TAREA 5 - Programación Paralela - CC332

Alumno : Walter Jesús Felipe Tolentino **Código :** 20172714F

1. Compilar y ejecutar el código secuencial de vibración de onda en una dimensión.

Filename : onda_sec.cpp
Compiling and running : g++ onda_sec.cpp -o sec && ./sec
Salida : Los puntos de la foto final de la onda, i.e en el **step = 1000.**

Parámetros de ejecución y condiciones iniciales de la ecuación de la onda :

```
void init_param(void){
   tpoints = 800;
   nsteps = 1000;
}

void init_line(void)
{
   double x, k=0.002, tmp=tpoints - 1;
   for (int j = 1; j <= tpoints; j++){
         x = k / tmp;
         values[j] = sin(2.0 * PI * x);
         k = k + 1.0;
   }
}
```

Gráfico de la onda de salida :

```
void print_foto_final_de_la_onda(void){
   double x, k=0.002, tmp = tpoints - 1;
   for (int j = 1; j <= tpoints; j++){
      x = k / tmp;
      printf("(%.7f,\t %.7f)\n", 2.0 * PI * x , values[j]);
      k = k + 1.0;
   }
}
```

***Nota** : El método **update**() permite actualizar los valores de la onda en cada step.*

2. Paralelizar el código

Filename : onda_mpi.cpp
Compiling and running : mpic++ onda_mpi.cpp -o onda_mpi && mpirun --oversubscribe -np [size] ./onda_mpi

Nota: Uso la bandera **--oversubscribe**, puesto que estoy usando Open MPI v.4.0.3, en esta versión mpi trabaja con los núcleos por defecto de procesador, en mi caso solo 2, así que se coloca esa bandera para que cree más procesos (virtuales). Me parece que de trabajar con un versión inferior no es necesario,

Salida : Los puntos de la foto final de la onda, i.e en el step=1000, por número de procesos (size).

Parámetros de ejecución y condiciones iniciales de la ecuación de la onda :

En este caso vamos a tener un vector **master_root_rank_values[.]** que va almacenar toda la data que le envíen todos los procesos, puede ser por cada step para graficar step por step (animación) o en el step = 1000 (final).

Y cada proceso tendrá sus propios vectores que van a manipular cierta región del dominio (particionado), **local_values[.], local_old_values[.], local_new_values[.].**

Las siguientes funciones se van a ejecutar para todos los procesos :
Lo que está sombreado es lo que se diferencia del código secuencial

```
void init_param(int size){
    tpoints = 800;
    nsteps = 1000;
    nlocal = tpoints/size;
}

void init_line(int rank, int size){
    double x, k = 0.002 + rank * nlocal, tmp = size * nlocal - 1;
    for (int i = 1; i <= nlocal; i++){
        x = k / tmp;
        local_values[i] = sin(2.0 * PI * x);
        k = k + 1.0;
    }
    for (int i = 1 ; i <= nlocal; i++) local_old_values[i] = local_values[i];
}

void do_math(int i){
    double dtime= 0.3, c= 1.0, dx= 1.0, tau, sqtau;
    tau = (c * dtime / dx);
    sqtau = tau * tau;
    local_new_values[i] = (2.0 * local_values[i]) - local_old_values[i] +
(sqtau * (local_values[i - 1] - (2.0 * local_values[i]) + local_values[i + 1]));
}
```

Falta el método ***update()*** es donde se van a realizar todas las comunicaciones entre los procesos.

Dentro del método ***update()*** lo nuevo es :

```
 if(rank == 0 ){
      MPI_Send(&local_values[nlocal], 1, MPI_DOUBLE, right, LtoR, MPI_COMM_WORLD);
      MPI_Recv(&local_values[nlocal + 1], 1, MPI_DOUBLE, right, RtoL , MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   }
 else if(rank == size - 1 ){
      MPI_Send(&local_values[1], 1, MPI_DOUBLE, left, RtoL, MPI_COMM_WORLD);
      MPI_Recv(&local_values[0], 1, MPI_DOUBLE, left, LtoR , MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   }
else{
      MPI_Send(&local_values[1], 1, MPI_DOUBLE, left, RtoL, MPI_COMM_WORLD);
      MPI_Send(&local_values[nlocal], 1, MPI_DOUBLE, right, LtoR, MPI_COMM_WORLD);
      MPI_Recv(&local_values[0], 1, MPI_DOUBLE, left, LtoR , MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Recv(&local_values[nlocal+1], 1, MPI_DOUBLE, right, RtoL , MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   }
```

