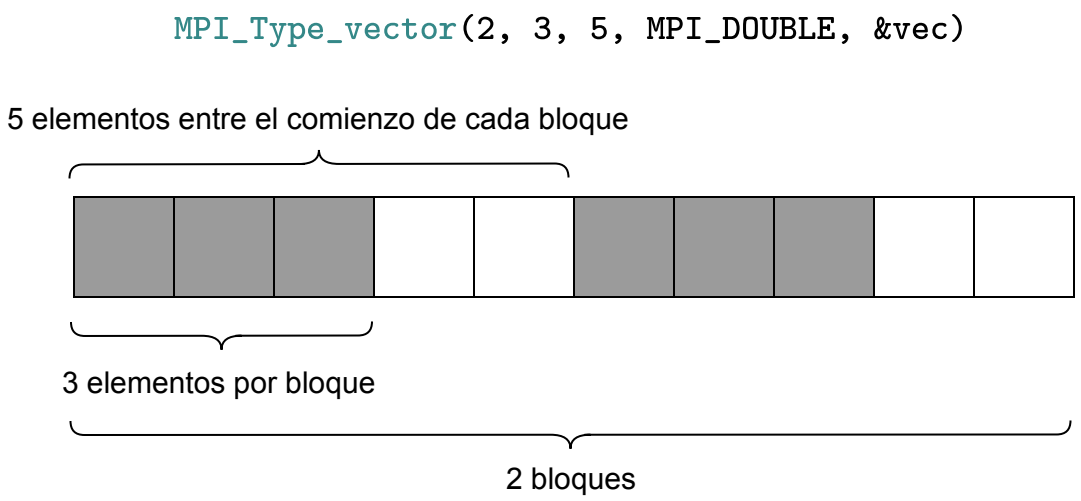
movimiento en una topología cartesiana teniendo en cuenta la dirección y cantidad. Tiene como parámetros:

- `comm:` Comunicador con topológica cartesiana.
- `direction:` Dimesión sobre el cual se realizará el movimiento.
- `disp:` Cantidad de desplazamiento.
- `vecino[0]:` Proceso fuente en la dimensión `direction` y a distancia `displasment`.
- `vecino[1]:` Proceso destino en la dimensión `direction` y a distancia `displasment`.

```
MPI_Cart_shift(comm2D,
               direction,
               displasment,
               &vecino[0],
               &vecino[1]);
```

La función `MPI_Type_vector` crea un tipo de dato que consta de un número especificado de un tamaño especificado. Tiene como parámetros:

- `count:` Número de bloques.
- `blocklength:` Número de elementos de cada bloque.
- `stride:` Número de elementos entre el comienzo de cada bloque.
- `oldtype:` Tipo de dato de cada elemento.
- `typename:` Variable en donde se almacenará el nuevo `Datatype`.




```
MPI_Type_vector(Ny, 1, 1, MPI_DOUBLE_PRECISION, &type_row)
```

La función `MPI_Type_commit` confirma el tipo de dato. Este solo consta de un parámetro `typename` que es la variable que almacena el nuevo tipo de dato.

```
MPI_Type_commit(&tipo_col);
```

La función `MPI_Sendrecv` envia y recibe un mensaje en la misma operación. Tiene como parámetros:

- `sendbuf:` Dirección del buffer de salida.
- `sendcount:` Número de elementos dfel tipo de dato especificado a enviar.
- `sendtype:` Tipo de dato del buffer de salida.
- `dest:` Identificador del destino.
- `sendtag:` Etiqueta del mensaje de salida.
- `recvbuf:` Dirección del buffer de entrada.
- `recvcount:` Máximo número de elementos de entrada.
- `recvtype:` Tipo de dato del bufer de entrada.
- `source:` Identificador del origen.
- `recvtag:` Etiqueta del mensaje de entrada.
- `comm:` Comunicador en el cual se realizará la operación de envío y recibimiento.
- `status:` Variable en donde se almacena el estatus de la operación.

```
MPI_Sendrecv(&u[2][1], 1, tipo_col, vecino[0], etiqueta,
             &u[Nx][1], 1, tipo_col, vecino[1], etiqueta,
             comm2D, MPI_STATUS_IGNORE);
MPI_Sendrecv(&u[Nx - 1][1], 1, tipo_col, vecino[1], etiqueta,
             &u[1][1], 1, tipo_col, vecino[0], etiqueta,
             comm2D, MPI_STATUS_IGNORE);
```

