Trabajo Práctico III de Programación Concurrente

Juan Pablo Saucedo, Sebastián Andrés Mas Casariego, Tomás Martin, Nehemias Mercau.
Grupo: Los concurrentes.
Profesores: Dr. Ing. Orlando Micolini, Ing. Luis Ventre, Ing. Mauricio Ludemann.
Facultad de Ciencias Exactas, Física y Naturales, Universidad Nacional de Córdoba.

***Resumen*— En el siguiente documento se desarrollan los pasos para implementar un simulador de un procesador con dos núcleos, utilizando una Red de Petri que representa a un procesador mono núcleo. Se responde un interrogante importante para la implementación, el cual es la cantidad de hilos o threads que se deben utilizar. También se probarán diferentes parámetros para observar su comportamiento.**

I. INTRODUCCIÓN

Los problemas de exclusión mutua y sincronización representan un gran aumento en los tiempos de ejecución de programas. Existen diversos métodos para implementar la sincronización al acceso de recursos compartidos. Los más conocidos y desarrollados son pipes, semáforos y monitores. Estas técnicas son implementadas por software, mediante intervención del sistema operativo para hacer la asignación de recursos, ocasionando así el aumento de los tiempos de ejecución.

La herramienta utilizada para el modelado y resolución de problemas de concurrencia es el formalismo de Redes de Petri, pudiendo con esta modelar y resolver las técnicas mencionadas en el párrafo anterior.

En este trabajo se desarrollará un monitor de concurrencia extendiendo la Red de Petri de la Fig. 1, la cual representa el funcionamiento de un procesador mono núcleo, a una red que represente un procesador de dos núcleos. A esta se la utilizará en la implementación del software del simulador en lenguaje Java.

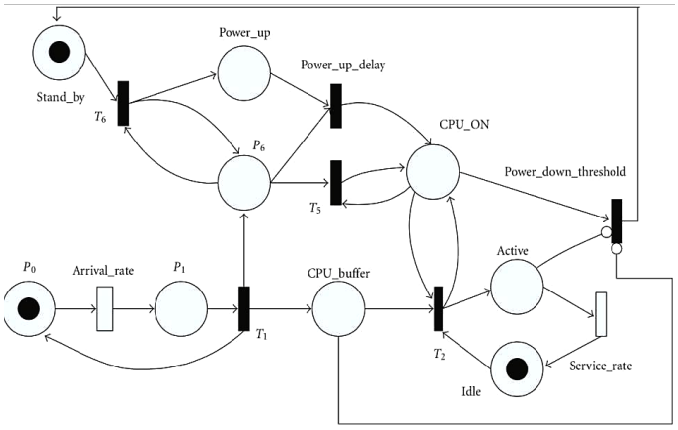


Fig. 1 Red de Petri de un procesador mono núcleo.

El trabajo se encuentra dividido en las siguientes secciones: Red de Petri, Implementación, Pruebas y Conclusión.

Dentro de la sección Red de Petri se explican las propiedades de la Red de Petri extendida. Estas son necesarias para la comprensión de las decisiones tomadas en la Implementación.

En la sección Implementación se desarrolla cómo se logró implementar este simulador y los criterios tomados para la implementación.

En Pruebas se testea el sistema con diferentes parámetros.

Por último se realiza una conclusión.

II. RED DE PETRI

En la Fig. 2 se encuentra la Red de Petri extendida a dos núcleos.

Para el modelado de la red se utilizó la herramienta PIPE.

A. Funcionamiento

Para comprender que sucede en la RdP extendida, se enumeran los pasos para la ejecución de una tarea por parte del procesador:

- La transición temporizada Arrival Rate se encuentra sensibilizada y dentro del rango de tiempo permitido por lo tanto se dispara. Si las transiciones temporizadas comienzan a estar sensibilizada en el instante t, y si continúan sensibilizadas, el disparo de la transición se producirá no antes del instante t +α y no más tarde del instante t + β . Para esta RdP se toma un beta infinito, por lo que es imposible que una transición supere el mismo. Esto lleva a solo tener que esperar un tiempo alfa desde que se sensibilizan.

