ACAP

Práctica 3

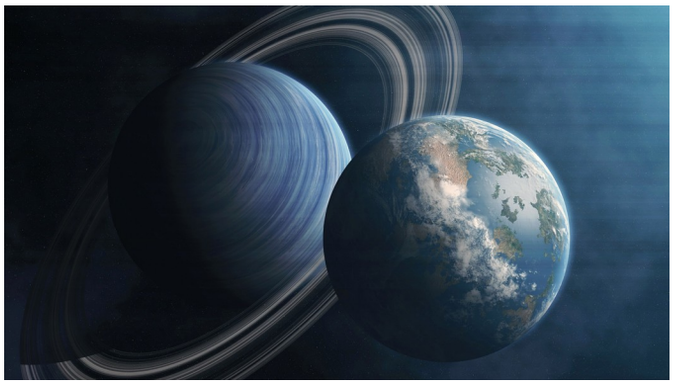
Antonio Priego Raya

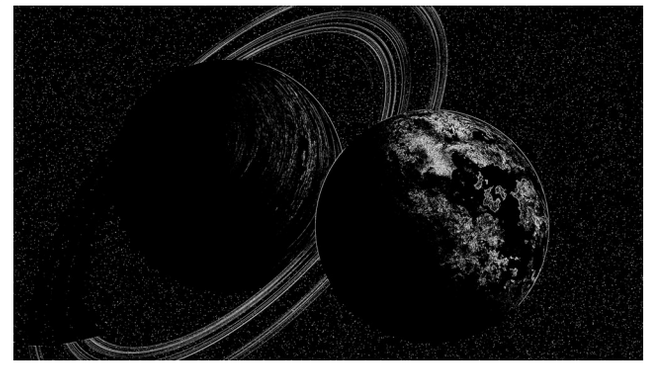
Filtro de Sobel – Paralelismo con MPI.

En primer lugar, aclarar que finalmente, como no he recibido instrucciones sobre qué hacer y he acabado resolviendo la práctica completamente con openCV, no he podido compilar y ejecutar mi código en el ATCgrid.
Por tanto, todos los datos que muestro en esta memoria son de mi portátil, para lo cual es importante recordar que tiene 4 cores. Intentaré compensarlo haciendo un análisis razonado de qué obtendría frente a qué he obtenido.

De igual forma, si se me comunica alguna forma de resolver esta situación o se acaba instalando la librería, no tengo problema en modificar todo este documento, ya que sería cambiar los campos de los tiempos de la hoja de cálculo por los del ATCgrid.

Puedo igualmente suponer qué tiempos arrojaría, pese a que no serían semejantes a los que he obtenido con mi portátil. Lo iré comentando e iré razonando por qué sí o por qué no, los tiempos que he obtenido serían representativos respecto al ATCgrid (guardando la evidente mejora en presentaciones, hago referencia a la proporcionalidad).





DESCRIPCIÓN DEL PROGRAMA, CARGAS E IMPLEMENTACIONES (1)

El programa va a realizar un procesamiento de imágenes basado en gradientes. En este caso aplicaremos la máscara de Sobel:

```
//
// Devuelve el gradiente de 'x' del punto(x,y) de la imagen
//    Proceso: Aplicar la máscara
//
//   Orden       Máscara (x)
// [z1 z2 z3]    [-1  0  1]
// [z4 z5 z6]    [-2  0  2]
// [z7 z8 z9]    [-1  0  1]
//
// Gx= (z3+2*z6+z9) - (z1+2*z4+z7)
```

Ilustración 1: Comentarios previos a funcion x_gradient() en mi código

```
//
// Devuelve el gradiente de 'y' del punto(x,y) de la imagen
//    Proceso: Aplicar la máscara
//
//   Orden       Máscara (y)
// [z1 z2 z3]    [-1 -2 -1]
// [z4 z5 z6]    [ 0  0  0]
// [z7 z8 z9]    [ 1  2  1]
//
// Gy= (z7+2*z8+z9) - (z1+2*z2+z3)
```

Ilustración 2: Comentarios previos a funcion y_gradient() en mi código

De forma que, aplicando este filtro a cada pixel, para lo cual debemos tener en cuenta que necesitamos los píxeles inmediatos; obtenemos la imagen procesada. En este caso, el procesamiento se aplica a detección de bordes. También tener en cuenta que es un filtro aplicable a escala de grises.