La función `MPI_Allreduce()` reduce un valor de un grupo de procesos a un mismo valor. Tiene como parámetros:

- `sendbuf:` Dirección del buffer de en envío.
- `recvbuf:` Dirección que recibirá el resultado de la operación de reducción.
- `count:` Número de elementos que se va enviar.
- `datatype:` Tipo de dato de cada elemento.
- `op:` Operación de reducción.
- `comm:` Comunicador en el que se realizará la operación de reducción.

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

La función `MPI_Type_free` ibera el tipo de dato creato. Unicamente recibe la variable que contiene el `datatype` creado.

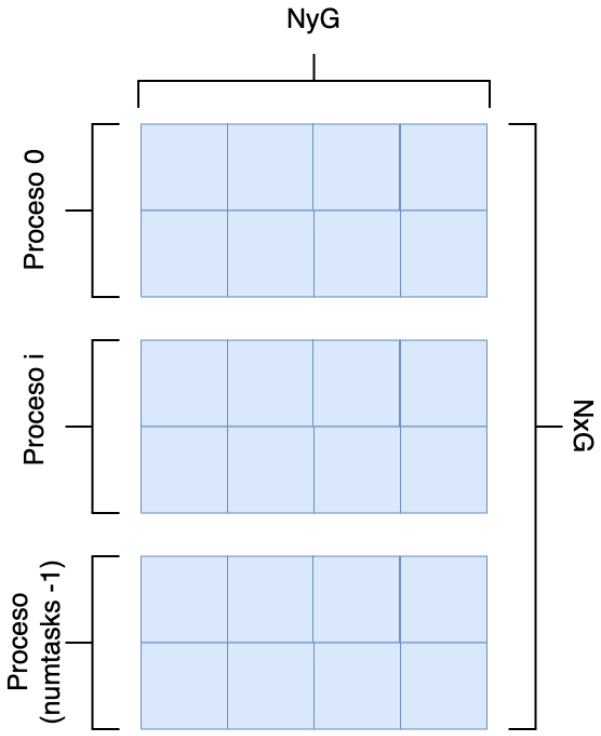
```
MPI_Type_free(&tipo_col);
```

La función `MPI_Finalize` finaliza la comunicación paralela entre los procesos.

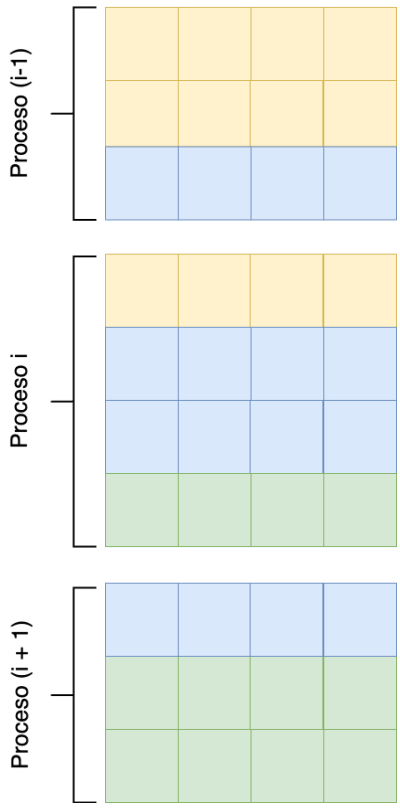
```
MPI_Finalize();
```

Las funciones anteriores son las funciones utilizadas para la paralelización en Open MPI. A continuación presentaremos la lógica de paralelización utilizada.

