Resumen

En esta práctica se han realizado pruebas con hilos en el cálculo de una operación matemática fácilmente paralelizable como es un sumatorio, para comprobar el rendimiento en un sistema multicore.

Como se puede observar en las pruebas empíricas, el número óptimo de hilos es igual al número de núcleos virtuales disponibles en el sistema.

Introducción y objetivos

Se va ha realizado un experimento consistente en comprobar cómo afecta el número de hilos respecto al tiempo realizando una operación matemática paralelizable, concretamente un sumatorio, distribuyendo el cálculo entre diferentes hilos en un mismo sistema.

En teoría, el número óptimo de hilos de un procesador es la cantidad de procesadores lógicos de los que dispone, ya que una cantidad menor deja procesadores inactivos, y una cantidad mayor provoca cambios de contexto que ralentizan el procesado.

En esta práctica pues, se comprobará si esto es así, mediante pruebas empíricas realizadas en sistemas informáticos, utilizando Java como lenguaje de programación, realizando diferentes cálculos con un número diverso de parámetros y posteriormente analizando los tiempos de cómputo.

Diseño del algoritmo

Para la implementación, se ha elegido como versión de Java objetivo (target) la 1.6 (Java 6), utilizando únicamente clases y métodos disponibles en la misma.

A fin de proveer de cierto nivel de abstracción, se han creado dos clases que son:

- SummatoryFunction: interfaz que tiene un método “calculate”, cuyo único parámetro es un entero (la “i” del sumatorio) y devuelve un doble correspondiente al valor en dicho punto.
- SummatoryCalculator: clase abstracta que realiza sumatorios de clases que implementen SummatoryFunction en un rango predeterminado, siempre en pasos de 1.

Para esta prueba no obstante, sólo se hará uso de una implementación de ambos, una clase anónima que calcula la raíz cuadrada, y ThreadingSummatoryCalculator, la implementación del sumatorio empleando hilos.

Mecanismo de reusado de hilos

Respecto al diseño de esta implementación, a fin de experimentar y aprender a trabajar con el concepto de workers y comunicación asíncrona entre hilos, he decidido emplear hilos permanentes (daemons) que no se descartan al finalizar el cálculo.

Esto permite reducir el overhead al crear y eliminar hilos, una operación altamente costosa, en pro de dormitarlos para poder utilizarlos más adelante.

Estos workers se tratan de hilos que se arrancan cuando se crea el ThreadingSummatoryCalculator, cuyo constructor acepta un entero que corresponde con el número de workers del pool.

Cuando se crea la instancia de ThreadingSummatoryCalculator, se crean dichos hilos y son arrancados mediante .start(). Éstos se inicializan asíncronamente, y se sale del constructor.

Así mismo, esta pool se puede redimensionar dinámicamente tras la creación mediante .resizeWorkers(numeroWorkers).

Cálculo – hilo principal

Cuando llega un cálculo, el hilo principal realiza las siguientes operaciones en el objeto ThreadingSummatoryCalculator:

- Almacena todos los parámetros de los cálculos en fields del objeto.
- Instancia un CoundDownLatch llamado finishedCountdown, con el número inicial establecido al número de hilos.
- Reinicializa el resultado global AtomicDouble a cero.
- Invierte la barrera, de modo que los hilos entiendan que hay que empezar a trabajar.
- Despierta los posibles hilos dormidos mediante un .notifyAll() sobre el objeto de espera. La ventaja de usar un sólo objeto de espera frente a uno por hilo es que no hay que iterar sobre los hilos para despertar los que están dormidos.

En este momento, los hilos que estuvieran esperando comenzarán a calcular. El hilo principal, por su parte, emplea el método .await() del finishedCountdown y dormita.

Cuando cada hilo ha finalizado, el método .await() vuelve, y sólo hay que devolver el contenido del AtomicDouble.

Cálculo – workers

Cuando se arranca el worker, éste como parte de la inicialización y mecanismo de espera, comprueba la barrera del objeto padre. En caso de que sea igual al valor actual nuestro, entendemos que no hay nada que computar y que se puede esperar sobre el objeto de espera sin problemas.