He utilizado como muestras de carga 3 imágenes:

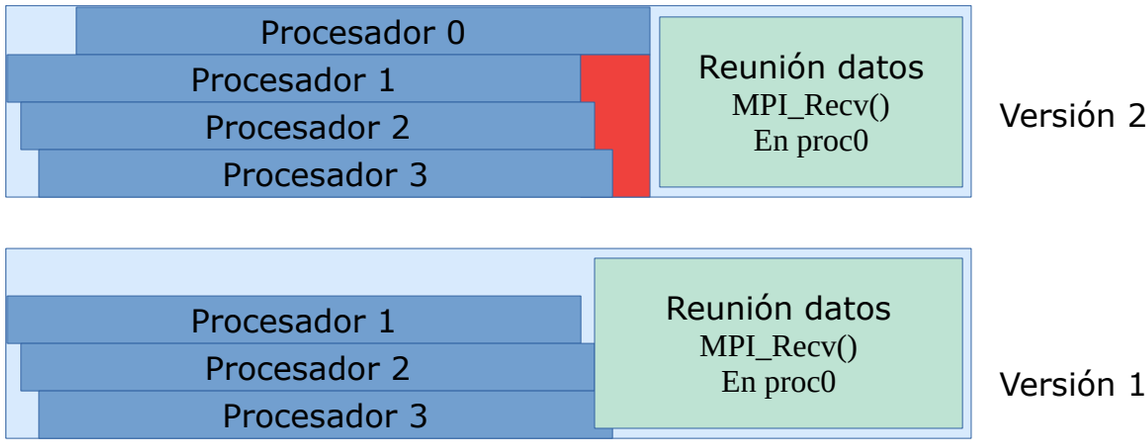
- Muestra muy pequeña 793x533
- Muestra grande 3840x2160(4K)
- Muestra muy grande 7680x4320(8K)

He elegido estos tamaños, porque quería un tamaño que evidenciara un procesamiento de carga baja, uno de carga media, aunque consideremos una resolución alta, y por último una carga alta que evidenciara un procesamiento de carga considerable.

Mis dos implementaciones finales se basan en la idea de procesos maestro-esclavo. En la v1 hago que un procesador(id=0) se encargue exclusivamente de repartir la carga al resto de procesadores y reunir los resultados.
Sin embargo, pese a que no es una versión privativamente mejor, en un contexto de "pocos recursos", en este caso cores; la v2 está mejorada. Ya que ponemos al procesador 0 a trabajar y realizar el filtrado una vez termina el reparto, para que no permanezca ocioso mientras espera la recolección de datos.

DESCRIPCIÓN DEL PROGRAMA, CARGAS E IMPLEMENTACIONES (2)

En un contexto de muchos más procesadores, la v1 sería mejor, ya que, el procesador 0 puede tardar más que el resto, y esto haría que el resto de procesadores tardara más en enviar su procesamiento, y por tanto, en llegar a reunirse toda la imagen de nuevo.
Gráficamente, ocurriría algo así:



Sin embargo, debemos tener en cuenta que en la versión 1 hay un procesador menos realizando cómputo. Por tanto la ejecución de cada uno de los procesadores será mayor, ya que la carga está menos repartida.
Al estar trabajando con pocos procesadores, la versión 2 hace que notemos una mejora respecto a la primera versión.
No ocurriría lo mismo en un cluster de 100PCs (caso desproporcionado para nuestras muestras de carga, planteado para enfatizar la reflexión), donde un procesador más que menos, no es relevante.

Algo que no varía entre las dos versiones es el envío de trozos. En mi código he llamado trozo al conjunto de filas de la imagen que le toca procesar a cada procesador, de manera que se aprovecha el envío de mensajes al máximo. Solo se hace un envío a cada procesador con su trozo más la(s) fila(s) de vecindad. Esto es así porque decidiendo el tamaño de los trozos, antes de programar, me di cuenta de que solo sería eficiente dividir los trozos si la carga de trabajo fuera muy alta y si implementaramos el paso de mensajes de v1, dejando al proceso 0 "ocioso" a espera de reunir y enviar microtrozos de cada proceso.