- La plaza P6 obtiene el token sensibilizando tanto la transición T4 como la T10, estas alimentan al buffer del core 1 y del core 2, respectivamente. Aquí se produce un conflicto el cual se resolverá por una política que se detalla en la implementación.

- Suponiendo que la política se decidió por el core 1, se dispara T4 y se otorga un token al buffer de este. También, un token pasa a la plaza inicial para seguir generando tareas y otro es utilizado para encender el core en caso de estar apagado. En caso de que esté prendido se dispara el garbage collector (T2) para eliminar este token.

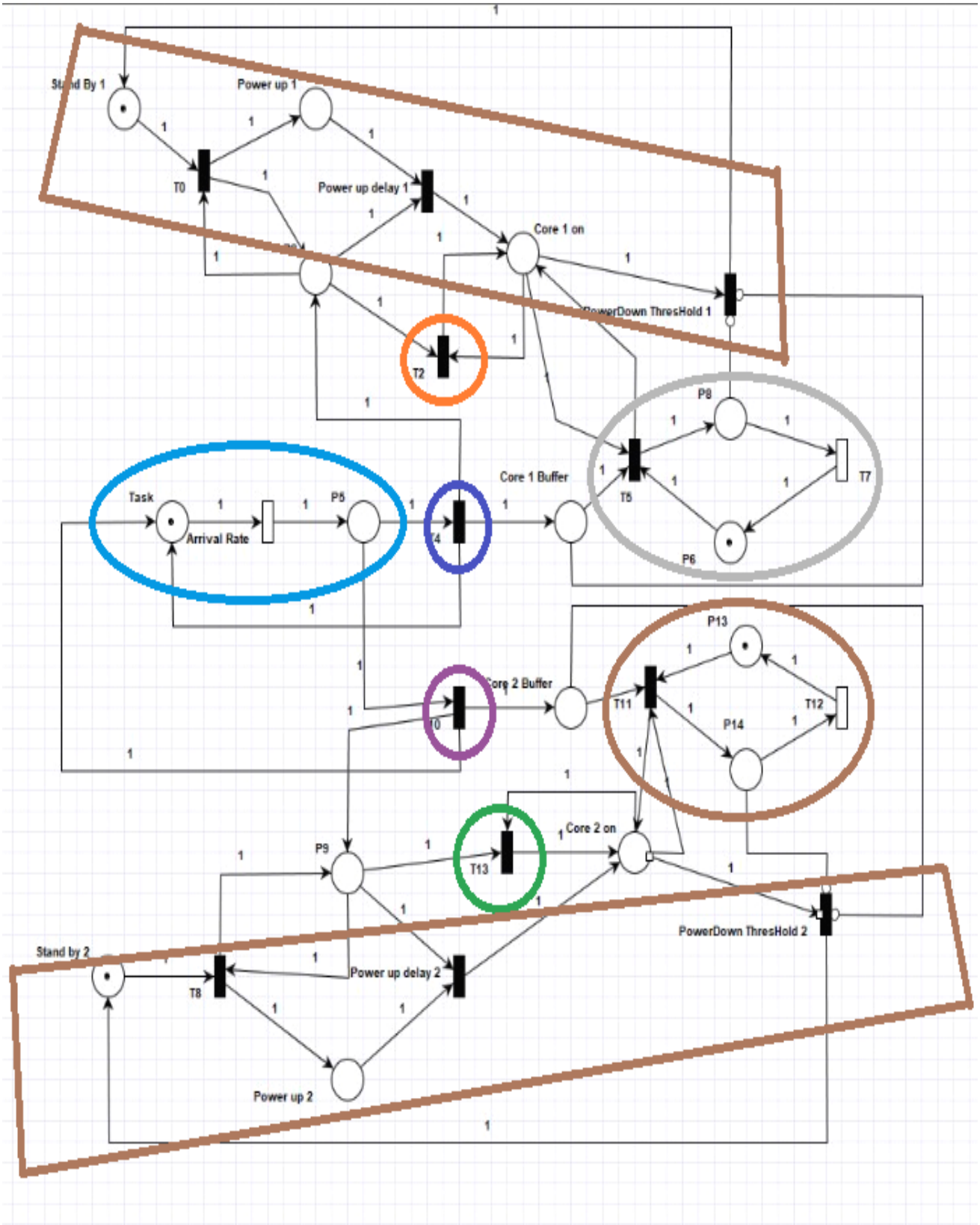


Fig. 2 Red de Petri extendida.

- Para encender el core, primero se dispara la transición T0. Esta se encuentra sensibilizada ya que la plaza Stand By 1 tiene un token y a la otra plaza ya se le otorgó uno cuando se disparó T4. Se puede apreciar la presencia de un autoloop ya que de T0 se tiene un arco que entra y sale conectando a la misma plaza.
  Al dispararse T4 se sensibiliza la transición Power up delay 1, la cual le confiere un token a la plaza Core 1 on cuando se dispara.

- Mientras Core 1 on tenga una marca, se ejecutarán las tareas del buffer disparándose la transición T5 y alimentado la plaza P8. Esta plaza hace referencia al procesamiento de esta tarea que tardará un tiempo alfa indicado por la transición T7.

- Cuando no haya más tareas en el buffer ni procesándose y el core este prendido se sensibilizará la transición PowerDown ThresHold 1 que apagará al núcleo. Esta transición está conectada a las plazas Core 1 Buffer y P8 con arcos inhibidores, lo que significa que para que la transición esté sensibilizada estas plazas deben estar vacías.

B. Propiedades generales de la Red de Petri extendida

Consultado el State Space Analysis de la red en PIPE se obtiene:

Petri net state space analysis results

Bounded	false
Safe	false
Deadlock	false

Fig. 3 State Space Analysis de la RdP.

En la Fig. 3 se observa que la Red de Petri es limitada, no segura, y está libre de interbloqueo:

- Es limitada porque la cantidad de tokens de las plazas alcanzables no supera un número natural k.
- No es segura, ya que para cada marca alcanzable, cada lugar no contiene necesariamente cero o un token.
- Está libre de interbloqueo debido a que contiene transiciones que se pueden disparar para no llegar a un punto muerto. Si está libre de interbloqueo, es una red viva.

Luego se consultó la clasificación de la red obteniendo:

Petri net classification results

State Machine	false
Marked Graph	false
Free Choice Net	false
Extended Free Choice Net	false
Simple Net	false
Extended Simple Net	false

Fig. 4 Clasificación de la RdP.

Como se puede ver en la Fig. 4 la Red de Petri:

- No es una máquina de estado ya que las transiciones tienen más de un lugar de entrada y lugar de salida.
- No es un Gráfico de Marcado porque las plazas tienen más de una transición de salida y de entrada
- No es de Libre elección ya que en un conflicto las transiciones involucradas tienen más de un una plaza de entrada. Como no se cumple que es simple tampoco será de libre elección.
- No es de Libre elección extendida porque en un conflicto las transiciones no tienen el mismo conjunto de plazas como entrada.
- No es Simple debido a que una transición involucrada se ve afectada por más de un conflicto.
- No es Simple extendida ya que una transición se ve afectada por más de un conjunto de entradas. Si no ocurre que es simple tampoco será simple extendida.

B. T-Invariants y P-Invariants

Se realizó un análisis de los invariantes de transición y los invariantes de plaza o marcado.

