REPÚBLICA BOLIVARIANA DE VENEZUELA

MINISTERIO DEL PODER POPULAR PARA LA EDUCACIÓN SUPERIOR

UNIVERSIDAD CENTRO OCCIDENTAL LISANDRO ALVARADO

DECANATO DE CIENCIAS Y TECNOLOGÍA

BARQUISIMETO – ESTADO LARA

Asignación 2

Integrantes:

Torrealba Luis

C.I.: 26.121.249

Michael Montero

C.I.:26561077

Asignatura:

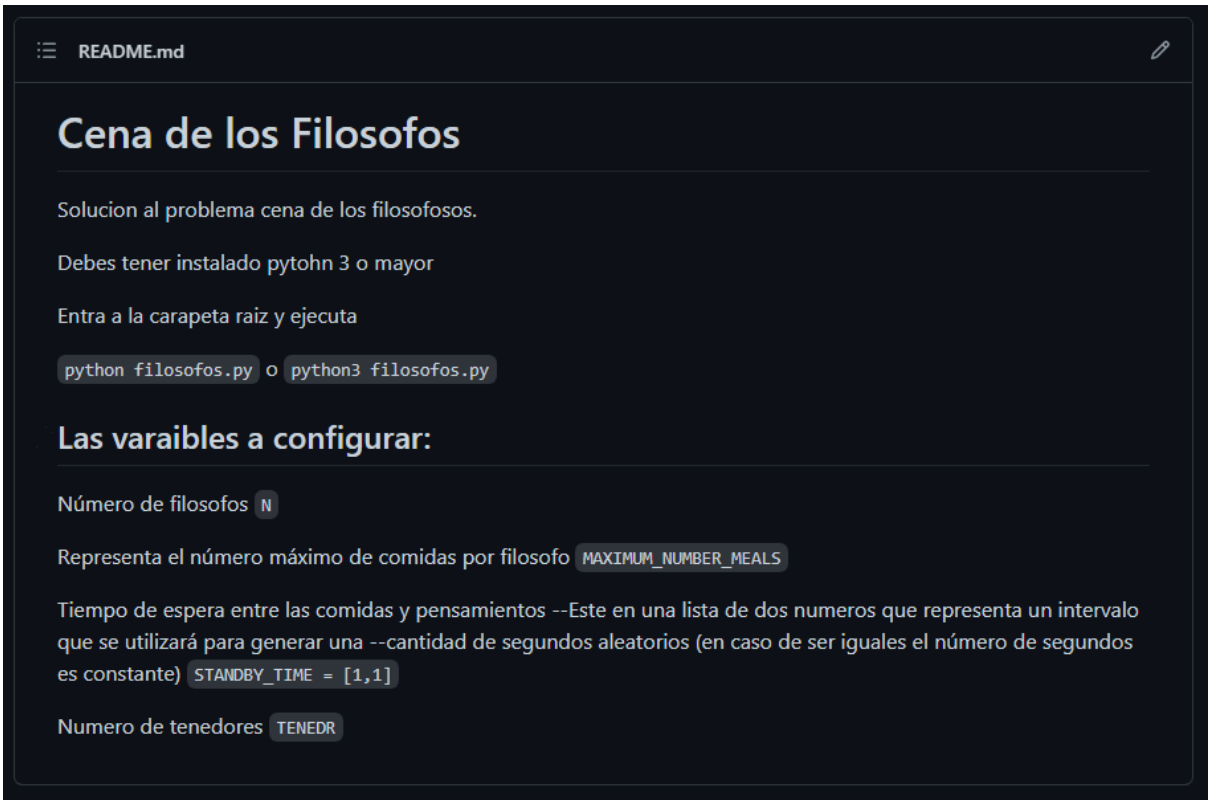
Sistemas Operativos

Barquisimeto, 07 de Julio del 2021

Repositorio del código implementado

https://github.com/Damurq/so_filosofos

(Instrucciones en el archivo README.MD)


README.md

Cena de los Filosofos

Solucion al problema cena de los filosofosos.

Debes tener instalado pytohn 3 o mayor

Entra a la carapeta raiz y ejecuta

`python filosofos.py` o `python3 filosofos.py`

Las varaibles a configurar:

Número de filosofos `N`

Representa el número máximo de comidas por filosofo `MAXIMUM_NUMBER_MEALS`

Tiempo de espera entre las comidas y pensamientos --Este en una lista de dos numeros que representa un intervalo que se utilizará para generar una --cantidad de segundos aleatorios (en caso de ser iguales el número de segundos es constante) `STANDBY_TIME = [1,1]`

Numero de tenedores `TENEDR`

1-Aumentar el número de filósofos:

El número de filósofos aumenta al mismo tiempo que lo hace el de tenedores, teniendo entonces que si la cantidad de filósofos es **n**, el número de tenedores también lo es, además se debe recordar que el número máximo de filósofos que pueden comer al mismo tiempo es de:

- **n/2 cuando el número de tenedores es par**
- **(n-1)/2 cuando el número de tenedores es impar**

Donde **“n”** es el número de **tenedores** .

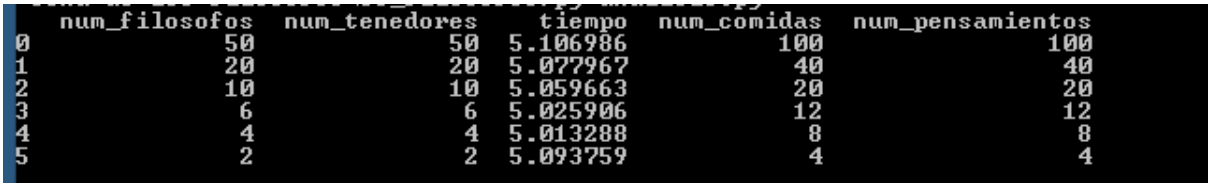
Siendo de este modo procedimos a realizar varias pruebas para ver cómo se comporta el algoritmo al aumentar el número de filósofos:

Prueba 1 (Aumentar número de filósofos, números pares de filósofos):

Condiciones:

- El tiempo que tardan los filósofos en comer o pensar es de 1 segundo para todos (posteriormente se realizará una prueba con un tiempo aleatorio).
- Todos los filósofos deben comer y pensar la misma cantidad de veces (en este caso se decidió que el número de veces sería 2), de este modo nos aseguramos que ningún filósofo se apropie de todos los recursos.