En el momento en que llega un cálculo nuevo, pueden darse dos casos, que el hilo haya entrado al .wait() y esté dormido, o no. En el primero, el hilo principal lo despertará con el .notifyAll(), y para el segundo, no entrará al .wait() y empezará directamente el cálculo gracias a la barrera.

Cada hilo de trabajo tiene un ID. Mediante este ID y los parámetros que sacan de la instancia de ThreadingSummatoryCalculator, calculan un rango propio de cómputo mediante

Cada hilo de trabajo opera sobre un bloque determinado del rango total de cómputo, que cada hilo calcula autónomamente accediendo a las variables de "sólo lectura" del ThreadingSummatoryCalculator al que pertenecen, de modo que el cálculo de dichos bloques es también paralelo.

Dichos segmentos son calculados mediante las siguientes operaciones:

$$chunk_{start}(id) = \left\lceil \frac{(id)(end - start + 1)}{threads} - 0.5 \right\rfloor$$

$$chunk_{end}(id) = \left\lceil \frac{(id+1)(end - start + 1)}{threads} - 0.5 \right\rfloor$$

Donde $\lceil \square - 0.5 \rfloor$ se trata del redondeo Half-Down, implementado en Divide.roundHalfDown(dividendo, divisor).

Una vez han calculado el trozo de cómputo, cada hilo calcula el sumatorio entre su $chunk_{start}$ y $chunk_{end}$ en una variable local.

Cuando ha terminado de realizar el cómputo local, se acumula en una variable de tipo AtomicDouble del TSC, que es como un doble pero todas las operaciones son atómicas, incluyendo sumas. Ésta es una clase que emplea "under the hood" un AtomicLong.

Tras finalizar, decrementa el finishedCountdown para informar al hilo principal que ha finalizado, invierte la barrera local y repite el ciclo.

Metodología de las pruebas

Se han realizado pruebas, variando el número de hilos para calcular las operaciones y variando el rango de operación, para calcular cómo el número de núcleos y cómo la sincronización entre hilos afectan al rendimiento, respectivamente.

La prueba de hilos se ha calculado entre 1 y 1024 hilos, en serie geométrica de 2, mientras que la prueba del rango se ha calculado entre 1 y 10.000.000, en serie geométrica de 10. Cada configuración se ha ejecutado 500 veces y se ha calculado la media aritmética del tiempo de ejecución.

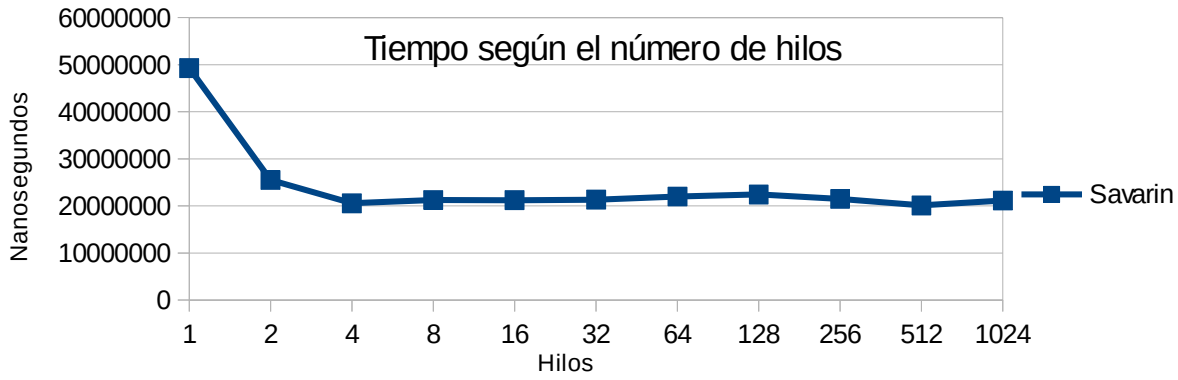
Estas pruebas se han realizado en los siguientes sistemas:

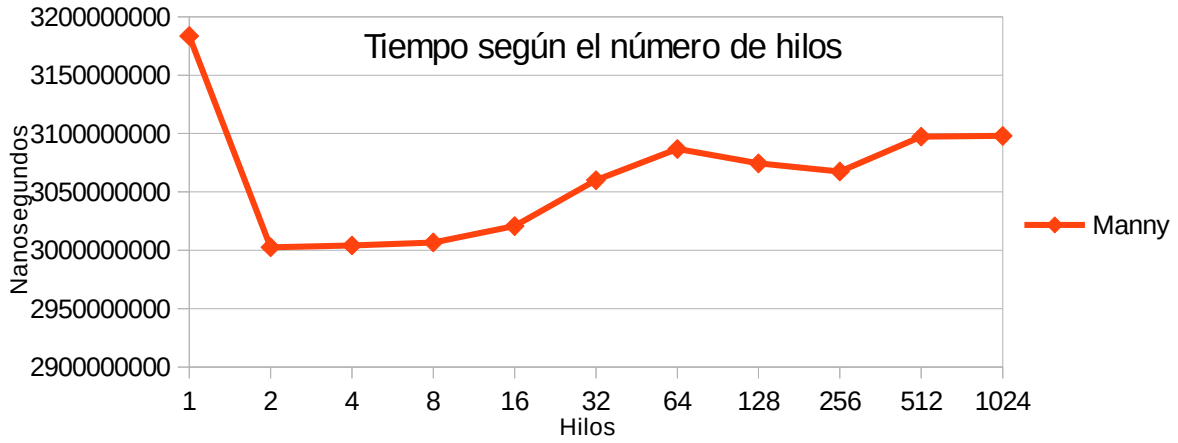
Sistema	Savarin	Manny
Tipo	Sobremesa	Netbook
Procesador	Intel i3 4330 (Haswell)	AMD C70
Núcleos físicos	2	2
Núcleos lógicos	4	2
Memoria RAM	8 GB DDR3 @ 1866MHz	2 GB @ 1333MHz
Sistema operativo	Xubuntu 15.10 (64 bits)	Windows 10 (32 bits)
Máquina virtual	OpenJDK 8u66	Oracle JVM 8u66

Resultados de las pruebas

Tiempo de ejecución según el número de hilos

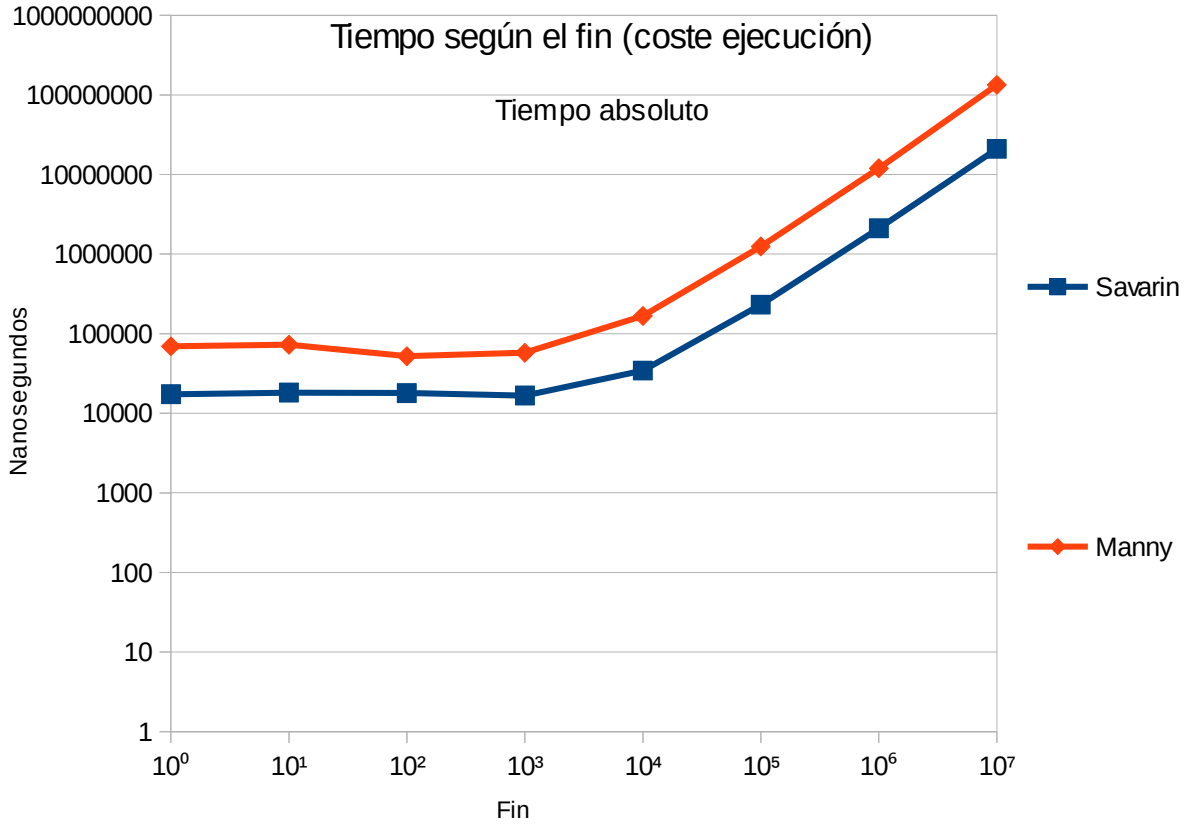
Tiempo en nanosegundos, escalas lineales. Por la diferencia entre ambos valores se dan en dos tablas independientes.

