

Practica 1 ACAP

Hlib Bukhtiev

Ejercicio 1

Para implementar el producto escalar he creado dos arrays que se inicializan de forma que el valor de cada elemento del primer array es su índice y el valor de cada elemento del segundo array es "tamaño del array - su índice".

Para implementar el cálculo paralelo he hecho que el master mandara a cada proceso la parte de cada array que le corresponda. Para no sobrecargar el sistema con mensajes he decidido empaquetar los datos y mandarlos con un único mensaje. Sin embargo, en esta implementación faltaba un asunto por resolver – si el número de ítems del array no es múltiplo del número de procesos, el proceso cuyo identificador es el último se encargará de computar el resto de los dos arrays. Otra cosa a comentar es que en el seminario se nos propone usar funciones Probe y Get_Count porque el proceso a priori no sabría cuántos elementos recibiría, pero dada la naturaleza del problema que resolvemos simplemente se podría crear una función a la que llamaría cada proceso para saber cuántos elementos se le van a corresponder y reservar memoria previamente.

Capturas de ejecución:

```
● buttfucker3000@uwu:~/uni/acap/pl$ mpiexec -n 5 ej1
Invalid MIT-MAGIC-COOKIE-1 key

Array 1 :
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Array 2 :
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Scalar product: 560
```

```
buttfucker3000@uwu:~/uni/acap/pl$ mpiexec -n 2 ej1
Invalid MIT-MAGIC-COOKIE-1 key

Array 1 :
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Array 2 :
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Scalar product: 560
```

```

buttucker3000@uwu:~/uni/acap/p1$ mpiexec -n 8 ej1
Invalid MIT-MAGIC-COOKIE-1 key

Array 1 :
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Array 2 :
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Scalar product: 560

```

Ejercicio 2

Leibniz:

```

Execution pi effort : 10000
Media: 0.0000882

Execution pi effort : 50000
Media: 0.000269

Execution pi effort : 1000000
Media: 0.005565

Execution pi effort : 5000000
Media: 0.0196046

Execution pi effort : 10000000
Media: 0.0303952

Execution pi effort : 20000000
Media: 0.0600972

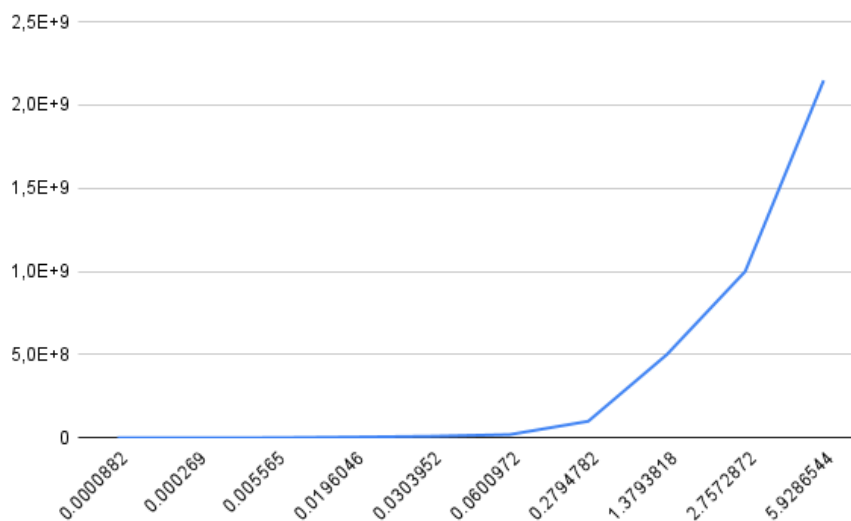
Execution pi effort : 100000000
Media: 0.2794782

Execution pi effort : 500000000
Media: 1.3793818

Execution pi effort : 1000000000
Media : 2.7572872

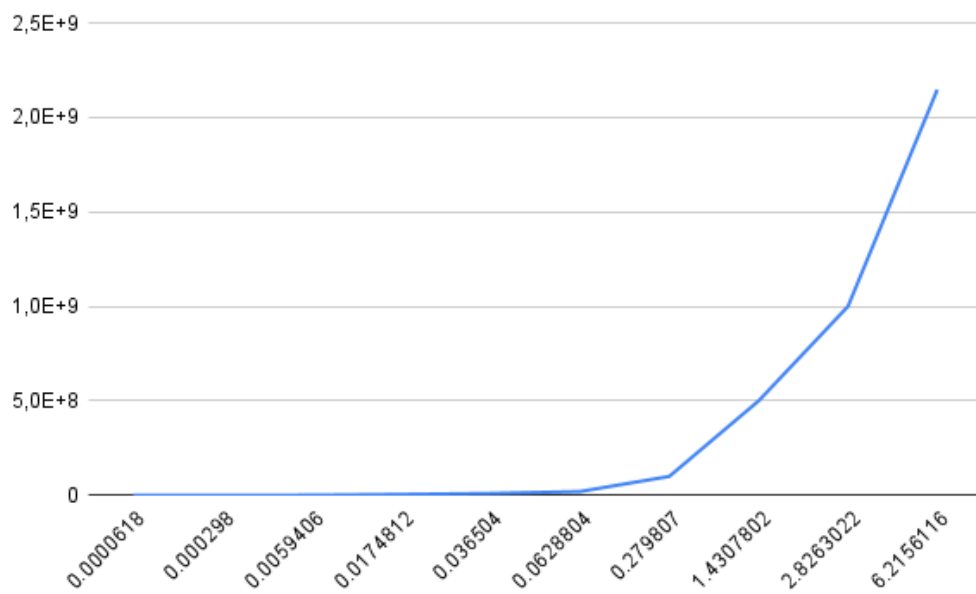
Execution pi effort : 2147483647
Media: 5.9286544

