Memoria Practica 1
ACAP – Calcular Pi

Grado: Ing. Informática.
Grupo: 3º IC
Profesor: Nicolás Calvo Cruz
Alumno: Carlos López Martínez

Documento de Practicas

1 Medicion de tiempos secuencial y paralelamente.

1.1 Medir tiempo

Para la medición de tiempo se ha usado get_wall_time(). Esta función se puede ver en cualquiera de los archivos de código proporcionados.

```
double get_wall_time()
{
    struct timeval time;
    if (gettimeofday(&time, NULL))
    {
        printf("Error de medición de tiempo\n");
        return 0;
    }
    return (double)time.tv_sec + (double)time.tv_usec *
0.000001;
}

time = get_wall_time();
double result = foo();
time = get_wall_time() - time;
```

De esta forma se puede calcular el tiempo que tarda en ejecutarse la función foo();

1.2 Codigo Secuencial

El código para la ejecución de forma secuencial esta como archivo llamado pi_sec.c.

```
for (int i = 0; i < TRYTIMES; i++)
{
        double timeLeiblocal = 0.0;
        double timeRectlocal = 0.0;

        timeLeiblocal = get_wall_time();
        double pipart = piLeibniz(steps);
        timeLeiblocal = get_wall_time() -
timeLeiblocal;
        timeLeibMean += timeLeiblocal;
}
```

Para medir el tiempo y hacer la media lo que hemos hecho es hacer la media de lo que tardamos en ejecutar el código 10 veces. Haciendo lo mismo para la ejecución de Rectangles.

1.3 Codigo Paralelo

El código para la ejecución de forma paralela esta como archivo llamado pi.c. Los métodos se han diseñado de la siguiente forma.

- Leibniz:

```
double piLeibnizParallel(int steps, int rank, int size)
{
    int interval = steps / size;
    int start = rank * interval;
    int end = (rank + 1) * interval;
    double pipart = 0.0;
    double num = 1.0;
    double denom = 1.0;
    for (int i = start; i < end; i++)
    {
        pipart += num / denom;
        num = -1.0 * num;
        denom += 2.0;
    }
    return 4.0 * pipart;
}
```

Le añadimos a la cabecera de la función dos nuevos parametros, estos van a ser para identificar que parte del calculo van a calcular. Estos los vamos a obtener con el Rank que ocupan de procesos y cuantos de ellos hay en total.

La forma en la que se separa el trabajo lo podemos ver como una recta donde cada uno tiene que trabajar con un rango de valores. El Intervalo que le corresponde a cada viene dado por la cantidad de interacciones que se van a hacer en total y se va a separar entre N procesos (size en nuestro caso). Donde va a comenzar a calcular cada proceso y donde va a acabar este viene dado por start y end donde empieza en el intervalo por el rango que ocupa y acaba donde empieza el siguiente proceso. El resto de la función es igual que la normal.

- Rectangles:

```
double piRectanglesParallel(int intervals, int rank,
int size)
{
    int interval = intervals / size;
    int start = rank * interval;
    int end = (rank + 1) * interval;
    double width = 1.0 / intervals;
    double sum = 0.0, x;
    for (int i = start; i < end; i++)
    {
        x = (i + 0.5) * width;
        sum += 4.0 / (1.0 + x * x);
    }
    return sum * width;
}
```

Como el calculo en esta función originalmente es indistingiblemente de Leibniz salvo por los cálculos internos que se realizan, el proceder es el mismo, dividimos la carga en intervalos y asignamos estos intervalos según el puesto que ocupe el proceso que la ejecuta.

1.4 Obtencion de Datos y Grafica

Para la medición del tiempo se ha hecho un Script que va ejecutando el programa con diferentes valores para las iteraciones. Va desde 0 hasta 10^9 (~ Max Número Int) . Este va aumentando de 10 millones es 10 millones consiguiendo un total de 100 datos, estos también se encuentran como archivos, para los datos registrados de forma paralela tenemos “pi_parallel.dat” y para los secuenciales tenemos “pi_secuencial.dat”

Se han hecho dos scripts, uno para la maquina local y otro para atcGrid, ya que en atc no se permite trabajos de más de 1 min. Se ha hecho de forma logarítmica el escalado de las iteracciones.
Los Script se llaman “comexec.sh” para la maquina local y “comexecatc.sh” para AtcGrid. Ambos tiene varios parametros.

```
/ScriptName.sh NombreArchivo(sin .c) NumProcesos.
```