Los invariantes P (invariantes de plaza) son el conjunto de lugares en los cuales el conteo ponderado de marcas es constante durante el disparo de las transiciones.

De la Red de Petri extendida se obtiene mediante un Invariant Analysis de PIPE las siguientes ecuaciones:

P-Invariant equations:

M(P13) + M(P14) = 1
M(P6) + M(P8) = 1
M(Core1 on) + M(Power up1) + M(Stand By1) = 1
M(Core2 on) + M(Power up2) + M(Stand By2) = 1
M(P5) + M(Task) = 1

Estas indican que el marcado de las plazas involucradas en cada ecuación es igual a 1. Es decir que, si una tiene una marca o token, la otra no debe contener ninguno.

Los invariantes T (invariantes de transición) indican el número de veces que cada transición debe dispararse para, partiendo de un marcado inicial, llegar nuevamente a este marcado.

Se hizo un mapeo de las transiciones a índices del array de transiciones obtenida con el Invariant Analysis de PIPE:

0. Arrival Rate
1. PowerDown ThresHold 1
2. PowerDown ThresHold 2
3. Power up delay 1
4. Power up delay 2
5. T0
6. T10
7. T11
8. T12
9. T13
10. T2
11. T4
12. T5
13. T7
14. T8

Los invariantes de transición de la RdP extendida son los siguientes:

1. Arrival Rate, PowerDown ThresHold 1, Power up delay 1, T0, T4, T5, T7. Mapeado a índices del array de transiciones: **0, 1, 3, 5, 11, 12, 13**.
2. Arrival Rate, PowerDown ThresHold 2, Power up delay 2, T10, T11, T12, T8. Mapeado a índices del array de transiciones: **0, 2, 4, 6, 7, 8, 14**.
3. Arrival Rate, T10, T11, T12, T13. Mapeado al array de transiciones: **0, 6, 7, 8, 9**.
4. Arrival Rate, T2, T4, T5, T7. Mapeado al array de transiciones: **0, 10, 11, 12, 13**.

El invariante 1 se ejecuta siempre en la misma secuencia: **0 - 11 - 5 - 3 - 12 - 13 - 1**.

El invariante 2 también se ejecuta siempre en la misma secuencia: **0 - 6 - 14 - 4 - 7 - 8 - 2.**

El invariante 3 trae consigo indeterminismo en el orden de ejecución. Comienza siempre con la secuencia **0 - 6**. La transición **8** se debe ejecutar siempre después de la **7**. Por otro lado, la transición **9** se puede ejecutar en cualquier momento después de la secuencia **0 - 6**, es decir, antes de las transiciones **7** y **8**, entre ellas o después.

Siendo las secuencias posibles:

0 - 6 - 7 - 8 - **9**
0 - 6 - 7 - **9** - 8
0 - 6 - **9** - 7 - 8

El invariante 4, análogo al invariante 3, también tiene indeterminismo en el orden de ejecución. Comienza siempre con la secuencia **0 - 11**. Luego, la transición **13** se debe ejecutar siempre después de la **12**. La transición **10**, que viene siempre después de la secuencia **0 - 11**, se puede ejecutar antes de las transiciones **12** y **13**, entre ellas o después.

Obteniendo las secuencias posibles:

0 - 11 - 12 - 13 - **10**
0 - 11 - 12 - **10** - 13
0 - 11 - **10** - 12 - 13

III. IMPLEMENTACIÓN

Para realizar la implementación del software lo primero que se analizó fue la cantidad de threads que se iban a utilizar para la ejecución de tareas de forma concurrente.

A. Cantidad de Threads

Para determinar la cantidad de threads, se usó como base a los invariantes de marcado y de transición de la Red de Petri: los invariantes de marcado representan un recurso único, por lo que será deseable que lo maneje un único thread; además, con los invariantes de transición y de marcado, se puede identificar cuales transiciones corresponden a un proceso secuencial. Es decir, cuales transiciones sensibilizadas deben dispararse después de que sucedió alguna.