El método utilizado tiene como nombre *Overlaping Domain Descomposition* en el cual consideramos una cruadrícula indexada en uno, en nuestro caso dicha cuadrícula está dada de tamaño $NxG \times NyG$. Luego, es de nuestro interés paralelizar sobre el eje x, por lo tanto consideramos el número de procesos dado por la variable $numtasks$ y dividir el intervalo $[1, NxG]$ en $numtasks$ partes, entonces tenemos una división de la siguiente forma:

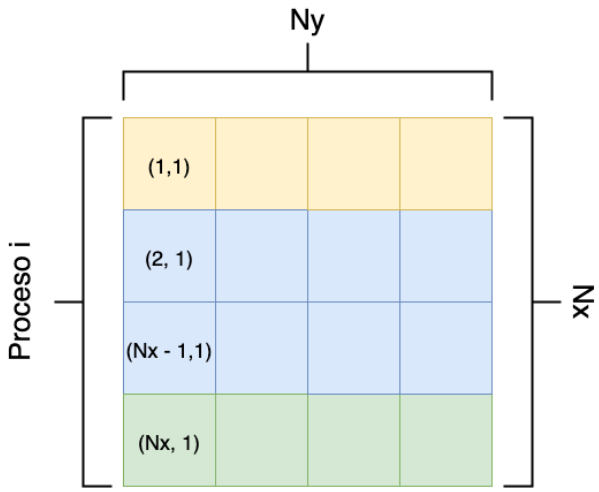


Como comentamos en la teoría del método de *diferencias finitas* para actualizar una casilla es necesario conocer las casillas vecinas en las cuadro distintas direcciones *norte, este, sur* y *oeste*, por lo tanto, las casillas que son frontera en la división de la cuadrícula dependen de la actualización realizada por procesos vecinos a este. La solución a este problema consiste en añadir a cada bloque de cada proceso dos filas más, uno en la parte superior y otro en la posterior, los cuales representarán las filas compartidas con los procesos vecinos necesarias para el cálculo de los puntos interiores del bloque correspondiente. El siguiente dibujo ejemplifica el añadido de filas para un proceso dado.



De esta manera nos aseguramos que cada proceso contenga las filas necesarias para el cálculo de los puntos interiores, Después de una iteración es necesario realizar la comunicación entre los procesos para enviar y recibir las actualizaciones de las filas compartidas. Por ello creamos un nuevo tipo de dato llamada `type_row` que representa una fila con `Ny` elementos utilizando la función `MPI_Type_vector` considerando `Ny` bloques de tamaño 1 y con 1 elemento de separación entre el comiento de cada bloque.

Luego, consideramos que cada proceso tiene un subcuadrícula de tamaño $Nx \times Ny$, queremos conocer cual es la dirección de los elementos que serán enviados a los procesos vecinos y cuales son las direcciones donde recibiremos información de elementos vecinos.



Como observamos en la imagen anterior, las direcciones de envío de información están dadas por las coordenadas $(2,\ 1)$ y $(Nx\ -\ 1,\ 1)$ mientras que las direcciones de recibimiento de información están dadas por las coordenadas $(1,\ 1)$ y $(Nx,\ 1)$. Está acción la podemos realizar con la función `MPI_Sendrecv`

a y con apoyo del `Datatype` de `type_row` (en la descripción de la función `MPI_Sendrecv` se especifica los parámetros utilizados para nuestra implementación).

Comparación en serie y paralelo

Primeramente comentaremos los componentes principales de la implementación en **serie.** En el código `HeatEquation_seq.cpp` se puede observar la implementación en serie, inicialmente tenemos la declaración, inicialización y reserva de memoria necesaria para el método de diferencias finitas. Seguido de ello realizamos el cálculo de los puntos x y y en el intervalo $[0, 1]$ de la siguiente manera considerando Nx y Ny el número de divisiones de la cruadrícula, dx y dy la separación entre cada elemento y x_ini, y_ini en inicio del dominio.

```
// Calculo de puntos de x
for(int i = 1; i <= Nx; i++)
  x[i] = x_ini + (i - 1) * dx;

// Calculo de puntos de y
for(int j = 1; j <= Ny; j++)
  y[j] = y_ini + (j - 1) * dy;
```