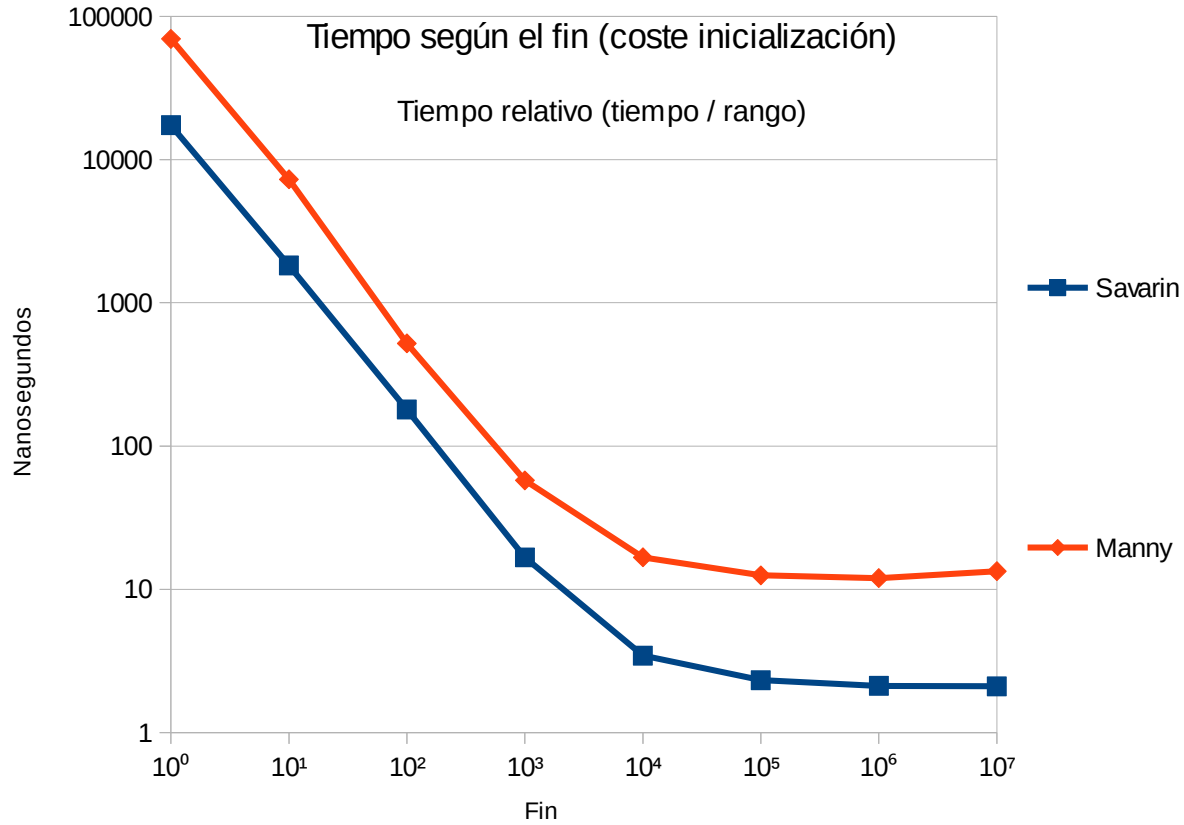




Tiempo de ejecución

En estas gráficas se muestra el tiempo de ejecución en función del tamaño del rango a calcular. Ambos ejes son logarítmicos.