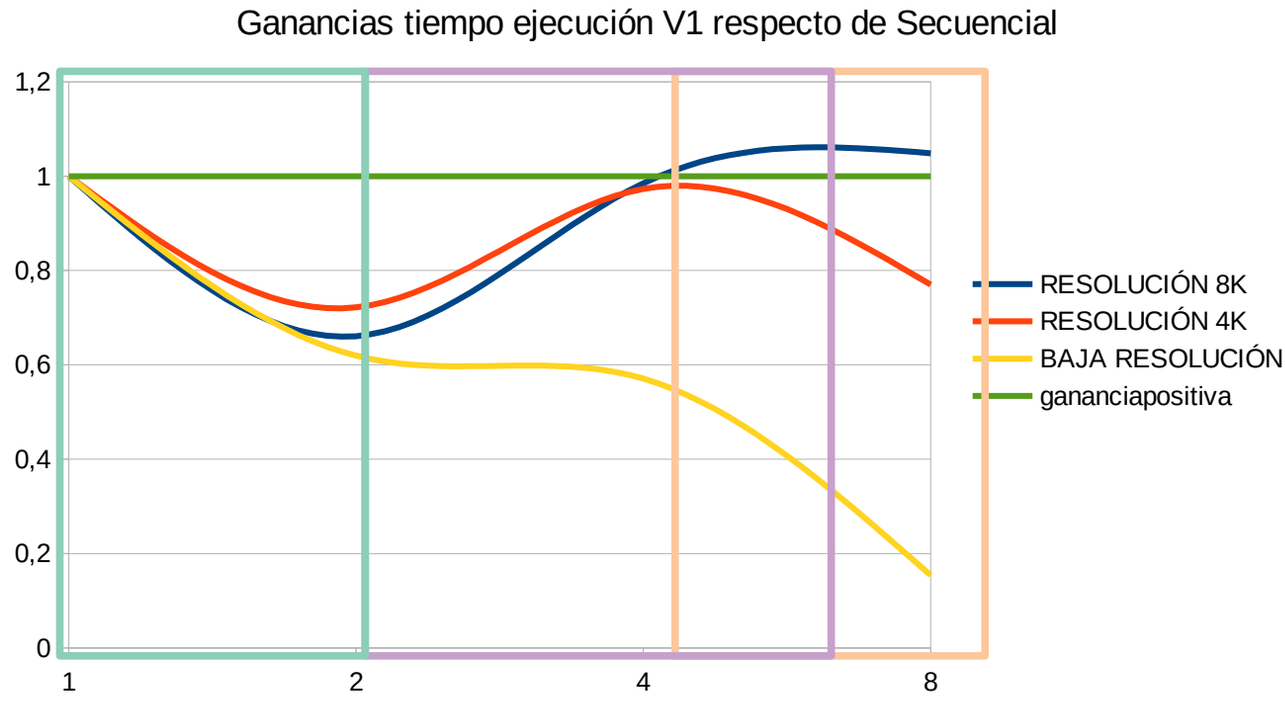
De nuevo, nuestro caso es un programa con cargas de juguete, por lo tanto, el tiempo que emplearíamos en el paso de mensajes, supondría pérdidas teniendo en cuenta que el reparto de carga de una muestra baja, no tiene mucho sentido.

ESTUDIO DE LOS TIEMPOS DE EJECUCIÓN OBTENIDOS v1 (1)

En las mediciones, hay que tener en cuenta que la v1 no es capaz de ejecutarse con un procesador, ya que sería el procesador maestro.
De igual forma, los tiempos obtenido para v2 con un procesador no coinciden con lo que se esperaría, así que los he obviado.

Yo voy a analizar las ganancias porque nos dan información suficiente para el estudio, pero el resto de datos los dejo en la última página, aunque también están disponibles en la hoja de cálculo y en los ficheros de /data , resultado de ejecutar el script en /script .

Comenzamos con las ganancias obtenidas para la versión 1.