Seguido de ello procedimos a realizar el cálculo de la cruadrícula inicial representado por la variable v que consta de todos sus elementos con valor $\sin(x + y)^2$ y la frontera con el valor 1. El siguiente bloque de código presenta la implementación de la inicialización.

```
// Condiciones iniciales interior
for(int i = 1; i <= Nx; i++)
  for(int j = 1; j<= Ny; j++)
    v[i][j] = u[i][j] = sin(x[i] + y[j]) * sin(x[i] + y[j]);

// Condiciones de frontera
for(int j = 1; j <= Ny; j++)
  u[1][j] = u[Nx][j] = 1.0;

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

Luego, procedemos a realizar Nt iteraciones actualizando la matriz u con la suma ponderada dada por el método de diferencias finitas. Sepuede observar que como trabajamos con memoria dinámica basca con realizar un `swap` entre las matrices para actualizar la iteración anterior y proceder al cálculo de la nueva iteración. También, dada la matriz inicial contiene un valor de frontera distinto a la condición de frontera global, es necesario realizar la modificación de frontera en la primera iteración, por último, como realizamos la actualización de la matriz al final de cada iteración es necesario realizar un último `swap` al finalizar todas

las iteraciones para que quede almacenada en la matriz u nuestra respuesta al problema. A continuación la implementación.

```
for(int k = 1; k <= Nt; k++){
  for(int i = 2; i <= (Nx - 1); i++){
      for(int j = 2; j <= (Ny - 1); j++){
          double c = 1.0 - (2.0 * (rx + ry));
          u[i][j] = (c * v[i][j])
                  + (rx * v[i - 1][j])
                  + (rx * v[i + 1][j])
                  + (ry * v[i][j - 1])
                  + (ry * v[i][j + 1]);
      }
  }
    // Actualizacion
    if(k == 1){
        for(int j = 1; j <= Ny; j++)
            v[1][j] = v[Nx][j] = 1.0;

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

Por último para comprobar el resultado obtenido realizamos la suma por elementos de la matriz resultante.

Ahora bien, procedemos a comentar las componentes principales de la implementación en paralelo. En la sección **Implementación en paralelo** se especifica los parámetros utilizamos para las funciones de Open MPI, por lo tanto únicamente comentaremos acerca de las funciones utilizadas.

En el programa HeatEquation_par.cpp se puede observar la implementación en paralelo y de manera similar que en la implementación en serie, primeramente iniciamos con la declaración e inicialización de variables, luego utilizandos la funciones `MPI_Init`, `MPI_Comm_size`, y `MPI_Comm_rank` iniciamos la sección en paralelo, obtenemos el número de procesos y el identificador del proceso. Seguido de ello creamos una topología cartesiana, obtenemos las coordenadas de cada proceso e identificamos los vecinos adyacentes sobre el eje x con las funciones `MPI_Cart_create`, `MPI_Cart_coords`, y `MPI_Cart_shift`.

Posteriormente, procedemos a obtener el intervalo correspondiente al proceso, para ello recordemos que cada proceso le corresponde una subcruadrícula de la cuadrícula original y le añadimos dos filas compartidas con otros procesos. Almacenamos dicho intervalo indexado en uno en las variables Nx y Ny, la implementación es la siguiente.

```
NN = floor(1.0 * NxG / numtasks);

if(numtasks == 1)                              // Caso serial
    Nx = NxG;
else
    if(taskid == 0)                            // Proceso 1
        Nx = NN;
    else
        if(taskid == numtasks - 1)             // Proceso n - 1
            Nx = NxG - NN * taskid + 2;
        else                                   // Proceso
     Nx = NN + 2;                        // 0<i<numtasks-1
// Division en eje Y
    Ny = NyG;
```

Luego de ello, realizamos el cálculo de un arreglo con nombre `index_global` es cual es una correspondencia de los índices del intervalo $[1,\ Nx]$ a los índices correspondientes al proceso en el intervalo $[1,\ NxG]$. El cálculo se realiza de la siguiente manera.