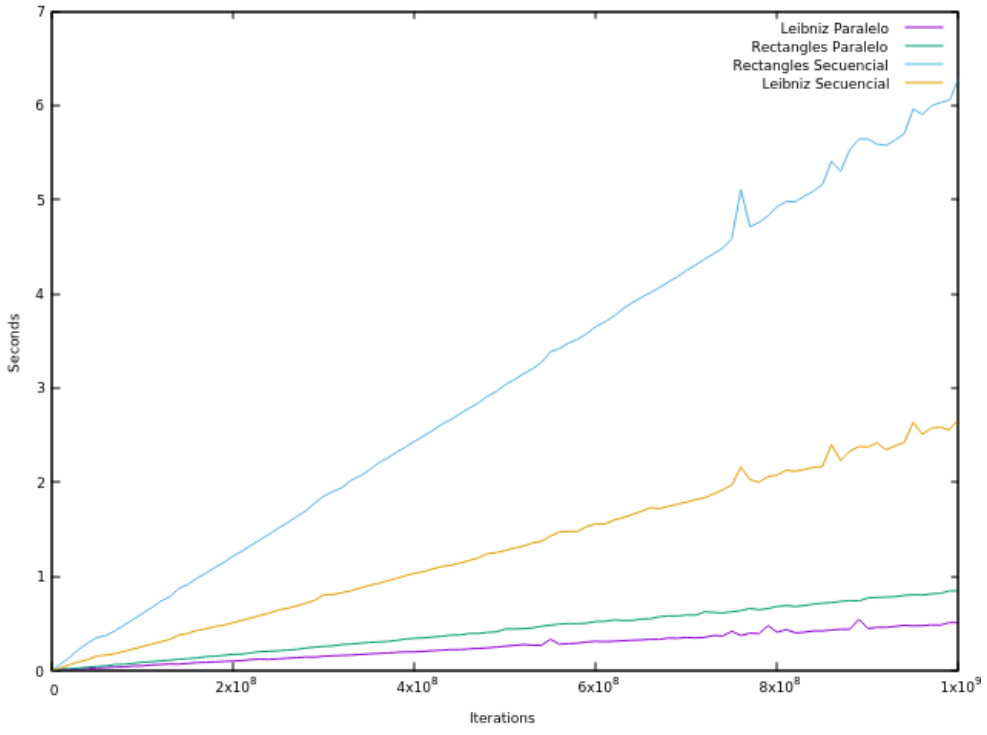
Si no se pone numero de procesos se ejecuta la versión secuencial del programa.

1.4.1 Localmente

Tras la ejecución del programa en la maquina local, medir ambos datos y hacer la grafica con GnuPlot usando los siguientes comandos:

```
>>set xlabel "Iterations"
>>set ylabel "Seconds"
>>plot "pi_parallel.dat" u 1:2 w l title "Leibniz
Paralelo","pi_parallel.dat" u 1:3 w l title "Rectangles
Paralelo","pi_secuencial.dat" u 1:3 w l title
"Rectangles Secuencial", "pi_secuencial.dat" u 1:2 w l
title "Leibniz Secuencial
"
```

Obtenemos la siguiente grafica. En los ejes lineales , para Y tenemos los segundos que ha tardado y en X tenemos la cantidad de iteracciones.