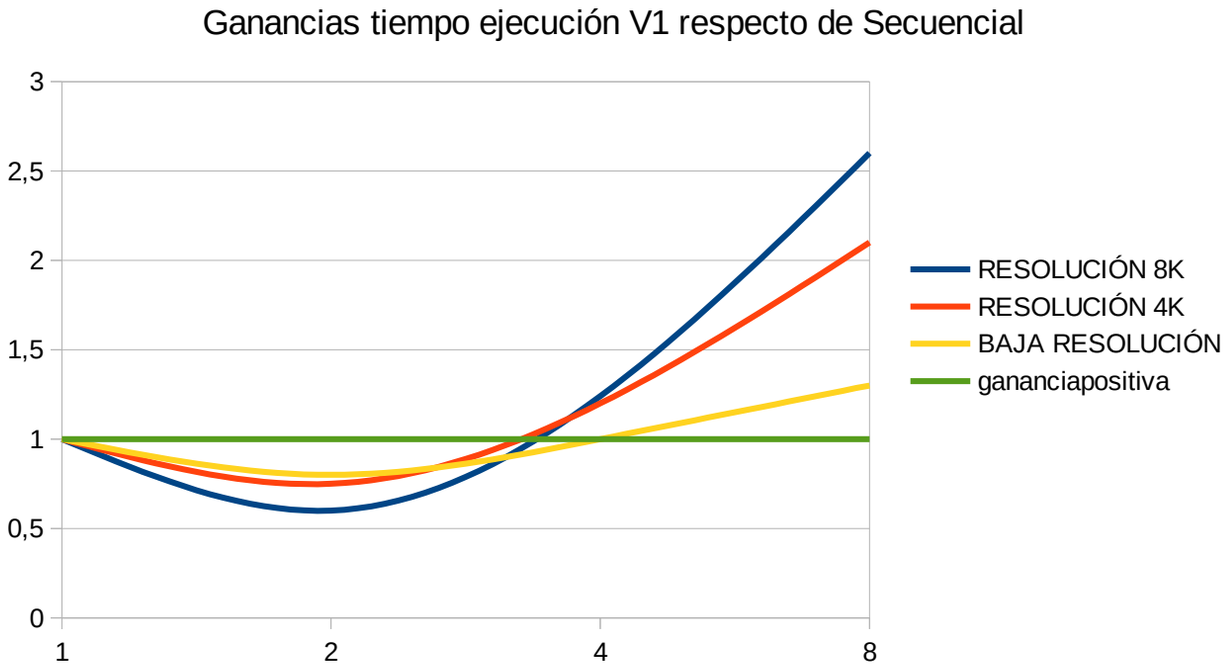
Ahora es cuando tengo que explicar todo lo que expuse en la introducción.
Y es que, como vemos, no hay escalabilidad a partir de 4 procesadores. Esto se debe a que mi portátil no posee 8 núcleos para poder hacer una ejecución simultánea de 8 procesos. Hace uso de los 2 hilos de los 2 cores que tiene, pero repartiendo los 8 procesos en el tiempo entre los 4 hilos.

Sin embargo, en el tramo de 2 a 4 procesos, sí que notamos una mejora. Aunque sigue sin alcanzar a los tiempos secuenciales. Y también observamos que sí que hay algo de escalabilidad más allá de los 4 procesos, esto se debe a que el proceso 0 está destinado, como ya introduje anteriormente, a logística de mensajes. Por tanto, pese a que sean "virtuales", se sigue apreciando algo de mejora al añadir más procesos; exclusivamente con carga alta, ya que con el resto de cargas, los procesos atribuidos a procesadores "virtuales", solo suponen más tiempo de creación de procesos y espera innecesaria.

Por último, y como era de esperar, con 2 procesadores, la mejora es nula, ya que introducimos creación y destrucción de procesos, envío de mensajes y

demás procedimientos solo para añadir un procesador de cálculo, ya que el proceso 0 en este caso solo entorpece los tiempos.

Para terminar con esta primera versión. Voy a hacer el ejercicio de razonar qué obtendríamos en el ATCgrid, de forma aproximada.