Nota : *Estas condiciones nos permite enviar* **MPI_Send** *y recibir* **MPI_Recv** *los valores frontera a los vecinos, teniendo en cuenta que si estamos en los <<procesos extremos>> solo enviamos y recibimos un valor.*

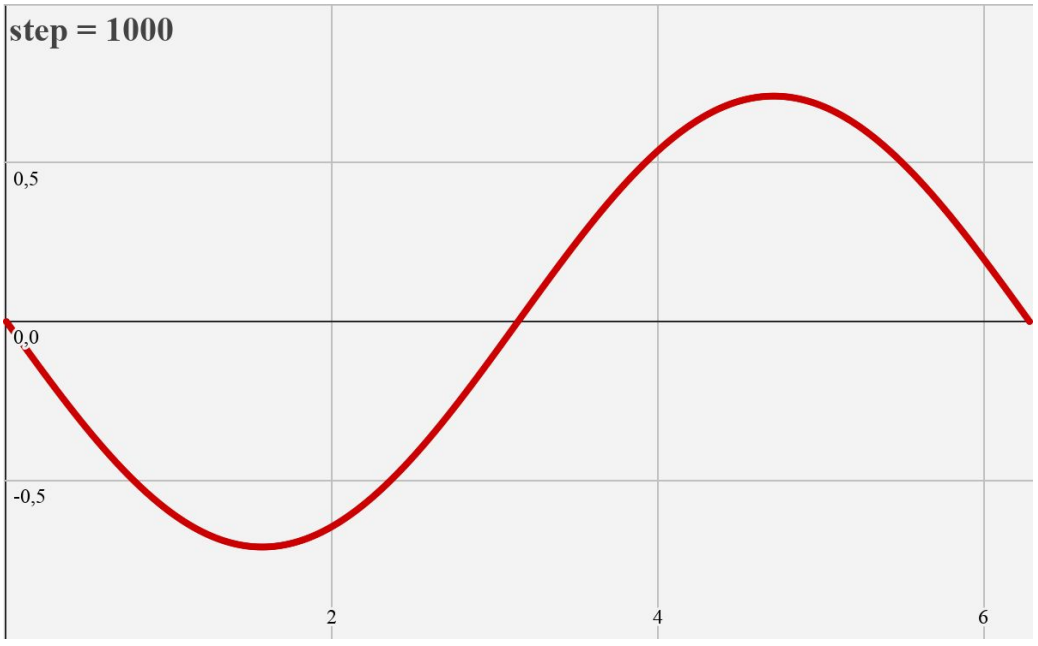
Luego para enviar la data al proceso maestro usamos **MPI_Gather** .

```
MPI_Gather(&local_values[1], nlocal, MPI_DOUBLE, &master_root_rank_values[1], nlocal, MPI_DOUBLE, 0,
                MPI_COMM_WORLD);
```

Y en el método principal ***main***() le pasamos al método update left y right para que conozca a sus vecinos.

```
init_param(size);
init_line(rank, size);
right = rank + 1; left = rank - 1;
update(left, right, rank, size);
if(rank == 0) print_foto_final_de_la_onda();
```

Finalmente obtenemos la siguiente gráfica dado ***print_foto_final_de_la_onda*()**.



3. Medir tiempos de ejecución para np=2,4,8 utilizando los mismos parámetros de simulación.

```
start = MPI_Wtime();
        init_param(size);
        init_line(rank, size);
        right = rank + 1; left = rank - 1;
        update(left, right, rank, size);
end = MPI_Wtime();
```

Tenemos el siguiente cuadro :

N° de procesos	tiempo comunicación + tiempo de cálculo
2	0.0098533170
4	0.0260977500
8	0.0573591190
16	0.1451014070
32	0.3969797210

Se observa que a medida que los procesos se incrementa el tiempo ya no se incrementa mucho, el incremento pasó de ser del triple (de p=2 a p=4) a prácticamente el doble (de p = 8 a p=16). En p = 32 los tiempos de comunicación consumen casi todo.

4. Animación de la onda

En el método **update()** agregar el código de color rojo.

```
for ( int  i  =  1 ;  i  <=  nsteps ; i++){
.
.
.
        if(rank == 0){
                printf("Foto de la onda en el paso : %d\n", i);
                print_foto_final_de_la_onda();
         }

}
```

Esto mostrará los puntos de la foto de la onda por cada step, que nos va permitir graficar, con OpenGL, por ejemplo.