Tecnologías del Sector Financiero

Práctica Lección 1

Latencias

ENTENDIENDO Y MIDIENDO LAS LATENCIAS

PRÁCTICAS

Universidad Carlos III de Madrid

Comunytek
CONSULTING & SOLUTIONS

Contenido

1. Introducción

El proyecto de práctica está escrito en Java con Maven para resolver las dependencias, lo podéis abrir en cualquier IDE de programación Java (Eclipse, IntelliJ, Netbeans) o utilizar la línea de comandos y un editor de texto si lo preferís.

El objetivo de la práctica consiste tanto en resolver los ejercicios como en analizar y comprender los resultados obtenidos.

Podéis encontrar el código de todas las prácticas en GitHub:
https://github.com/cnebrera/MasterUC3Practices

Se necesita tener instalado Java8 para poder trabajar con ellas.

2. Práctica 1: Medición simple de latencias

En esta práctica utilizaremos la librería HdrHistogram para realizar una medición simple de latencias.

El proyecto está escrito en Java con Maven para resolver las dependencias, lo podéis abrir en cualquier IDE de programación Java (Eclipse, IntelliJ, Netbeans) o utilizar la línea de comandos y un editor de texto.

a. Objetivo

Obtener el histograma de latencias en microsegundos de la ejecución de una operación síncrona simple.

Ir al fichero PracticeLatency1.java y buscar el método runCalculations(). El método es muy simple, ejecuta 100 mil veces una operación.

La clase SyncOpSimulRndPark realiza una espera activa de tiempo aleatorio en nanosegundos entre los dos valores dados, en este caso entre 100 nanos y 100 micros.

El objetivo es realizar el histograma sobre los tiempos de ejecución de cada una de las operaciones “executeOp” del código utilizando HdrHistogram.

Entregar el código de la clase con lo que sea necesario para hacer los cálculos y una copia de los resultados.

b. Ayudas

Aunque en teoría el método debería tardar entre 100 nanos y 10 micros, hay otros factores que pueden afectar a la ejecución. Si no tenemos claro el rango que dar al histograma podemos utilizar autoResize y cuando conozcamos los máximos definir un histograma con rangos fijos.

Recuerda que cuando pintamos los resultados de un histograma podemos dar un valor de escalado para transformar estos resultados en la salida.

c. ¿Qué debería obtener y por qué?

Deberíais ver unas latencias hasta el percentil 90 mas o menos que se ajustan con los tiempos de parada que debería tener la llamada del método. Sin embargo a medida que nos acercamos a percentiles por encima del 99 los tiempos se disparan considerablemente.

Hay muchos factores que pueden afectar a la ejecución de una operación: Bloqueos del SO, cambio de contexto de hilos, otros procesos, el gargabe collector en el caso de Java, etc. Cuando entra uno de estos factores los tiempos se pueden disparar. Dado que en general sucede pocas veces, únicamente afecta a los percentiles mas extremos.

Es casi imposible, con excepción de sistemas especiales de tiempo real el tener un Jitter homogéneo en percentiles extremos.

3. Práctica 2: Medición con y sin calentamiento

En esta práctica vamos a ver la diferencia de rendimiento de una operación que siempre hace lo mismo con el sistema “caliente” o “frio”.

a. Objetivo

Obtener la latencia media, mínima, máxima y percentil 99 y 99,9 en nanosegundos de la ejecución de una operación síncrona simple con y sin calentamiento del sistema.

Ir al fichero PracticeLatency2.java y buscar el método runCalculations(). El método es muy simple, ejecuta 200000 mil veces una operación. En este caso la operación siempre hace lo mismo, recorre un mapa, y realiza algunos cálculos sobre los valores.

Al igual que en la práctica 1 debemos medir la latencia de la operación executeOp(). Esta vez queremos comparar los resultados con el sistema “caliente” y “frio”. El sistema frio hace referencia a la primera o primeras ejecuciones de una operación, cuando el sistema lleva un rato en funcionamiento y se han ejecutado muchas operaciones decimos que el sistema esta caliente.

Entregar el código de la clase con lo que sea necesario para hacer los cálculos y una copia de los resultados.

b. Ayudas

No vale ejecutar el sistema varias veces desde el IDE, en cada ejecución se crea una máquina virtual nueva y el sistema comienza en frio.

La forma mas sencilla de hacerlo es ejecutar el método runCalculations() de forma continuada en el main(). El problema de esto es que estaremos instanciando el histograma y el simulador de latencia múltiples veces, lo cual afecta a los resultados.

Intenta instanciar una única vez el Histograma, recuerda que puedes utilizar reset() para reutilizar la misma instancia en cálculos sucesivos. Intenta instanciar también una única vez el simulador de latencias.

c. ¿Qué debería obtener y por qué?