Datos:

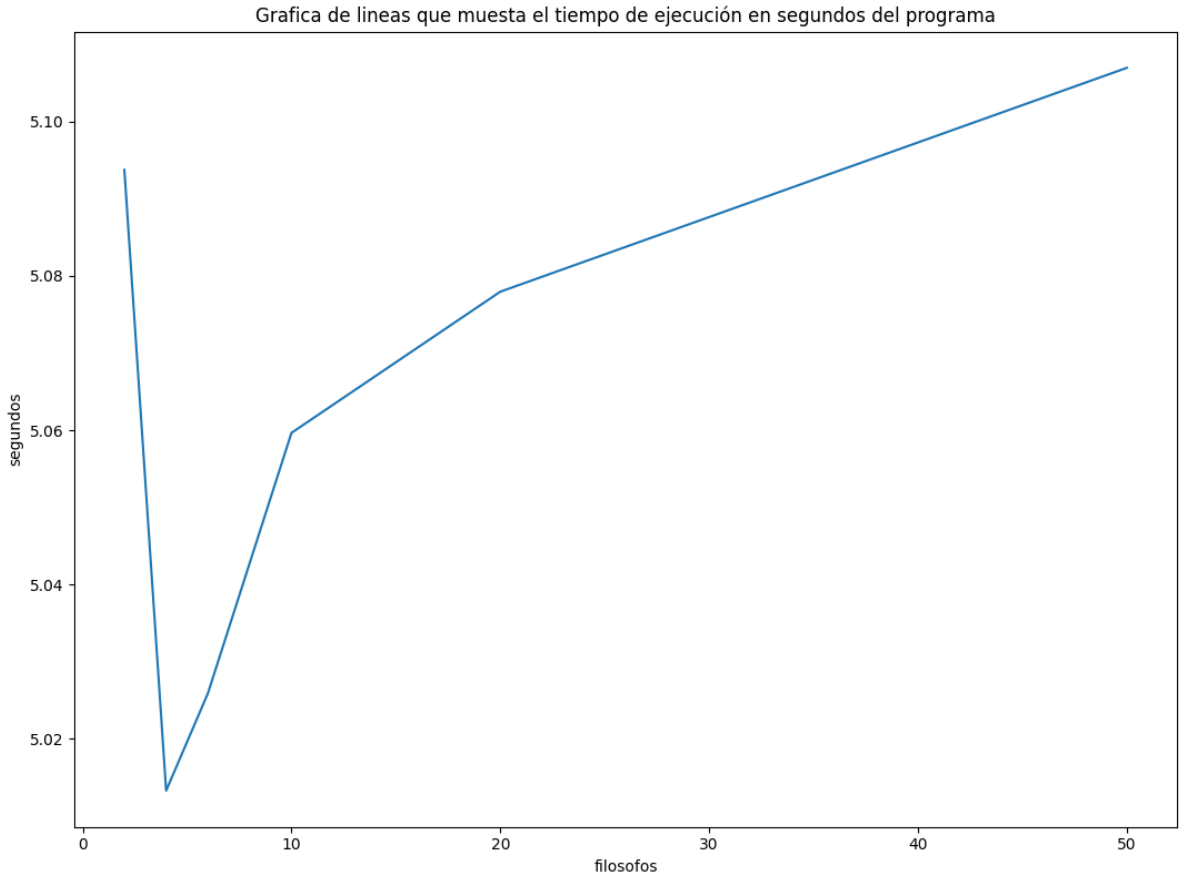


	num_filosofos	num_tenedores	tiempo	num_comidas	num_pensamientos
0	50	50	5.106986	100	100
1	20	20	5.077967	40	40
2	10	10	5.059663	20	20
3	6	6	5.025906	12	12
4	4	4	5.013288	8	8
5	2	2	5.093759	4	4

Como podemos observar a primera vista, para esta primera prueba se realizaron varias corridas con números de filósofos iguales a 2,4,6,10,20,50. Obteniendo tiempos de ejecución casi iguales aun cuando el número de filósofos aumenta de manera progresiva, algo importante que se debe destacar es que en todas estas corridas el número de filósofos es par, resultando de igual manera que el número de tenedores tambien sea par

Resultados y análisis:

A primera instancia se puede pensar que si se aumenta el número de filósofos el tiempo de ejecución aumentaría de manera exponencial pero al evaluar los datos podemos constatar que esto no es del todo así.



La siguiente gráfica fue generada con la librería matplotlib de python utilizando los datos obtenidos en las corridas.

Si bien el tiempo aumenta a medida que el número de filósofos lo hace, este aumento no es muy significativo y es algo que varía también dependiendo del hardware.

¿Por qué el tiempo de ejecución no varía mucho a pesar de que el número de filósofos está aumentando en grandes cantidades?

Teniendo en cuenta que cada filósofo representa un subproceso y que este solo puede ejecutar la acción comer cuando ambos tenedores (recursos) a su lado están disponibles, podemos concluir que ya que el número de palillos es igual al número de filósofos en el mejor de los casos al menos la mitad de estos filósofos estaran comiendo y la otra mitad pensando y ya que el tiempo para pensar y comer siempre es constante (1 segundo), entonces el tiempo en el que los filósofos terminaran de comer una misma cantidad de veces será en la mayoría de los casos muy parecida..