Si se quiere optimizar el tiempo de ejecución, es deseable realizar tareas de forma concurrente, por lo que cada invariante (proceso secuencial) será asignado a un thread distinto. De esta forma, se logra ejecutar el programa con un alto grado de concurrencia.

Por ejemplo, si se observa el invariante de las plazas que corresponden a la llegada de tareas: es deseable que la llegada de tareas, sea independiente del resto de la RdP, y se pueda hacer independientemente de si hay algún core trabajando, o de si los cores están encendidos o apagados.

Mediante el análisis anterior, se decidió usar un thread para cada invariante de plaza de la RdP. Esto nos da como resultado 5 threads, uno para el encendido, otro para el apagado, dos para el procesamiento del core 1 y del 2, y otro para la llegada de las tareas.

Por otro lado, necesitamos otros 2 threads que corren las transiciones T2 y T13 que son los garbage collector, ya que estas transiciones son totalmente independientes de las otras tareas. Además se agregan otros 2 threads, uno para cada transición que alimenta cada buffer.

De esta forma obtenemos en total **9** threads para correr el programa. En la Fig. 2 se resaltan las transiciones de las cuales se ocupará cada thread.

B. Implementación del Software

Para la implementación del software se crearon las siguientes clases:

- Sistema: Esta clase se encarga de instanciar todos los threads necesarios para la ejecución del programa, de monitorear los tiempos de ejecución final del programa, y la cantidad de productos que procesa cada core.

- Monitor: Esta clase realiza el control de la concurrencia. Cada vez que un thread quiere ejecutar una transición, entra al monitor el cual en base a la información de la Red de Petri y la política, decide si deja ejecutar la transición o manda al thread a una cola de espera. Cuando un thread termina de ejecutar su transición, el monitor se encarga(en base a la política) de despertar a un thread de los que ya están esperando para que ejecute su transición, De esta forma, el monitor le da prioridad a los threads que están esperando dentro del mismo.

- Política: Contiene la información necesaria para resolver los conflictos que se producen en la RdP. Como esta tiene un buffer para cada core, es trabajo de la política que la repartición de productos se haga de forma equitativa. La política elegida se basa en otorgar siempre el producto al buffer que tenga menor cantidad de productos al momento de la decisión. Esto se ve reflejado en la ejecución de tal forma que el core más rápido siempre va a procesar más tareas que el otro.

Además, la política permite establecer prioridades sobre las transiciones.

- RedDePetri: Se encarga de toda la lógica relacionada al disparo de las transiciones, y de decidir si una transición está sensibilizada o no. Para tener toda la información necesaria y decidir si todos estos tipos de transiciones están sensibilizadas, es necesario que esta clase contenga la información de la matriz de incidencia positiva y negativa (para los autoloop), la matriz de inhibición (para las transiciones inhibidoras) y los alfa y beta correspondientes a cada transición temporizada. Como se mencionó anteriormente, se toma un beta infinito, por lo tanto una transición nunca lo superará.

- Log: Es responsable de guardar en un archivo txt todas las transiciones en orden que se van ejecutando en el programa. Se encarga de verificar los invariantes de plaza cada vez que se dispara una transición comprobando que la suma de las marcas de estos invariantes sea correcta. También, al finalizar la ejecución del programa, verifica que el marcado final de la RdP sea el correcto. Originalmente, esta clase se encargaba de verificar los invariantes de transición, pero en la última versión del programa, se realizó un script de python que levanta las transiciones del log del programa y verifica los invariantes de transición.