```



Rectangulos:

```
Execution pi effort : 10000  
Media: 0.0000618  
  
Execution pi effort : 50000  
Media: 0.000298  
  
Execution pi effort : 100000  
Media: 0.0059406  
  
Execution pi effort : 500000  
Media: 0.0174812  
  
Execution pi effort : 1000000  
Media: 0.036504  
  
Execution pi effort : 2000000  
Media: 0.0628804  
  
Execution pi effort : 10000000  
Media: 0.279807  
  
Execution pi effort : 50000000  
Media: 1.4307802  
  
Execution pi effort : 100000000  
Media: 2.8263022  
  
Execution pi effort : 2147483647  
Media: 6.2156116
```



Ejercicio 3

Pruebo ejecutar el código en el nodo de cómputo acap... lol:

```
[acap3@atcgrid ~]$ srun -Acap -pacap ./pi 2147483647

PI por la serie de G. Leibniz [2147483647 iteraciones] =      3.141593
Real time elapsed for computing PI with leibnith is 17.2378480 sec

PI por integración del círculo [2147483647 intervalos] =      3.141593
Real time elapsed for computing PI with rectangles is 22.6752870 sec
```

Explicación del código

Para paralelizar el programa de cálculo de PI he recurrido lo que había comentado en el ejercicio 1: cada proceso llama a una función que según su identificador, el número de intervalos que se hayan pasado por parámetro y el número de procesos que van a participar en el cómputo devuelve el intervalo en el que este proceso tiene que calcular PI. Para ello también he modificado ligeramente las funciones de cómputo para que en vez de calcular el rango entero de valores se acote de acuerdo a los intervalos proporcionados. En este caso, la función del máster es hacer el cómputo de una parte dada de PI y combinar los resultados una vez recibidos de los otros procesos, MPI_Reduce ha encajado perfectamente. Lo último a comentar es la medida de los tiempos – he puesto dos barreras para que master y el resto de procesos empiecen a trabajar simultáneamente tanto para Leibniz como para cálculo con área.