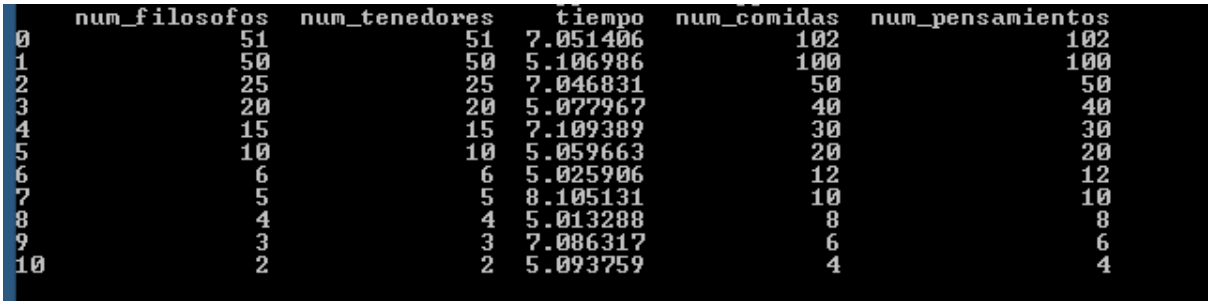
Nota: Posteriormente se realizó otra prueba incluyendo ahora un **número de filósofos impares**, obteniendo así datos que nos muestran un comportamiento distinto del algoritmo.

Prueba 2(Aumentar número de filósofos, números impares de filósofos):

Condiciones:

- El tiempo que tardan los filósofos en comer o pensar es de 1 segundo para todos (posteriormente se realizará una prueba con un tiempo aleatorio).
- Todos los filósofos deben comer y pensar la misma cantidad de veces (en este caso se decidió que el número de veces sería 2), de este modo nos aseguramos que ningún filósofo se apropie de todos los recursos.

Datos:

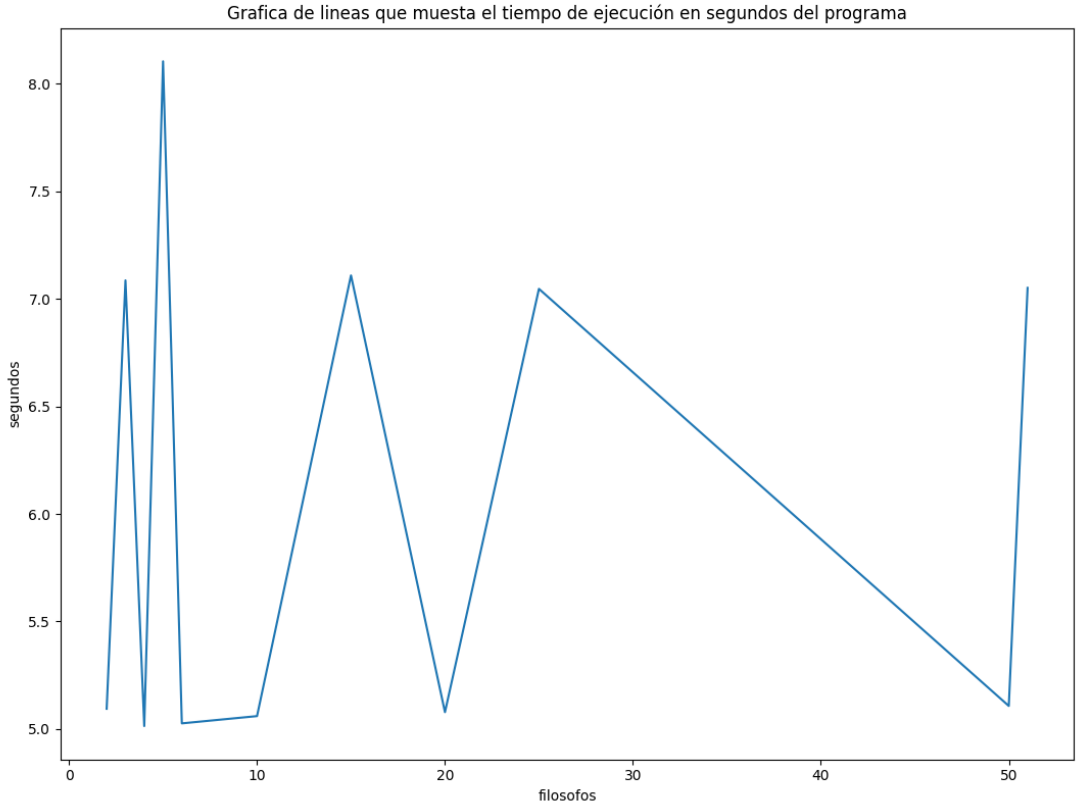


	num_filosofos	num_tenedores	tiempo	num_comidas	num_pensamientos
0	51	51	7.051406	102	102
1	50	50	5.106986	100	100
2	25	25	7.046831	50	50
3	20	20	5.077967	40	40
4	15	15	7.109389	30	30
5	10	10	5.059663	20	20
6	6	6	5.025906	12	12
7	5	5	8.105131	10	10
8	4	4	5.013288	8	8
9	3	3	7.086317	6	6
10	2	2	5.093759	4	4

Como podemos observar los resultados nos muestran algo distinto a lo que concluimos en la primera prueba, ya que al agregar un número impar de filósofos se nos presenta entonces una situación en la cual en estos casos el tiempo de procesamiento del programa es mucho mayor.

Resultados y análisis:

Es gracias a esta segunda prueba que logramos llegar a conclusiones distintas, esto se debe a que al realizar varias corridas con números de filosos impares como 3,5,15,25 y 51 y compararlos con los datos obtenidos con números pares podemos observar que hay una enorme diferencia entre estos resultados en cuanto a tiempo de ejecución se refiere. Dicha diferencia es más evidente en la siguiente gráfica.



La siguiente gráfica fue generada con la librería matplotlib de python utilizando los datos obtenidos en las corridas.