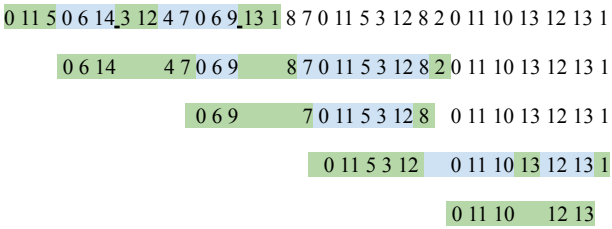
Además de las anteriores, existen clases que contienen el código de ejecución para las transiciones. Estas clases son instanciadas por Sistema, que luego son entregadas a un thread para que ejecute su código.

Para verificar que se cumplan los invariantes T, como se dijo anteriormente, en el log de ejecución de la aplicación se guardan los índices de las transiciones disparadas en orden. Esta será la entrada del algoritmo, que se desarrollará a continuación, para comprobar el cumplimiento de estos invariantes.

C. Algoritmo para comprobar los invariantes T

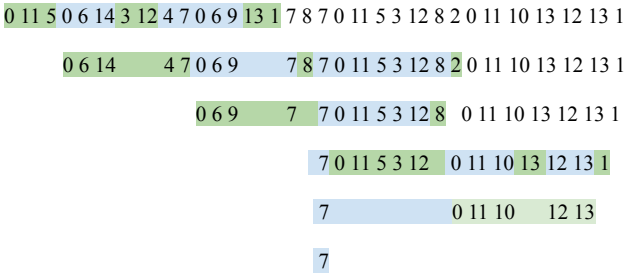
El algoritmo va igualando todas las transiciones obtenidas del Log con una expresión regular. La expresión regular incluye a todos los invariantes. Se matchea con el primer invariante que se detecta, capturando como grupos a todo lo que esté entre dos transiciones que forman parte del invariante. Luego se concatenan estos grupos y se reemplaza el match de la regex con esta concatenación. Esto se repite hasta que no hayan mas matches. Finalmente, se chequea que solo queden espacio en blanco.

Ejemplo de ejecución para el siguiente log:



Al quedar el log vacío, se cumplen los invariantes T.

Ejemplo en el que no se cumplen los invariantes T, se trata del log anterior con dos transiciones 7 que se ejecutan sin que se ejecute la 8 en el medio:



Al no quedar vacío, comprobamos que no se cumplieron los invariantes de transición.

La expresión regular encargada de llevar a cabo lo anterior es la siguiente:

```
\b0\b(.*?)(?:\b11\b(.*?)(?:\b10\b(.*?)\b12\b(.*?)\b13\b|\b12\b(.*?)(?:\b10\b(.*?)\b13\b|\b13\b(.*?)\b10\b)|\b5\b(.*?)\b3\b(.*?)\b12\b(.*?)\b13\b(.*?)\b1\b)|\b6\b(.*?)(?:\b9\b(.*?)\b7\b(.*?)\b8\b|\b7\b(.*?)(?:\b9\b(.*?)\b8\b|\b8\b(.*?)\b9\b)|\b14\b(.*?)\b4\b(.*?)\b7\b(.*?)\b8\b(.*?)\b2\b))
```

Para comprender la expresión regular se especifica que:

- \b hace referencia a separadores de palabras, como espacios en blanco o principios de línea.
- (.*?) matchea cualquier caracter 0 o más veces hasta que se matchee lo que le sigue.
- Las expresiones encerradas entre paréntesis son grupos, si tienen una ?: al comienzo, significa que ese grupo no se captura. Los grupos que se capturan son los que se concatenan luego para hacer el reemplazo.
- | es un OR, matchea lo de la izquierda del | o lo de la derecha.

Para crearla, visualizarla y entenderla mejor, se usó la web Regexper, que devuelve el siguiente gráfico (ver en el enlace).

En el gráfico se ve que se chequean todas las combinaciones posibles de ejecución de los invariantes T.

Este enfoque no escala ya que si tuviéramos más de una transición como los garbage collector (es decir, que se pueden ejecutar en distintas secuencias), la cantidad de combinaciones a chequear aumentan exponencialmente.