La primera ejecución debería ser la mas lenta y con peores latencias, a media que se ejecuta los valores mejorarán y se estabilizarán.

Esto es debido principalmente a 3 motivos entre otros:

1. El JIT (Just In Time Compiler) de Java optimiza la ejecución del código mas utilizado de la aplicación, pero necesita tiempo y muchas operaciones para entrar en acción. En la primera ejecución puede que no le haya dado tiempo a optimizar.
2. El Turbo Boots: Casi todos los procesadores modernos incrementan la frecuencia de la CPU cuando tiene carga, la primera ejecución aún tendrá parte de las operaciones ejecutando a “menos frecuencia”
3. El Garbage Collector: La clase que simula latencias crea un Mapa grande que consume mucha memoria. Al ejecutar, la primera vez que entra el recolector de basura tiene que mover esa memoria al espacio “Old”, lo cual requiere una pausa larga. En llamadas sucesivas del recolector esto no es necesario.

Como veis en el caso de Java es importante calentar el sistema para obtener un rendimiento óptimo. Hay muchos factores incluso en un código tan simple como este que pueden afectar a la latencia.

4. Práctica 3: Medición de latencias acumuladas y multi-hilo

En esta práctica vamos a hacer un cálculo de latencias acumuladas sobre un sistema con múltiples hilos de ejecución y comparar los resultados con latencias simples.

a. Objetivo

Obtener la distribución por percentiles de latencia sobre una operación para todos los hilos de ejecución. Comparar con la latencia acumulada teniendo en cuenta un ratio esperado máximo de llamadas.

El código está en la clase PracticeLatency3.java. Al comienzo de la clase tenéis 3 constantes, el número de hilos que van a ejecutar, numero de ejecuciones por hilo y el máximo esperado de ejecuciones por segundo.

El método runCalculations() lanzará tantos hilos como hayamos indicado en NUM_THREADS y luego esperará a que terminen. Todos los hilos comparten el mismo simulador de

operaciones, dado que el simulador está sincronizado, si un hilo realiza la llamada executeOp(), el resto de hilos tendrán que esperar a que termine.

Si observáis el código del hilo en el método run() lo que hace es calcular considerando el número de llamadas por segundo, cual tiene que ser el tiempo entre llamadas consecutivas. Antes de llamar comprueba si ese tiempo ha pasado, y después de llamar lo actualiza para la siguiente llamada.

Debemos modificar la clase para calcular la latencia agregada del método executeOp() para todos los hilos, considerando y sin considerar las latencias acumuladas. Empieza con un único hilo y ve aumentando hasta que las latencias acumuladas se disparen. Realiza todos los cálculos en milisegundos.

Entregar el código de la clase con lo que sea necesario para hacer los cálculos y una copia de los resultados.

b. Ayudas

Dado que queremos calcular la latencia para todos los hilos y no para hilos individuales, debemos compartir el mismo objeto de Histogram para todos ellos. Recuerda que Histogram no está sincronizado, debes utilizar una implementación que sea “thread-safe”.

Recuerda que la latencia acumulada se calcula comparando el tiempo máximo en que esperamos que termine la operación con el tiempo real. Si el tiempo de terminación es mayor que el máximo, debemos sumar la diferencia al resultado de latencia.

Para simplificar solemos considerar el tiempo máximo esperado como el tiempo donde se debería realizar la siguiente llamada, ya que si sobrepasamos ese tiempo se está produciendo congestión.

c. ¿Qué debería obtener y por qué?

Con un único hilo las latencias normales y acumuladas deberían ser iguales. Hemos definido 50 ejecuciones por segundo, es decir un máximo de 20 milisegundos máximo entre ejecuciones consecutivas.

Cada ejecución duerme unos 10 milisegundos con lo entran en tiempo. A medida que incrementamos el número de hilos los tiempos suben, dado que todos acceden a un recurso compartido, si el hilo A y B acceden a la vez, el segundo tardará 20 milisegundos en vez de 10 ya que tiene que esperar a que el primero termine.

Con dos hilos comenzaremos a ver congestión ya que 20 milisegundos es el tiempo que tengo entre operaciones consecutivas, a medida que subimos el número de hilos la congestión aumenta y las diferencias entre latencias normales y acumuladas también.

Las latencias normales se van a mantener en tiempos razonables, pero las acumuladas se van a disparar debido a la congestión.

Esta práctica intenta demostrar que aunque las latencias “sean mas o menos buenas”, tenemos que considerar siempre que ratio de llamadas espero, ya que eso nos da una idea de cuanta congestión tenemos y que latencia real estamos experimentando para el ratio de llamadas esperadas.