En esta gráfica resalta a simple vista la existencia de picos en cuanto a tiempo de ejecución se refiere, pero ¿Donde ocurren esos picos?, pues estos picos ocurren cuando el número de filósofos es impar. ¿Por qué razón ocurren? Una respuesta simple de la razón por la cual ocurren es debido a que estamos agregando un subproceso extra (filósofo) que necesita el doble de recursos(tenedor) de los que aporta, sumandole a esto que el algoritmo está corriendo bajo la premisa de que cada filósofo debe comer y pensar la misma cantidad de veces para no acaparar los recursos, resultando en operaciones donde no se está sacando el máximo provecho al algoritmo.Por ejemplo:

Partiendo del hecho que el número de filósofos y tenedores es igual y que **n** representa el número de tenedores disponibles, tenemos que.

	número máximo de filósofos que pueden comer a la vez	número de filósofos en espera (pensando)
Cuando n es par	n/2	n/2
Cuando n es impar	(n-1)/2	(n/2)+1

Es evidente entonces que si el número de palillos es impar la cantidad de filósofos que quedan en espera (pensando) es mayor a la de filósofos siendo atendido, sumado al hecho de que el algoritmo intenta equilibrar el uso de los recursos para que los filósofos puedan comer de forma igualitaria, se traduce en un número mayor de ejecuciones para igualar la cantidad de veces en las cuales los filósofos comen y piensan.

Prueba 3 (Aumentar número de filósofos a valores gigantescos):

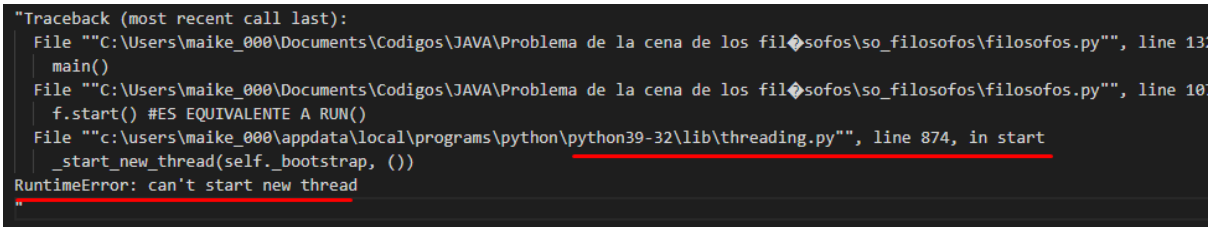
Condiciones:

- El tiempo que tardan los filósofos en comer o pensar es de 1 segundo para todos (posteriormente se realizará una prueba con un tiempo aleatorio).
- Todos los filósofos deben comer y pensar la misma cantidad de veces (en este caso se decidió que el número de veces sería 2), de este modo nos aseguramos que ningún filósofo se apropie de todos los recursos.

Datos:

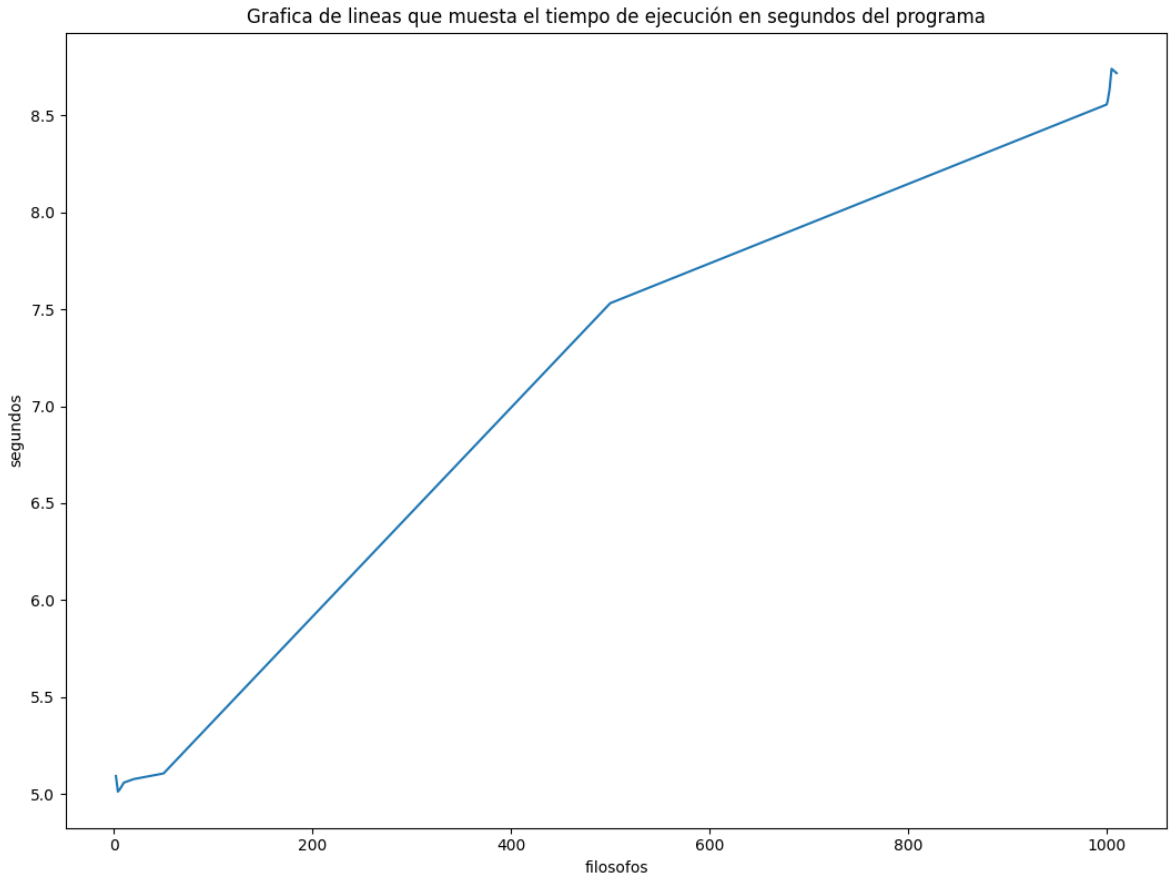
num_filosofos	num_tenedores	tiempo	num_comidas	num_pensamientos
2	2	5.093759	4	4
4	4	5.013288	8	8
6	6	5.025906	12	12
10	10	5.059663	20	20
20	20	5.077967	40	40
50	50	5.106986	100	100
500	500	7.531335	1000	1000
1000	1000	8.555491	2000	2000
1001	1001	8.572276	2002	2002
1003	1003	8.634337	2006	2006
1005	1005	8.739645	2010	2010
1010	1010	8.717503	2020	2020