En cualquier caso (éxito o fracaso) y tanto para los P-Invariants como para los T-Invariants, se escribe un mensaje de información al final de los registros de ejecución de programa.

D. Testing

Se realizaron Unit Tests e Integration Tests para las clases más relevantes y más susceptibles a error del programa, estas son las clases relacionadas a la Política, la Red de Petri y el Monitor.

Los Unit Test se crearon para probar el correcto funcionamiento de los métodos de todas estas clases, por lo que existe uno por cada método relevante (todo método que no sea un getter o un setter)

Los Integration Test están relacionados al buen funcionamiento en conjunto de los métodos. Se hizo un Integration Test para el método principal del Monitor: este test verifica que la Política, la Red de Petri y el Monitor trabajen en conjunto de la forma que se espera, cambiando parámetros de cada uno de estos.

Es necesario resaltar, que el Integration Test verifica distintos tiempos relacionados a la llegada de los threads al Monitor y cambios en los estados de los threads, por lo que podría fallar para distintas computadoras. Sin embargo esto no está relacionado al funcionamiento en sí del programa, si no a la rapidez con la que van a llegar los threads al Monitor o la rapidez con la que terminan de ejecutar su código, que depende enteramente del dispositivo sobre el que está corriendo el programa.

Existe también una prueba de aceptación (System Test) que verifica que el tiempo de ejecución sea el esperado para la cantidad de tareas dada. Se hace una estimación del tiempo mínimo que debe tardar el programa en base a la transición temporizada de mayor alfa y verifica que la cantidad de productos sea correcta.

IV. PRUEBAS

Se probó el sistema con los siguientes parámetros:

α : $arrival\ rate\ [tareas\ /\ ms]$

β_1 : $servicerate\ core\ 1\ [tareas\ /\ ms]$

β_2 : $servicerate\ core\ 2\ [tareas\ /\ ms]$

T_T: $tareas\ totales\ [tareas]$

Y se consiguieron los siguientes resultados:

T_1: $tareas\ procesadas\ por\ core\ 1\ [tareas]$

T_2: $tareas\ procesadas\ por\ core\ 2\ [tareas]$

t_T : $tiempo\ total\ de\ ejecución\ [ms]$

i	α	β_1	β_2	T_T	T_1	T_2	t_T
1	1/10	1/50	1/50	1000	500	500	26240
2	1/10	1/50	1/100	1000	558	442	45165
3	1/10	1/50	1/150	1000	576	424	64450

Analizando los resultados y el sistema en sí, podemos concluir primero, que obviamente el procesamiento de las tareas no puede terminar antes de que se produzcan, y a partir de ese tiempo de arribo se va a tardar lo que tarde el core más lento en procesar las tareas que le quedan en el buffer.

Si el retraso en la llegada de tareas es mayor que el tiempo máximo que se tarda en procesarlas, entonces el tiempo total usado por el sistema va a ser simplemente el producto entre el tiempo de llegada de cada tarea por la cantidad de tareas totales

En caso de que las tareas lleguen más rápido de lo que se las procesa:

$$t_{arribo} = \frac{T_T}{\alpha} \quad (1)$$

Las tareas restantes por procesar al momento en el que termina la producción de tareas viene dada por la siguiente fórmula, son las tareas restantes menos las tareas que fueron procesadas durante la producción:

$$T_{restantes} = T_T - t_{arribo}\left(\beta_1 + \beta_2\right) \quad (2)$$

Estas se dividen equitativamente en cada buffer, por lo que en el momento en que termina la producción de tareas, cada core tiene asignadas $\frac{T_{restantes}}{2}$ tareas en su respectivo buffer.

Por lo que podemos escribir que:

$$t_{TOTAL} = t_{arribo} + \frac{T_{restantes}}{2min\{\beta_1,\beta_2\}} \quad (3)$$

Dado que se termina la ejecución una vez termina el core más lento de procesar sus tareas.