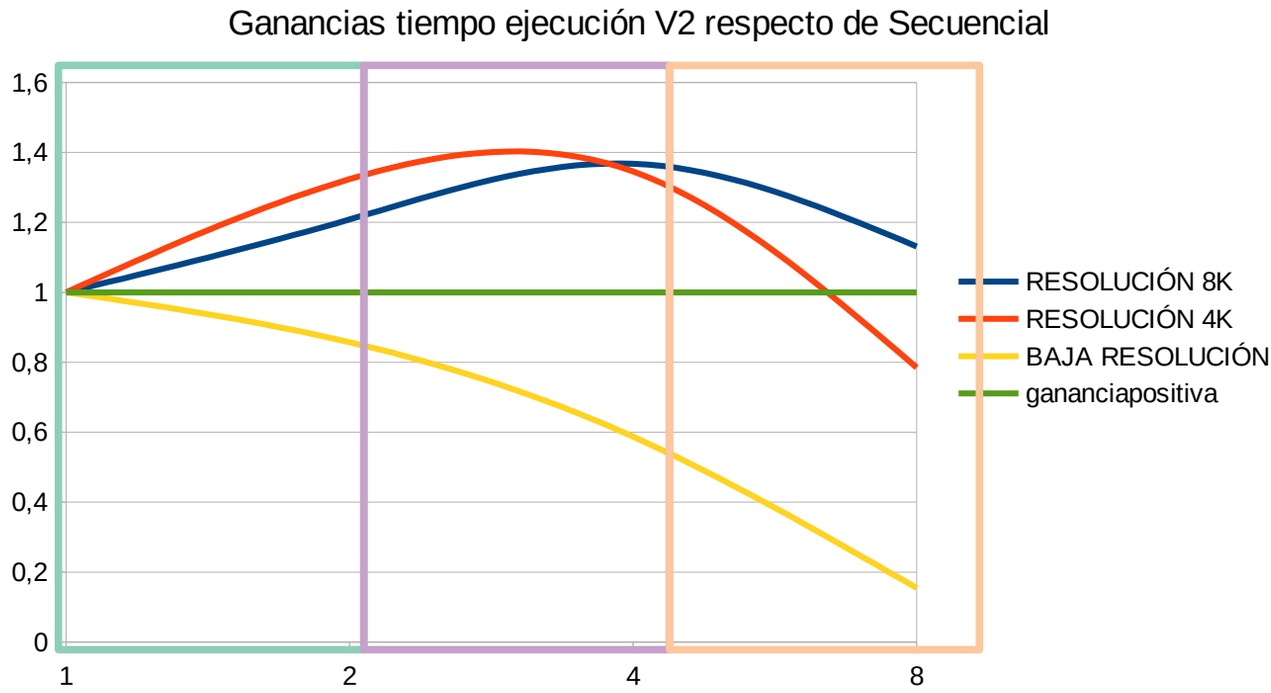
Con 2 procesadores, no obtendríamos buenos resultados, por el mismo motivo que expliqué anteriormente.
Cosa que se acentuaría con mayores cargas.

Con 4 procesadores comenzaríamos a remontar la línea para igualar al tiempo secuencial. 3 procesos de cálculo me parecen suficientes para compensar la implementación de herramientas para paralelismo. Sobre todo con cargas altas.

Por último, con 8 procesadores comenzaría la mejora real, el punto donde la implementación de paralelismo tiene sentido. Ahora sí, ya que el ATCgrid puede ejecutar 8 procesos de forma simultánea real.

ESTUDIO DE LOS TIEMPOS DE EJECUCIÓN OBTENIDOS v2 (1)

Ya expliqué por qué v2 es superior a v1 para el número de procesos que he elegido, por tanto, vamos a analizar directamente los resultados.