Errores:



Al aumentar el número de filósofos a 2000 nos encontramos con que python nos arroja una excepción específicamente la librería threading, esto debido a que hemos llegado al límite de thread que podemos crear.
Una curiosidad de la que me di cuenta es que aunque detengas el programa u ocurra una excepción los hilos que ya han sido creados seguirán ejecutándose hasta que finalicen de ejecutar todas sus tareas.

Resultados y análisis:



La siguiente gráfica fue generada con la librería matplotlib de python utilizando los datos obtenidos en las corridas.

Podemos ver que después de 500 thread el tiempo que tarda el algoritmo va creciendo cada vez más.

2.- ¿Qué sucedería si disminuye el número de tenedores ? Plantee una solución

Si seguimos las reglas del enunciado de manera estricta, eliminar uno de los tenedores resultaría en la muerte de dos de los filósofos ya que estos nunca lograran comer, debido a que para comer cada filósofo necesita dos tenedores de manera adyacente y eliminar uno de estos implica que en el lado izquierdo de un filósofo faltara un tenedor y en lado derecho de otro también estará ausente, ya que los filósofos utilizan el mismo comedor.

De este modo a medida que el número de tenedores va disminuyendo, el número de filósofos que pueden comer también lo hace, así dependiendo de la posición del

tenedor eliminado la cantidad de filósofos que puede comer se reduce de manera variable.

Posible solución:

Una posible solución manteniendo las reglas del enunciado es tener una mesa donde se atiendan **n** filósofos, donde **n es el número de tenedores**, mientras que los filósofos restantes esperan para poder comer y a medida que estos terminan de comer van desocupando la mesa para así permitir que los filósofos que estaban esperando coman con los tenedores disponibles.

Implementación:

Factores a tener a en cuenta:

El número de tenedores no puede ser menor a 2.
En caso de tener menos de 2 tenedores todos los filósofos mueren y el tiempo de ejecución es cero porque ninguno puede comer.

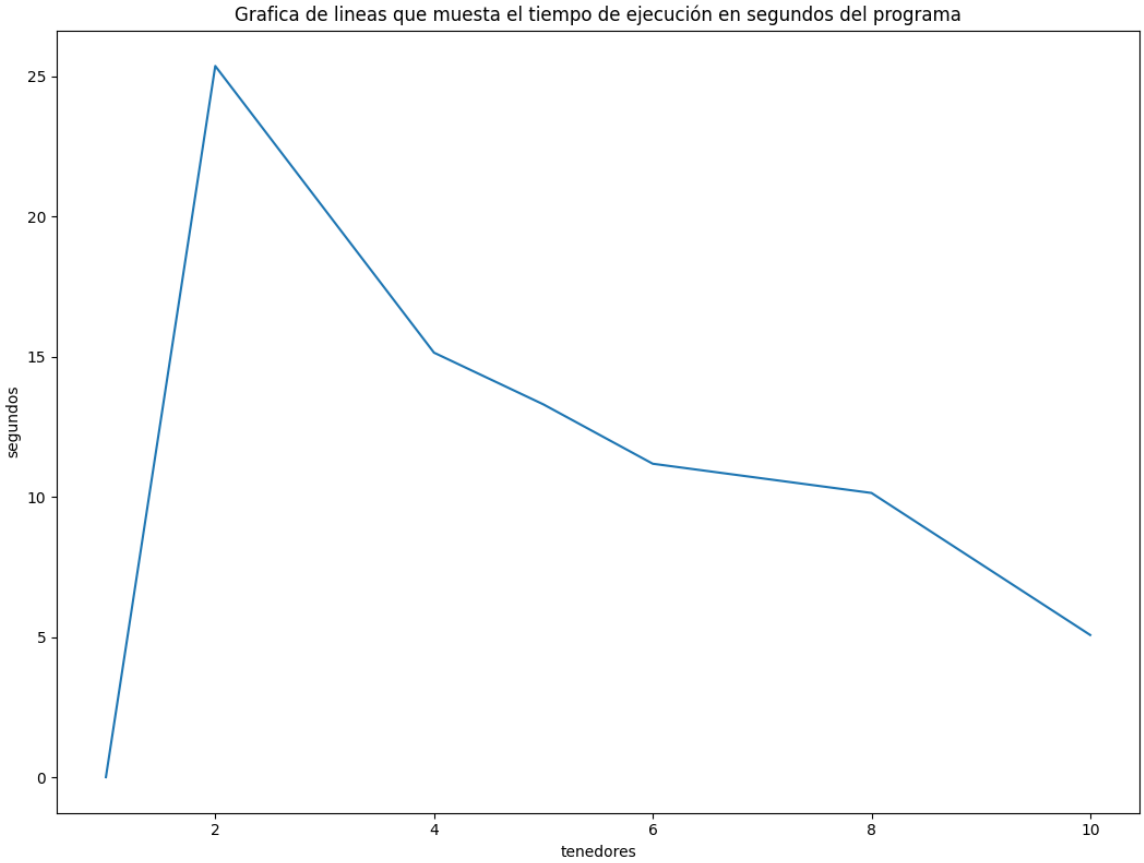
Condiciones:

- El tiempo que tardan los filósofos en comer o pensar es de 1 segundo para todos (posteriormente se realizará una prueba con un tiempo aleatorio).
- Todos los filósofos deben comer y pensar la misma cantidad de veces (en este caso se decidió que el número de veces sería 2), de este modo nos aseguramos que ningún filósofo se apropie de todos los recursos.
- El número de filósofos se colocó en 10 y se fue disminuyendo el número de tenedores disponibles

Datos:

num_filosofos	num_tenedores	tiempo	num_comidas	num_pensamientos
10	1	0.000000	0	0
10	2	25.363811	20	20
10	4	15.140066	20	20
10	5	13.295618	20	20
10	6	11.178545	20	20
10	8	10.135846	20	20
10	10	5.070211	20	20

Resultados y análisis:



La siguiente gráfica fue generada con la librería matplotlib de python utilizando los datos obtenidos en las corridas.