Aplicando a los 3 casos anteriores (3):

$$t_{arribo} = \frac{1000}{0.1} = 10000\,ms$$

Para 1)

$$T_{restantes} = 1000 - 10000\left(\frac{1}{50} + \frac{1}{50}\right) = 600$$

Para 2)

$$T_{restantes} = 1000 - 10000\left(\frac{1}{50} + \frac{1}{100}\right) = 700$$

Para 3)

$$T_{restantes} = 1000 - 10000\left(\frac{1}{50} + \frac{1}{150}\right) = 733$$

Lo que implica que el tiempo total en cada caso es:

Para 1) $t_{TOTAL} = 12500 + \frac{600}{\frac{2}{50}} = 25000\,ms$

Para 2) $t_{TOTAL} = 12500 + \frac{700}{\frac{2}{100}} = 45000\,ms$

Para 3) $t_{TOTAL} = 12500 + \frac{733}{\frac{2}{150}} = 65000\,ms$

En cuanto a la cantidad de tareas que procesa cada core, esto viene dado por la cantidad que procesan durante el tiempo de arribo de las tareas sumado a la mitad de las tareas restantes.

Para 1) $T_1 = 10000\,(\frac{1}{50}) + \frac{600}{2} = 500$

$T_2 = 10000\,(\frac{1}{50}) + \frac{600}{2} = 500$

Para 2) $T_1 = 10000\,(\frac{1}{50}) + \frac{700}{2} = 550$

$T_2 = 10000\,(\frac{1}{100}) + \frac{700}{2} = 450$

Para 3) $T_1 = 10000\,(\frac{1}{50}) + \frac{733}{2} = 567$

$T_2 = 10000\,(\frac{1}{150}) + \frac{733}{2} = 433$

Se observan dos cosas: la primera es que las tareas tienden a agruparse mayormente en el core más rápido, lo cual es lógico ya que en la decisión de a que buffer alimentar, se decide por el buffer con menos tareas. Si un core es más lento, es lógico que acumule más marcas (tareas) en su plaza, y por tanto la política elija más seguido al buffer del core más rápido.

Lo segundo que se puede observar es que mientras más lento sea el core más lento de los dos, más tiempo de ejecución va a necesitar el sistema. A pesar de esto, el core más rápido compensa en cierta medida la lentitud del otro core gracias a la distribución equitativa de las tareas.

Estos resultados son estimativos, y pueden variar ligeramente con respecto al tiempo real de ejecución.

V. CONCLUSIÓN

Inicialmente se contempló la posibilidad de asignar un mismo thread al encendido, apagado y al procesamiento de las tareas en el núcleo. Esto se debe a que el encendido del core y el proceso de las tasks son ambas tareas excluyentes, y por tanto podrían ser ejecutadas secuencialmente por el mismo thread.

Si se implementara el programa como se mencionó anteriormente, no se ganaría nada en tiempo de ejecución. La decisión final de separar las tareas de encendido y proceso en distintos threads se basa en el hecho, de que si le mismo thread tuviera que encender y procesar, se debería mantener en el thread información sobre la Red de Petri, que ayude a determinar cuál invariante debe ejecutarse a continuación. Esto no es deseable, lo mejor es que al thread se le asigne una única tarea, sin darle información sobre el estado del recurso y que el monitor se encargue de demorar al thread hasta que se pueda realizar su tarea.

En una de las primeras versiones del programa se tenía un mismo thread asignado a las transiciones de llegada de tasks y a la alimentación de los buffers. De la misma forma que en el caso anterior, para implementarlo de esta forma es necesario que el thread tenga información sobre qué buffer debería alimentar.

El uso de 9 threads minimiza el tiempo de ejecución de este simulador, minimizando también el uso de recursos del procesador. Sin embargo, siempre que uno de los núcleos sea más lento que el otro, va a generar un cuello de botella que va a ralentizar todo el sistema.