Comparación de la versión secuencial con la paralelizada entre 8 procesos:

```
buttfucker3000@uwu:~/uni/acap/p1$ mpiexec -n 1 ej3 2147483647
Invalid MIT-MAGIC-COOKIE-1 key
PI calculated by rectangles : 3.141593
Wall time consumed : 6.101188

PI calculated by leibniz : 3.141593
Wall time consumed : 5.917531

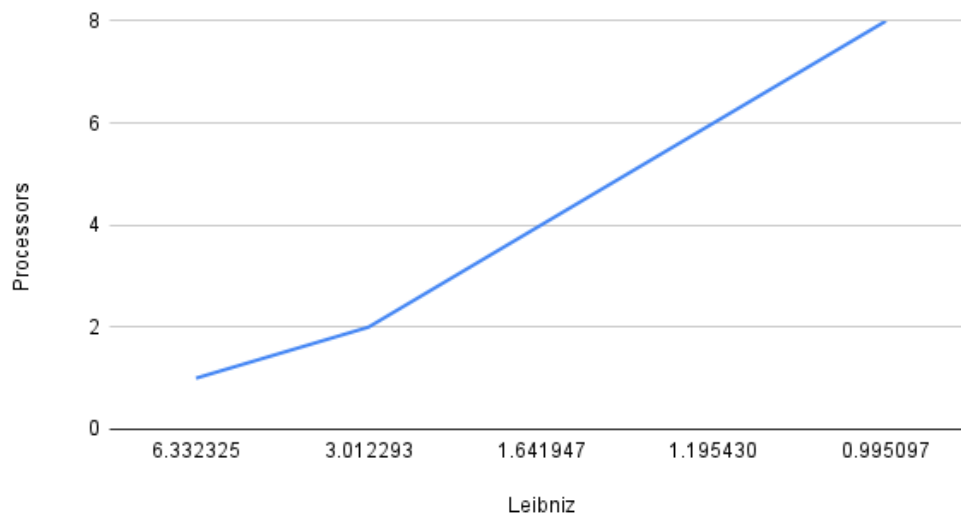
buttfucker3000@uwu:~/uni/acap/p1$ mpiexec -n 8 ej3 2147483647
Invalid MIT-MAGIC-COOKIE-1 key
PI calculated by rectangles : 3.141593
Wall time consumed : 1.041980

PI calculated by leibniz : 3.141593
Wall time consumed : 0.975229
```

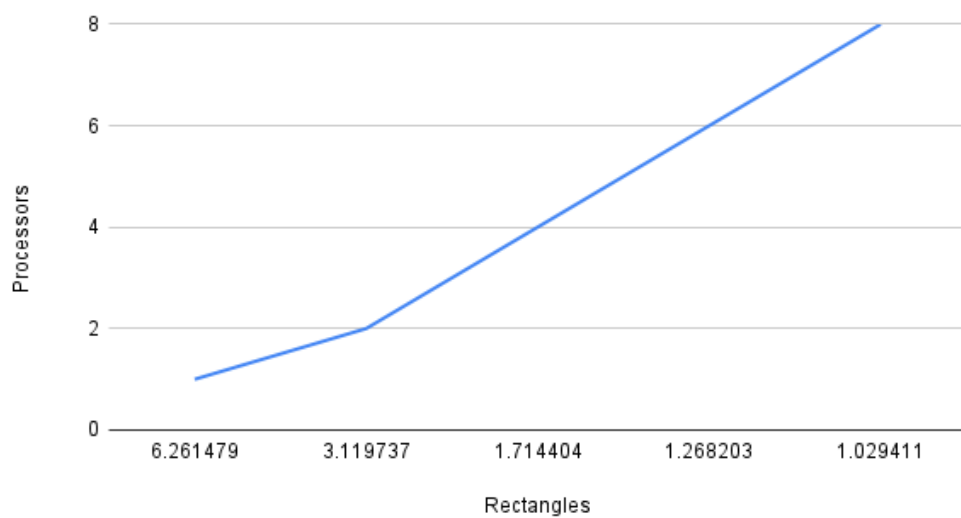
Calculo la aceleración de cada una de las versiones paralelas respecto a la secuencial:

Processors	Leibniz	Aceleracion	Rectangles	Aceleracion
1	6.332325	1.0	6.261479	1.0
2	3.012293	2.102161	3.119737	2.029762
4	1.641947	3.856595	1.714404	3.693601
6	1.195430	5.297110	1.268203	4.993147
8	0.995097	6.363525	1.029411	6.151405

Processors относительно параметра "Leibniz"



Processors относительно параметра "Rectangles"



Observamos que al añadir más procesadores para paralelizar el programa tenemos un tiempo de ejecución mejor, sin embargo la mejora no es lineal. En las condiciones reales, conforme se vayan añadiendo más niveles de paralelismo, la mejora va empeorando porque, aparte de la sobrecarga hardware generada, la fracción del programa paralelizable disminuye con lo que el impacto de añadir más procesadores también disminuye.