Al estudiar los datos y observar la gráfica es evidente que aunque es plausible ejecutar el algoritmo mientras van disminuyendo los tenedores, el tiempo aumenta exponencialmente a un ritmo acelerado porque ahora en lugar de realizar varios procesos a la vez (la acción de comer del filósofo), este proceso va disminuyendo y cada vez se pueden atender menos filósofos de manera paralela, ocasionando que más filósofos debían esperar e incrementando así el tiempo de ejecución.

3.- ¿Cómo podrían resolverse las situaciones 1 y 2 si ocurren al mismo tiempo ? Plantee una solución.

Nota:

Se debe tener en cuenta lo planteado en las preguntas 1 y 2.

Posible solución:

Una posible solución manteniendo las reglas del enunciado es tener una mesa donde se atiendan **n** filósofos, donde **n es el número de tenedores**, mientras que los filósofos restantes esperan para poder comer y a medida que estos terminan de comer van desocupando la mesa para así permitir que los filósofos que estaban esperando coman con los tenedores disponibles. En pocas palabras la solución es la misma que la respuesta anterior pero los resultados arrojados son totalmente distintos

Implementación:

Factores a tener a en cuenta:

El número de tenedores no puede ser menor a 2.
En caso de tener menos de 2 tenedores todos los filósofos mueren y el tiempo de ejecución es cero porque ninguno puede comer.

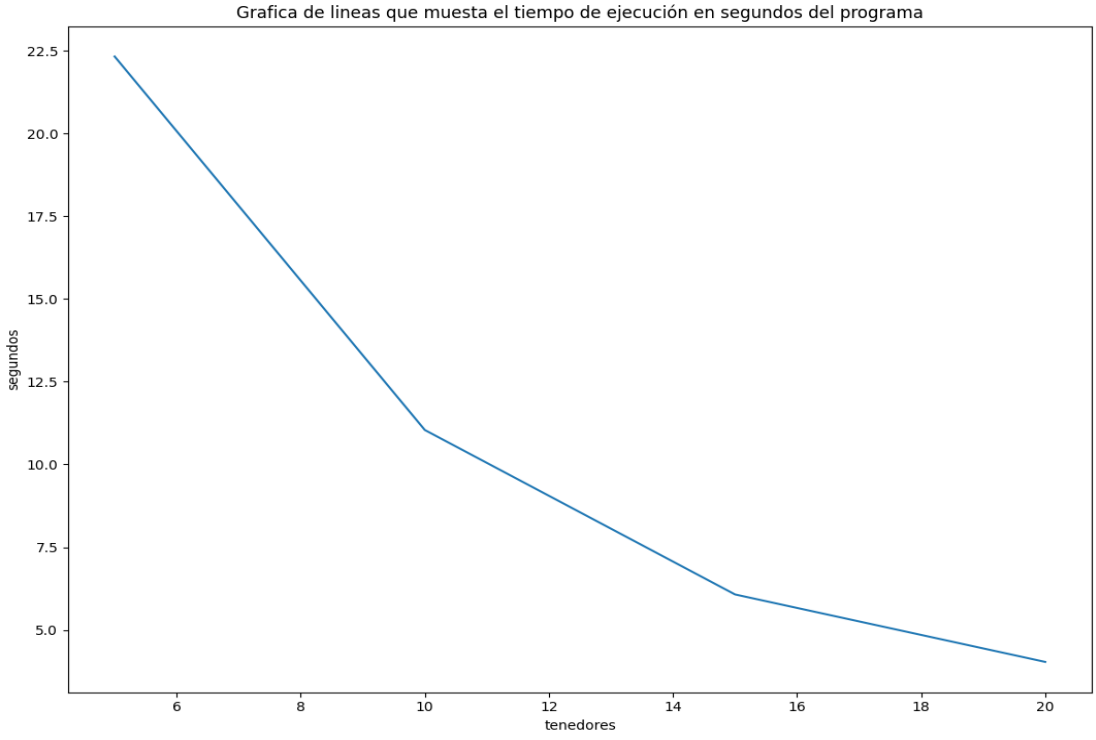
Condiciones:

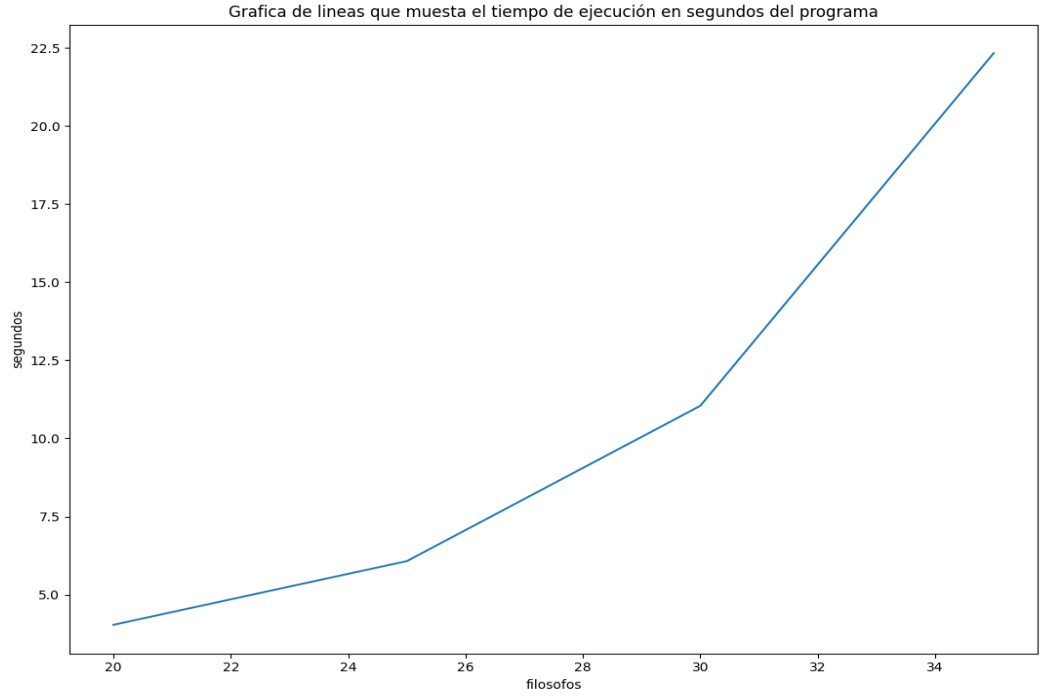
- El tiempo que tardan los filósofos en comer o pensar es de 1 segundo para todos (posteriormente se realizará una prueba con un tiempo aleatorio).
- Todos los filósofos deben comer y pensar la misma cantidad de veces (en este caso se decidió que el número de veces sería 2), de este modo nos aseguramos que ningún filósofo se apropie de todos los recursos.
- El número de filósofos se colocó en 20 al igual que el de tenedores y se fue disminuyendo la cantidad de tenedores al mismo tiempo que se aumentaba la cantidad de filósofos de cinco en cinco en cada iteración

Datos:

num_filosofos	num_tenedores	tiempo	num_comidas	num_pensamientos
35	5	22.327914	35	35
30	10	11.040476	30	30
25	15	6.071313	25	25
20	20	4.033305	20	20

Resultados y análisis:





La siguiente gráfica fue generada con la librería matplotlib de python utilizando los datos obtenidos en las corridas.

El programa se ejecutó y en cada corrida se disminuyó en 5 el número de tenedores al mismo tiempo que se aumentaba en 5 el número de filósofos, lo que resultó en un aumento exponencial en el tiempo de ejecución del programa. Concluyendo así de forma evidente que al aumentar el número de filósofos (hilos) y disminuir los tenedores (recursos) el tiempo aumenta de manera exponencial.

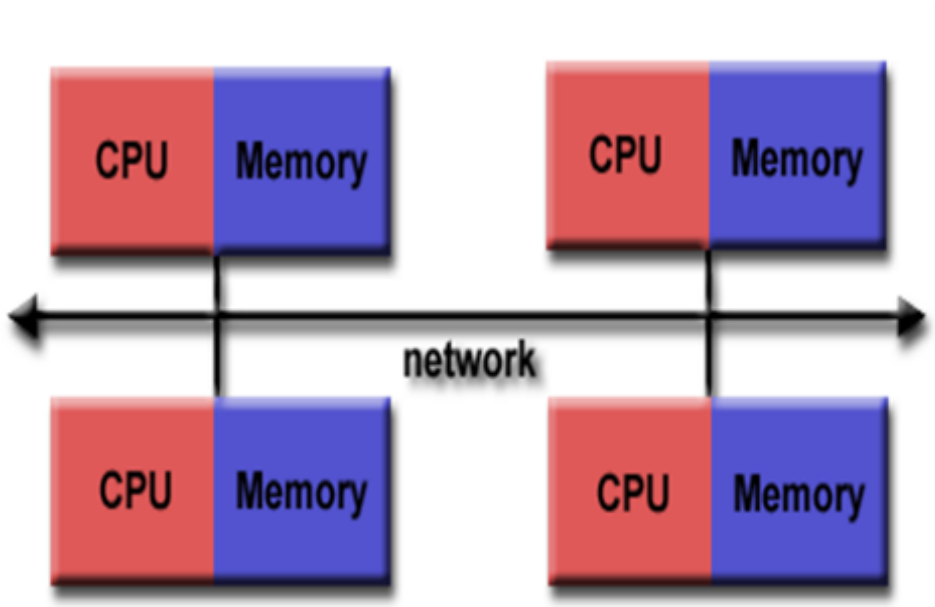
4- La pregunta 4 se contestó al contestar la pregunta 1 y 2 implementando el código del inicio

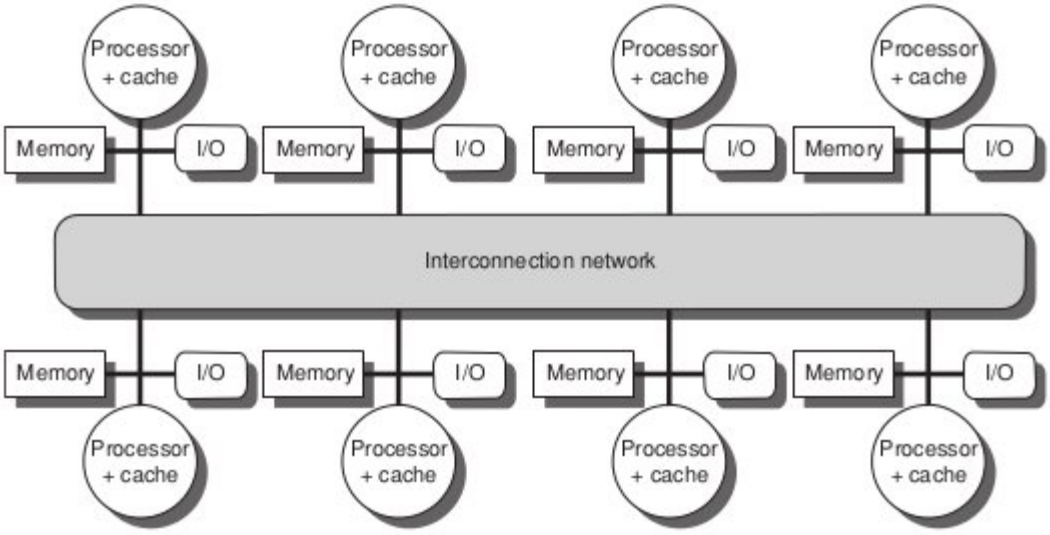
5.- **Escriba un algoritmo (puede ser en pseudocoigo) que permita resolver el siguiente problema:**