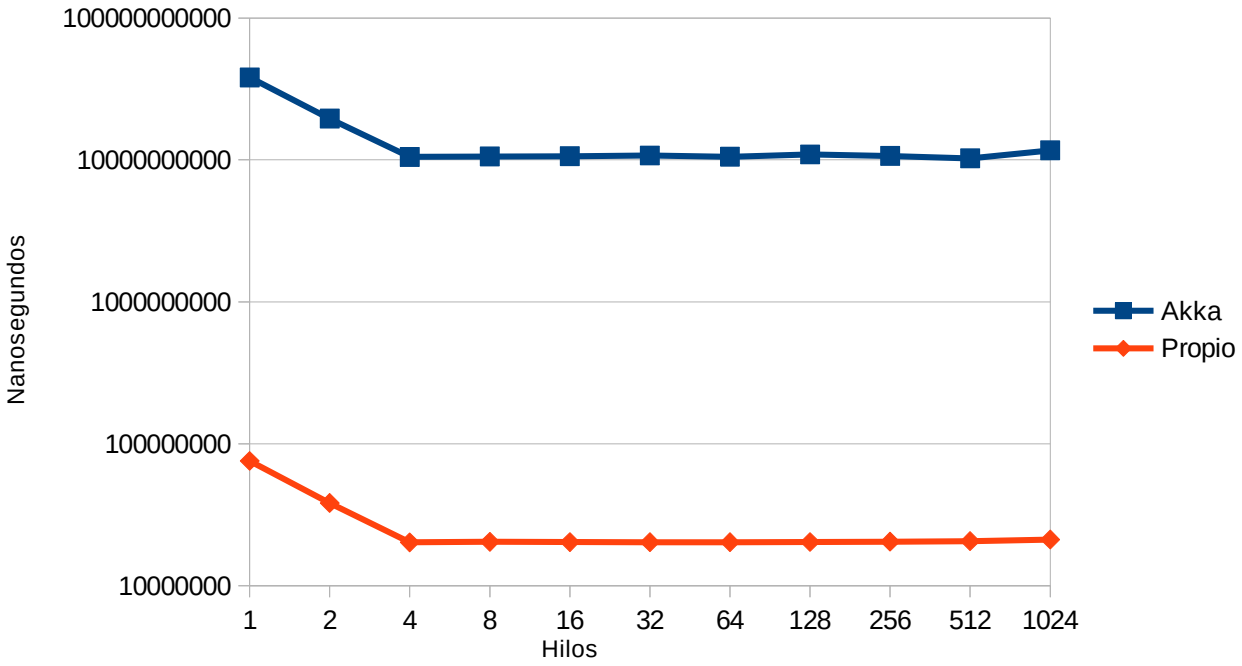




Comparación con Akka

Esta prueba ha sido realizada en un ordenador de prácticas, que tiene la siguiente configuración:

Sistema	Ordenador de prácticas
Tipo	Sobremesa
Procesador	Intel i5 3330 (Sandy Bridge)
Núcleos físicos	4
Núcleos lógicos	4
Memoria RAM	4 GB DDR3 @ 1333MHz
Sistema operativo	Windows 7 (64 bits)
Máquina virtual	Oracle JVM 7u45



Para Akka, el número de mensajes es equivalente al número de hilos/trabajadores.

Como se puede observar es más lento por dos ordenes de magnitud que el presentado aquí.

Conclusiones

Como se puede observar, utilizar diversos hilos con rangos pequeños afecta negativamente al rendimiento del proceso, ya que el coste de sincronización entre hilos supera al de realizar la propia operación matemática.

Solamente empieza a ser útil a partir de 10.000 raíces cuadradas como se puede observar en las tablas, cuando el tiempo comienza a incrementarse linealmente con el rango.

Respecto al número de hilos, se puede observar claramente como para ambos sistemas el número óptimo es el mismo que el número de núcleos virtuales de los que disponen, que pueden ser mayor a los físicos si disponen de HyperThreading como es el caso del sistema Intel.