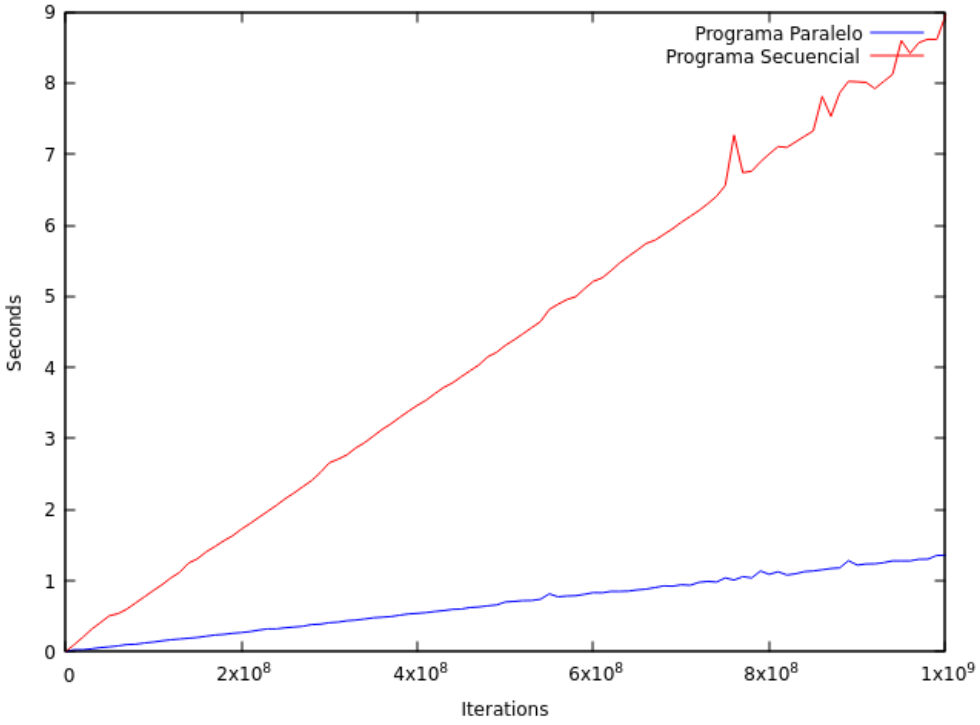


Vemos en nuestro caso que el mejor resultado se obtiene con la ejecución del algoritmo de Leibniz de Forma paralela , seguido de Rectangles. Donde la diferencia de rendimiento es muy cercana para los tamaños dados.
Sin embargo para una ejecución secuencial vemos que las diferencias son mucho mas marcadas. Más cuanto mas crecen las iteraciones .

Esta diferencia que hay entre los dos métodos disminuye cuantos mas nucleos se usen. Tras la ejecución con mas nucleos se puede observar que ambas líneas se van aproximando una a la otra.

Concluimos con que el método mas optimo es el de Leibniz y que para un uso máximo de nucleos de forma ideal las graficas de tiempo se superponen. Este no seria el caso ya que hay un punto donde la comunicación de los procesos supondría mas trabajo que el calculo en si.

Tambien se podría mirar de otra forma, si en vez de coger los procedimientos por separado los cojemos de forma conjunta, haciendo que se calcule el tiempo de ejecución del programa tenemos la siguiente grafica.



1.4.2 En ATCGrid

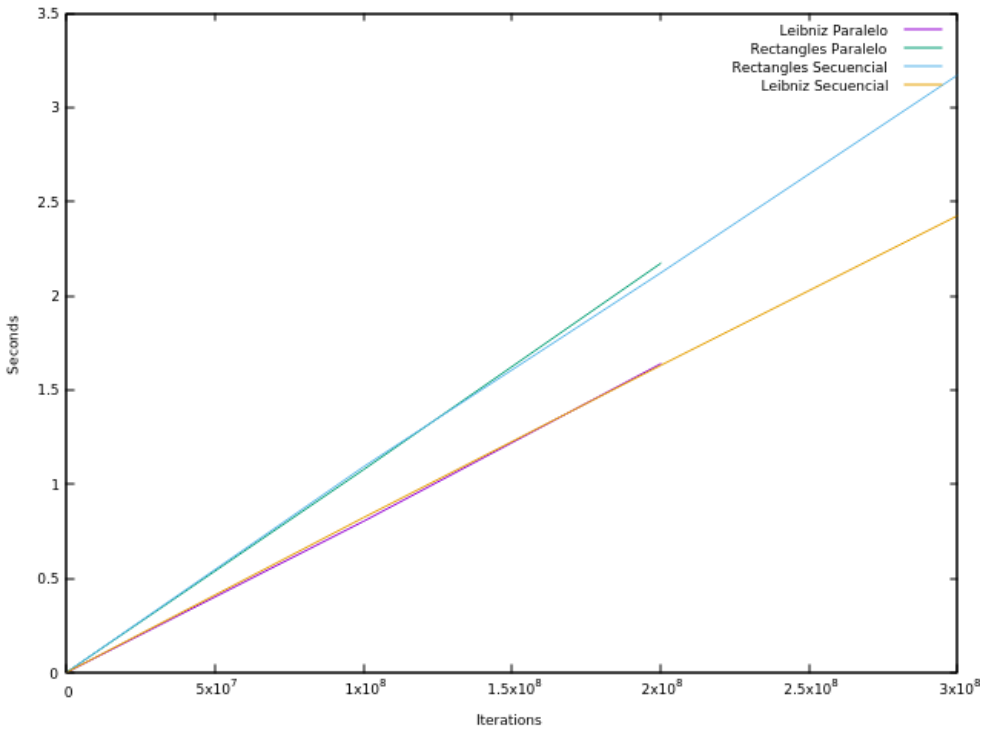
Para AtcGrid he seguido el mismo procedimiento que para la ejecución en local salvo que esta vez lo hemos hecho con Sbash para poder ejecutar los scripts.

```
sbatch -Aacap -p acap -N 1 -c 1 --hint=nomultithread --
exclusive --wrap "./comexecatc.sh pi 8"
```

En nuestro caso para la grafica de GNUPLOT usamos:

```
>>set xlabel "Iterations"
>>set ylabel "Seconds"
>>plot "pi_parallel_atc.dat" u 1:2 w l title "Leibniz
Paralelo","pi_parallel_atc.dat" u 1:3 w l title
"Rectangles Paralelo","pi_secuencial_atc.dat" u 1:3 w l
title "Rectangles Secuencial", "pi_secuencial_atc.dat"
u 1:2 w l title "Leibniz Secuencial"
```

El resultado obtenido es mas reducido en números ya que ATC tiene una restricción de tiempo de 32 segundos por lo que nuestro script no ha llegado muy lejos en calculo.