Un matemático requiere resolver fórmulas que consumen mucho tiempo de procesamiento, para reducir el tiempo de espera se le ocurre, dividir el problema en partes, y usar procesamiento en paralelo para resolver cada parte, Recuerdan divide y vencerás. Para esto requiere recibir las soluciones parciales para poder obtener la solución final. El detalle está en que el número de partes puede superar al número de procesadores disponibles, no dispone de una red local para comunicar entre sí los equipos de cómputo, pero si tiene acceso a Internet, y decidió solicitar a sus amigos (que trabajan en ciudades diferentes cada uno) que les enviaría una app para que la instalaran en sus teléfonos, y de esa forma poder enviar una parte del problema a cada amigo y recibir las soluciones por medio de una app que instalo en su teléfono, en resumen en lugar de PC se usan los Smartphone de él y sus amigos. Anexe diagramas, código, presentación, y todo lo que considere necesario para mostrar la solución del problema. Observe que deben escribir los algoritmos para el Matemático y el que usarán sus amigos. Tomen en cuenta que algún proceso puede monopolizar los recursos, y además otros problemas que se pueden presentar durante la ejecución.

Solución:

Para resolver este problema utilizamos el paradigma de programación de Paso de Mensajes en una arquitectura de memorias distribuidas. Esta arquitectura se basa en múltiples procesadores que cuentan con su propia memoria física privada, donde las tareas pueden operar solo con información local y necesitan de la comunicación para obtener información remota, a través también de procesadores remotos.

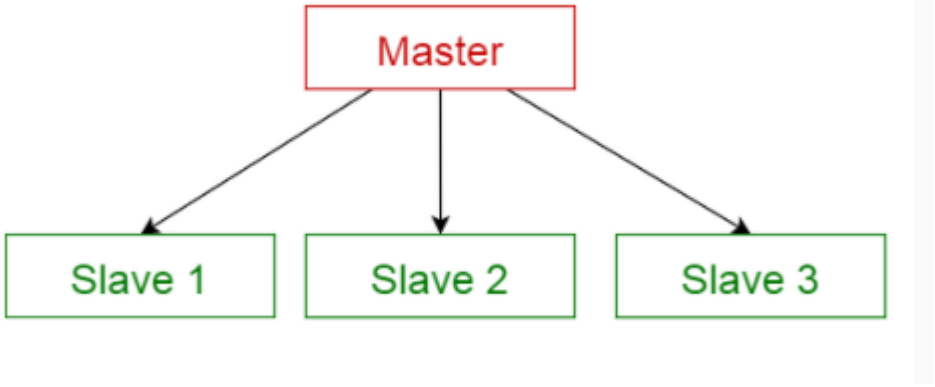




Img 1 y Img 2: Arquitectura de Memorias Distribuidas

La función del algoritmo a plantear es utilizar el envío explícito de mensajes (mediante una red de interconexión) tanto para intercambiar datos entre tareas que se ejecutan en diferentes procesos como también para sincronizarse. Siendo este el caso de un paralelismo de tareas.

Por lo tanto, cada proceso ejecuta una o más tareas, donde la dependencia de estos datos se garantiza mediante el envío de mensajes entre una tarea y otra. Así bien, para realizar ese análisis se utilizará el esquema de “maestro” y “esclavo “.



Img 1: Patrón de diseño “Maestro” y “Esclavo”

El proceso Maestro auto gestiona un gran cúmulo de tareas, el cual se encarga de repartir las tareas a los procesos esclavos a medida que van terminando las tareas asignadas anteriormente.

Pseudocodigo

```
import lib

def funtion_multiprocessing(argumento1, argumento2)

{

    init.connection()

    MPI_Init(&argc, &argv); // Inicializamos los procesos

    MPI_Comm_size(MPI_COMM_WORLD, &size); // Obtenemos el numero total
de procesos

    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Obtenemos el valor de
nuestro identificador
```

```
    //Imprime en consola el nucleo que ejecuto el proceso

     printf("soy el core nro. %d de %d\n", rank);

     MPI_Barrier(MPI_COMM_WORLD);

    start = MPI_Wtime();//ya seleccionados los procesos se inicializa

    //el proceso principal asigna las tareas

    if  MPI.COMM_WORLD.rank  !=  0 :

        MPI.COMM_WORLD.Send (argument1)

     else :

        comm.Bcast("mensaje")//Difunde un mensaje a todos los otro
procesos del grupo.



     /* hacer los calculos de los demás procesos*/

      local_int = funtion_cal(value.x);//operaciones de calculo

      finish = MPI_Wtime();

      MPI_Reduce(&argument1);

      /* Suma los resultado obtenidos */

      MPI_Reduce(&local_int, &total_int, 1, armugemt);

  if (rank == 0) {

    printf("El valor obtenido de los procesos es:" total_int);

  }

  /* Cerrar MPI */

  MPI_Finalize();



  return 0;

}
```

```
def connection()

{

from core import connect

nection = connect('remote.systeem.com', port=1234)

nection ('55774 : remote.systeem.com:1234 remote.systeem.com')

nection._lport

55774

nection._rport

1234

}
```