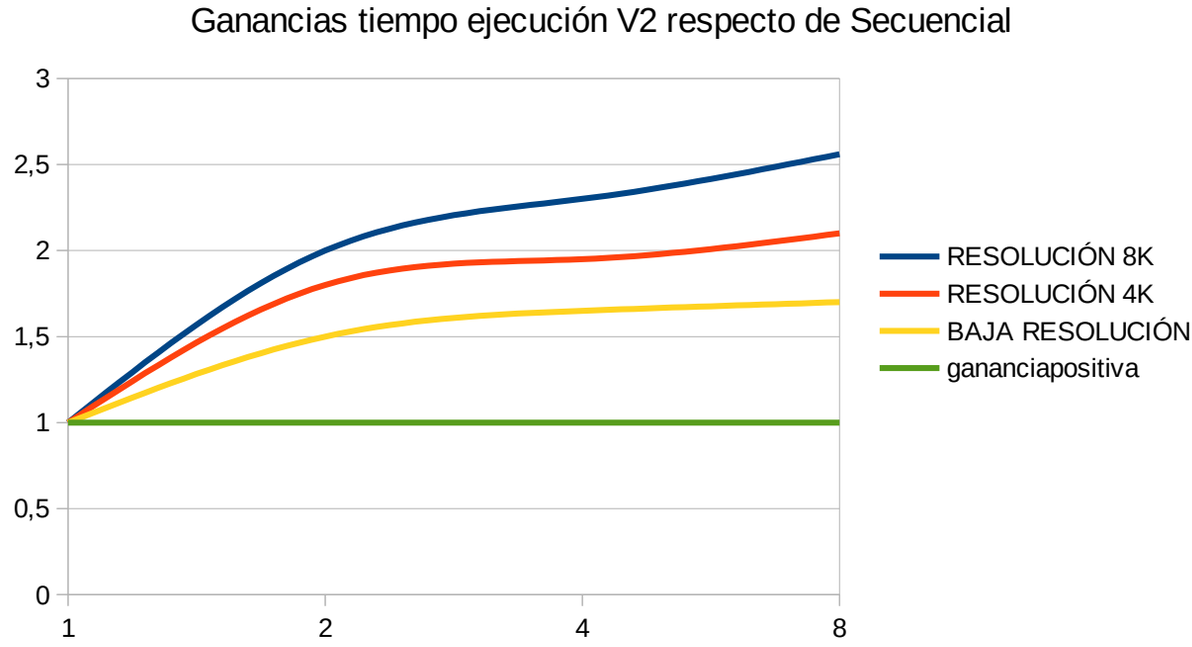
Vemos que ahora sí obtenemos ganancias mayores que 1, porque estamos aprovechando nuestro proceso 0 al máximo. Para estos números de procesos tan bajos, la espera al proceso 0 no ocasiona un perjuicio mayor al apoyo que ofrece descargar de un trozo menos al resto de procesos.

Si bien, para 4 procesos, no obtenemos tanta mejora como pensábamos en cargas bajas. Debido a que, con esta versión obtenemos más ganancia con pocos procesadores, pero en decremento de menor escalabilidad para más procesadores.

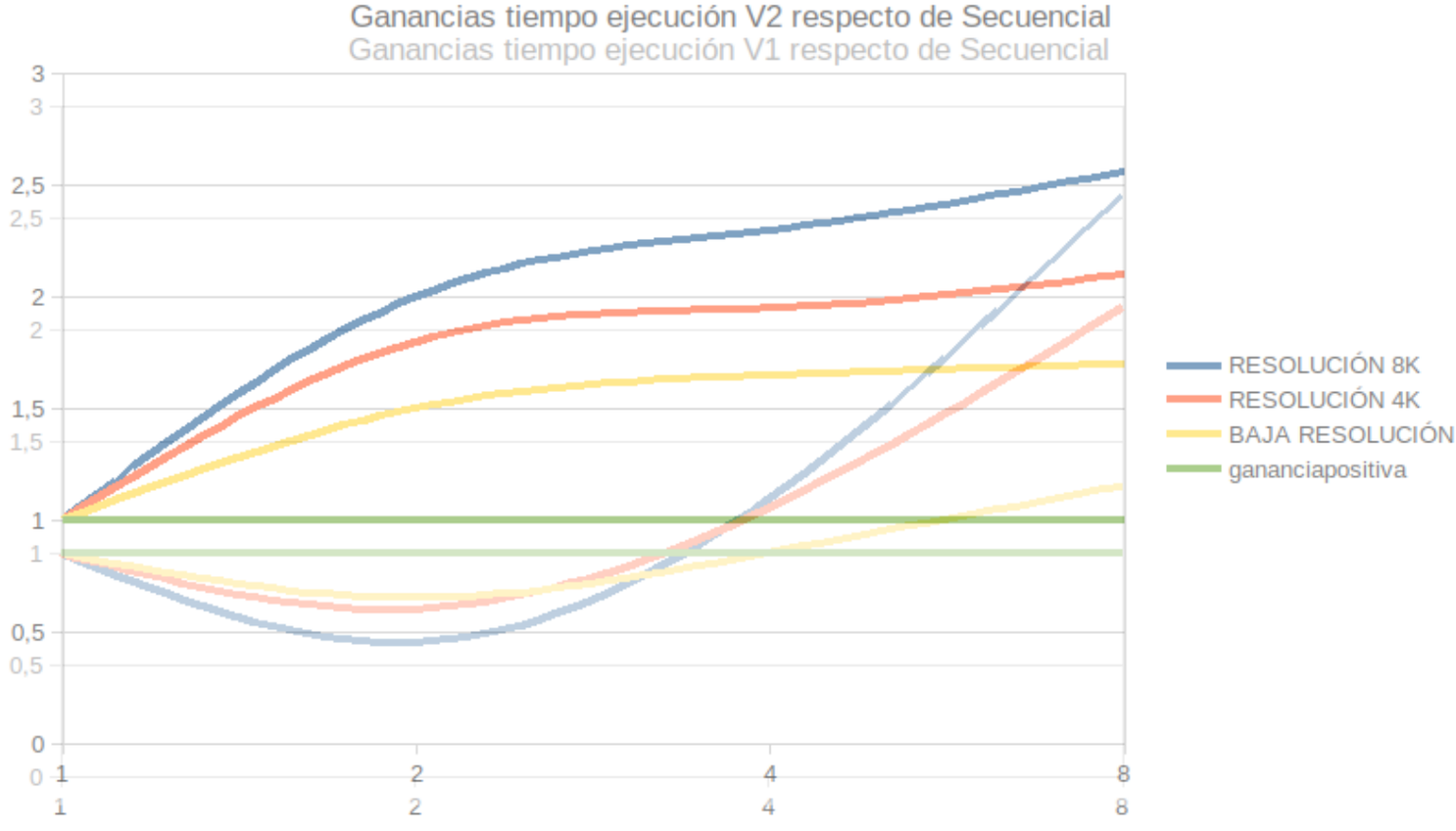
Ocurre lo mismo que en la v1 para más de 4 procesos. Que los datos no son concluyentes porque no representan la realidad de una ejecución con 8 procesadores. Aun así, para una alta carga, seguimos obteniendo mejoras sobre ejecución secuencial.