Como vemos el resultado tiene pocos valores pero podemos ver que los tiempos son muy parecidos entre si, eso si se sigue distinguiendo que en paralelo es mas rápido que secuencialmente.

1.5 Ganancia

Para calcular la ganancia del programa se ha elegido el máximo de iteraciones posible. En nuestro caso se ha hecho un Script para ejecutar solo con ese valor. El Script es comexecganancia.sh y se ejecuta de la forma.

```
./comexecganancia.sh pi_sec/pi N
```

Este nos dará los resultados de la ejecución en los archivos pi_secuencial_ganancia.dat y pi_paralelo_ganancia.dat. Juntaremos los datos de estos archivos con el comando paste de bash de la siguiente forma

```
paste pi_secuencial_ganancia.dat
pi_paralelo_ganancia.dat > SecParPi_Ganancia.dat
```

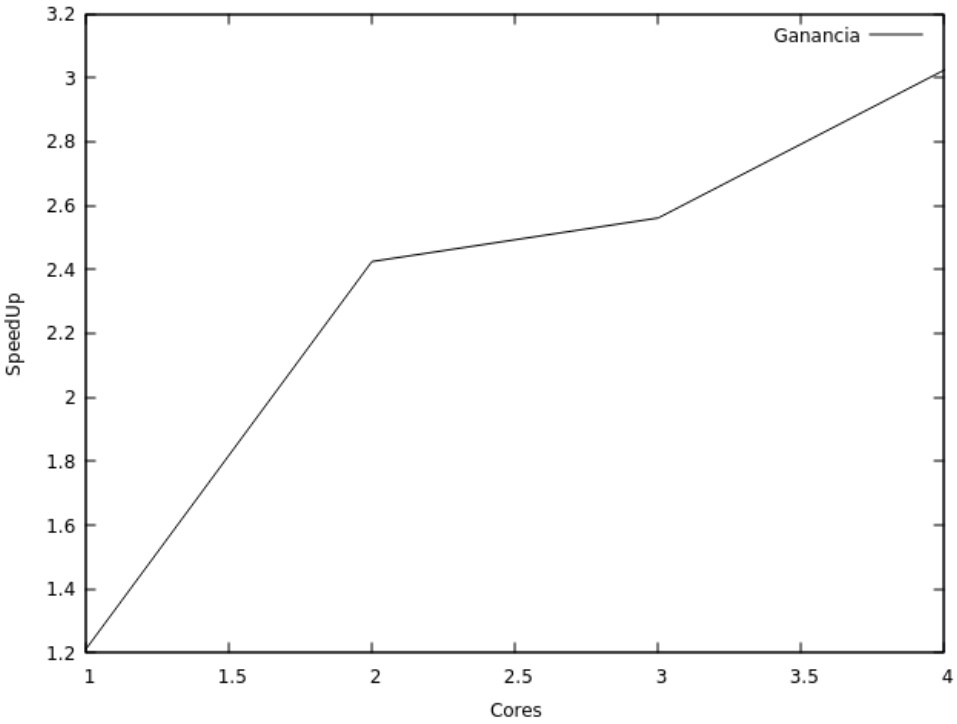
Este nos dará como resultado un archivo llamado "SecParPi_Ganancia.dat". Este lo usaremos con GNUPlot para mostrar la gráfica resultado de la ganancia. Lo haremos mediante el siguiente comando:

```
>>set xlabel "Cores"
>>set ylabel "SpeedUp"
>>plot "SecParPi_Ganancia.dat" u 1:(($2+$3)/($4+$5)) w
l title "Ganancia" lc rgb 'black'
```

Como sabemos el SpeedUp es S=(Tiempo Origen)/(Tiempo Mejorado).

$$Ganancia\ (S) = \frac{T_o}{T_m}$$

El resultado da la siguiente gráfica:



Al ser solo hasta 4 no somos capaces de ver la Ley de Amdalh a la perfección pero si pudiésemos seguir se podría ver como llega un punto donde esta gráfica se vuelve muy plana donde el rendimiento no puede mejorar casi nada. En nuestro caso si usáramos --oversuscribe veríamos incluso que el rendimiento llegado a un punto baja ya que la comunicación y creación de hebras virtuales sería más costoso que no tenerlas.