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

Con lo anterior, tenemos que la *modificación a la implentación del código serial* en el cálculo de los vectores x y y depende primeramenta del cálculo de los vectores xG y yG los cuales son los valores en el intervalo global, luego utilizando el vector `index_global` para obtener nuestros vectores correspondiente x y y al proceso.

```
// Calculo de puntos globales x
for(int i = 1; i <= NxG; i++)
  xG[i] = x_ini + (i - 1) * dx;

```

```
// Calculo de puntos globales y
for(int j = 1; j <= NyG; j++)
    yG[j] = y_ini + (j - 1) * dy;

// Calculo de puntos de x
for(int i = 1; i <= Nx; i++)
  x[i] = xG[index_global[i]];

// Calculo de puntos de y
for(int j = 1; j <= Ny; j++)
  y[j] = yG[j];
```

Posteriormente tenemos la inicialización del vector v inicial y la frontera de nuestra cuadrícula, lo anterior junto con el cálculo de la suma ponderada de la ecuación de calor es lo mismo a la implementación en serie, la única diferencia es al momento de realizar la comunicación entre procesos en donde primeramente definidos la variable `type_row` utilizando la función `MPI_Type_vector`, luego con la función `MPI_Sendrecv` realizamos la comunicación entre procesos de las filas compartidad.

Finalmente, es necesario realizar un ajuste en los intervalos de la matriz u resultante para la corroborar el correcto funcionamiento y no realizar la suma doble de una fila considerando aquellas que son compartidas, para ello tenemos las siguientes condiciones.

```
if(numtasks == 1){
  it_ini = 1;
  it_fin = Nx;
}else{
  if(taskid == 0){
    it_ini = 1;
    it_fin = Nx - 1;
  }else if(taskid == numtasks - 1){
    it_ini = 2;
    it_fin = Nx;
  }else{
    it_ini = 2;
    it_fin = Nx - 1;
  }
}
```

Con esto concluimos las componentes principales de la implementación en paralelo y las modificaciones al código serial.

Resultados

Primeramente corriendo el programa serial en cinco ocasiones obtenemos en promedio que realiza el método de *diferencias finitas* en un tiempo de 54.9130 **segundos**. Ahora bien, realizaremos el mismo análisis para nuestra implementación en paralelo utilizando 4, 8 y 12 procesadores. Compararemos los resultados utilizando el *speedup* y la *eficiencia* respecto al tiempo en serie.

Numero de procesadores	4 procesadores	8 procesadores	12 procesadores
Tiempo en segundos	2.7769	1.6720	1.5429
SpeedUp	19.7749	32.8427	35.5907
Eficiencia	4.9437	4.1053	2.9658

Conclusión

Notamos que la *ecuación de calor* junto el *método de diferencias finitas* tienen propiedades que encajan para realizar un cálculo computacional lo que permite un desarrollo iterativo secuencial. Luego, gracias al *método Overlaping Domain Descomposition* y las funciones de *Open MPI* podemos realizar una versión en paralelo sin realizar significantes modificaciones a la implementación serial y como se observa en los resultados tenemos una gran diferencia en tiempo respecto a la versión serial con las versiones en paralelo. En particular vemos que con 4 procesadores tenemos mejor eficiencia pero el tiempo *speedup* es mejor en el caso de 12 procesadores, siendo el caso de 8 procesadores aquel que tiene mejor relación en tiempo y eficiencia.

Bibliografía

- **Teoría ecuación de calor:** Universidad de Santiago de Compostela Thermal Engineering Cantorsparadise UNAM
- **MPI_Init:** Universidad de Granada
- **MPI_Comm_size:** Universidad de Granada
- **MPI_Comm_rank:** Universidad de Granada
- **MPI_Cart_create:** Universidad de Granada Hot Examples
- **MPI_Cart_coords:** MPIch
- **MPI_Cart_shift:** MPIch BuTech open-mpi.org Stackoverflow Rookiehpc Universidad de Granada
- **MPI_Type_vector:c** Hot Examples Microsoft open-mpi-org Rookiehpc Universidad de Granada
- **MPI_Type_commint:** open-mpi.org Microsoft MPIch Universidad de Granada
- **MPI_Sendrecv:** Universidad de Granada MPIch open-mpi.org Microsoft Rookiehpc
- **MPI_Allreduce:** Universidad de Granada MPIch Microsoft Rookiehpc