ESTUDIO DE LOS TIEMPOS DE EJECUCIÓN OBTENIDOS v2 (2)

Turno de volver a suponer los resultados del ATCgrid.



Obtendríamos una gráfica de ganancias donde la escalabilidad a partir de 4 procesos más tenue que en v1, pero para 2 habría una mejora muy sustancial respecto a v1.



CONCLUSIÓN

La conclusión es un pequeño resumen de todo lo que he ido comentando.
Si queremos aprovechar el paralelismo con pocos procesadores, tendremos que utilizar la segunda versión. Mientras que si queremos una escalabilidad para altas prestaciones, deberemos escoger la primera versión. De 2 a 8 procesadores nos conviene hacer uso de la segunda versión, como hemos visto en el estudio de los resultados y venía previendo desde el comienzo.

Por lo tanto no podemos decir de forma determinista que la primera versión sea peor que la segunda.

Una idea interesante, sería consultar el número de procesadores dentro del programa. Y utilizar una u otra versión en función del número de procesos elegidos.

DATOS EXTRAIDOS CRUDOS

Extracción de hoja de cálculo (en /data):

Vers | T(s)	1	2	4	8	Tiempos Sec
Secuencial	0,04903825				0,0497963
v1		0,07916193	0,08591977	0,3187491	0,0494426
v2		0,05720761	0,08353433	0,31790326	0,0483628
Gananc v1	1	0,6194676	0,57074468	0,15384593	0,0485513
Gananc v2	1	0,85719807	0,58704306	0,15425526	

BAJA RESOLUCIÓN

	v1(n=2)	v1(n=4)	v1(n=8)	v2(n=1)	v2(n=2)	v2(n=4)	v2(n=8)
Tiempos Calculo	0,0563785	0,0291631	0,0267727	6,32404e-32	0,0289333	0,0220664	0,0400314
	0,0555262	0,0286544	0,0318474	6,32404e-32	0,0404652	0,0218394	0,0420043
	0,0529525	0,0366291	0,0322404	6,32404e-32	0,0262312	0,0223756	0,0443935
	0,0510884	0,0292357	0,0382004	6,32404e-32	0,0288856	0,0236884	0,0363024
Tiempos Recepción	0,00062437	0,00032882	0,00022143	-5,61E+158	0,00039564	0,00094303	0,00213221
	0,00064315	0,0002541	0,000226	6931,45	0,00034897	0,00092416	0,00186094
	0,00063521	0,00028101	0,00023378	5,83E+183	0,00034371	0,00106987	0,00150064
	0,00061708	0,0611318	0,00024728	4,05E+261	0,0003834	0,00353457	0,00902263
Tiempos Creación y Destrucción procesos	0,0274408	0,0451455	0,290706	0,0158521	0,0245818	0,0600539	0,29058
	0,023531	0,0492828	0,288885	0,0160619	0,0299929	0,0589829	0,225898
	0,0242675	0,00031363	0,293337	0,0162059	0,0226438	0,0602284	0,277716
	0,022943	0,0632591	0,272079	0,0167855	0,0256249	0,0584307	0,300171

Vers | T(s)	1	2	4	8	Tiempos Sec
Secuencial	0,948508				0,945157
v1		1,31421714	0,97476799	1,2315864	0,94784
v2		0,71669486	0,70497638	1,20911798	0,948452
Gananc v1	1	0,72172853	0,97306027	0,77015141	0,952583
Gananc v2	1	1,32344748	1,34544651	0,78446274	

RESOLUCIÓN 4K

	v1(n=2)	v1(n=4)	v1(n=8)	v2(n=1)	v2(n=2)	v2(n=4)	v2(n=8)
Tiempos Calculo	1,12843	0,600695	0,613787	6,32404e-32	0,54578	0,465309	0,540365
	1,13706	0,594198	0,6801060		0,553978	0,447618	0,555028
	1,13622	0,762745	0,5011430		0,545094	0,45268	0,5501
	1,13617	0,755588	0,5170866	6,32404e-32	0,552858	0,459757	0,504836
Tiempos Recepción	0,00968564	0,00558407	0,00565346	6870,47	0,00655705	0,0117095	0,0357942
	0,00858742	0,00563048	0,00478868	6873	0,00655041	0,0152908	0,0202923
	0,0100171	0,00570506	0,00492882	6,88E+03	0,00654377	0,0117132	0,0217932
	0,00904439	0,00672135	0,00478063	6,88E+03	0,00659819	0,011696	0,0707552
Tiempos Creación y Destrucción procesos	0,184319	0,32656	0,6058350	0,0919027	0,160225	0,236704	0,639738
	0,165822	0,320764	0,58267	0,0917855	0,159503	0,235902	0,644149
	0,167095	0,259951	0,732553	0,0910028	0,159825	0,235013	0,659838
	0,164418	0,25493	0,673014	0,0919229	0,163267	0,236513	0,593783

Vers | T(s)	1	2	4	8	Tiempos Sec
Secuencial	3,8426575				3,83364
v1		5,81486878	3,90141273	3,66449498	3,843
v2		3,18026018	2,8099748	3,39820575	3,84008
Gananc v1	1	0,66083306	0,98494001	1,04861858	3,85391
Gananc v2	1	1,208284	1,36750604	1,13079012	

RESOLUCIÓN 8K

	v1(n=2)	v1(n=4)	v1(n=8)	v2(n=1)	v2(n=2)	v2(n=4)	v2(n=8)
Tiempos Calculo	5,25712	2,64418	2,569986	6,32404e-32	2,70226	2,11912	2,23337
	5,48002	2,55711	2,585636	6,32404e-32	2,67322	2,08086	2,20279
	5,28396	2,68169	2,226746	6,32404e-32	2,70844	2,16846	2,0264
	5,20116	3,68018	2,512066	6,32404e-32	2,73063	2,17221	2,23017
Tiempos Recepción	0,0570865	0,0235825	0,023874	7,07E+03	0,0296905	0,0678341	0,103114
	0,0561291	0,0234901	0,0226758	7082,57	0,0310772	0,127906	0,138302
	0,0558177	0,0266482	0,0229684	1,16E+194	0,0297967	0,0575413	0,251173
	0,0585648	0,0256601	0,0230417	7,10E+03	0,0303953	0,0765138	0,133446
Tiempos Creación y Destrucción procesos	0,484947	0,973176	1,13223	0,115879	0,438237	0,563509	1,04619
	0,421978	1,10523	1,04355	0,113274	0,441169	0,633896	0,989608
	0,467003	1,26596	1,3992	0,114171	0,464392	0,578772	1,19361
	0,435689	0,598744	1,09603	0,116879	0,441733	0,593277	1,04465

Tiempos obtenidos de la ejecución del script en /script. También disponibles en texto plano en /data.