Universidad Nacional Autónoma de México
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Posgrado en Ciencia e Ingeniería de la Computación

Comparación del desempeño del framework Fork-Join respecto a una implementación empleando el patrón Manager-Worker en el lenguaje de programación Java.

TESINA
Que para obtener el grado de
Especialista en Cómputo en Alto Rendimiento

PRESENTA:
Alán Aarón Barrón Sánchez

Directora de Tesina:
Dra. María Elena Lárraga Ramírez
Instituto de ingeniería

Ciudad Universitaria, Ciudad de México.

PROTESTA UNIVERSITARIA DE INTEGRIDAD Y

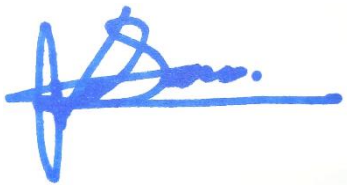
HONESTIDAD ACADÉMICA Y PROFESIONAL

De conformidad con lo dispuesto en los artículos 87, fracción V, del Estatuto General, 68, primer párrafo, del Reglamento General de Estudios Universitarios y 26, fracción I, y 35 del Reglamento General de Exámenes, me comprometo en todo tiempo a honrar a la institución y a cumplir con los principios establecidos en el Código de Ética de la Universidad Nacional Autónoma de México, especialmente con los de integridad y honestidad académica.

De acuerdo con lo anterior, manifiesto que el trabajo escrito titulado *Comparación del desempeño del framework Fork-Join respecto a una implementación empleando el patrón Manager-Worker en el lenguaje de programación Java*, que presenté para obtener el grado de Especialista en Cómputo de Alto Rendimiento, es original, de mi autoría y lo realicé con el rigor metodológico exigido por mi Programa de Posgrado, citando las fuentes, ideas, textos, imágenes, gráficos u otro tipo de obras empleadas para su desarrollo.

En consecuencia, acepto que la falta de cumplimiento de las disposiciones reglamentarias y normativas de la Universidad, en particular las ya referidas en el Código de Ética, llevará a la nulidad de los actos de carácter académico administrativo del proceso de titulación/graduación.

Atentamente



Alán Aarón Barrón Sánchez

Cta. 523013868

Dedicatoria.

A mis padres.

Agradecimientos

A DIOS,

A mi familia, a mi madre que es el amor de mi vida y a mi padre por siempre apoyarme incondicionalmente en todas mis metas y proyectos, por forjarme con valores y carácter. A mis hermanos por estar para mí en todo momento y por ser motivo de mis alegrías. A mis tíos Angela García, Luis Rodríguez, Andrea García y Pedro Campos, por sus consejos, apoyo, amor brindado y confianza que en mi se depositó.

A la Universidad Nacional Autónoma de México por abrirme las puertas de una institución tan prestigiada siendo, sin duda alguna, un pilar muy importante a nivel mundial; por todo lo que me ha brindado, ayudándome a reafirmar mis pasiones y por darme la coyuntura de cumplir un sueño más.

Al Posgrado en Ciencia e Ingeniería de la Computación, por haberme dado la oportunidad de seguir creciendo profesionalmente al mejor nivel educativo de este hermoso país.

A mi tutora, Dra. María Elena Lárraga Ramírez, por su tiempo, dedicación y paciencia; asimismo, por haberme motivado a confiar en mí mismo, poniendo de manifiesto el profesionalismo, liderazgo y calidad que la caracterizan como persona.

A los doctores, Ernesto Rubio Acosta, Lukas Nellen Filla, Jorge Luis Ortega Arjona y al Ing. Juan Luciano Díaz González por su trayectoria y gran dedicación a la labor científica, por transmitirme sus conocimientos, dedicando su tiempo a la docencia y por ser para mí un modelo a seguir. Así como también al Ing. Israel Velázquez Gutiérrez por los conocimientos transmitidos, su apoyo y comentarios que permitieron llevar a buen fin este trabajo de tesina.

A mis sinodales, el Dr. Ernesto Rubio Acosta y el Mtro. Juan Luciano Díaz González, por su dedicación, paciencia y tiempo para llevar a cabo este proyecto.

A todos los que hicieron posible este programa de Posgrado en Ciencia e Ingeniería de la Computación, comenzando por el Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas(IIMAS), al coordinador el Dr. Javier Gómez Castellanos y a la asistente de procesos la Sra. Ma. de Lourdes González Lora.

Resumen

El cómputo paralelo es de suma importancia en la actualidad debido a su capacidad para agilizar el procesamiento de grandes volúmenes de datos y resolver problemas complejos de manera más competente. Permite dividir tareas en múltiples unidades de procesamiento, para que trabajen simultáneamente en diferentes partes del problema. Por lo que, a lo largo de varias décadas han surgido metodologías para la construcción de algoritmos paralelos y de software para su implementación. En esta dirección, el lenguaje de programación Java, uno de lo más usados a nivel mundial, proporciona soporte para la implementación y la ejecución de tareas en paralelo. Para ello, a partir de la versión 7 del JDK de Oracle, Java incluye el framework nombrado Fork-Join, el cual se diseñó con la finalidad de poder ejecutar tareas de manera paralela. Desde entonces, Fork-Join ha sido modificado a través de las distintas versiones del JDK que se han liberado; de tal manera que, para la versión 8 ya se habían incluido la mayoría de las funcionalidades que actualmente contempla y que le aportan una mayor robustez y facilidad de uso respecto a su liberación original.

Fork-Join se basa en la idea de dividir una tarea en varias subtareas independientes y luego combinar los resultados de estas subtareas para llegar a un resultado final. Así, la ejecución de la tarea principal se divide (fork) en múltiples threads o procesos más pequeños, cada uno de los cuales se encarga de una subtarea. Una vez que todas las subtareas se han completado, los resultados se unen (join) para producir el resultado final. Este framework es especialmente útil cuando se tienen tareas recursivas o iterativas que pueden dividirse en subtareas de tamaño similar, ya que se puede aprovechar mejor el paralelismo y acelerar la ejecución en sistemas con múltiples núcleos o procesadores [1] [2] [3] [4].

Por otra parte, el patrón de diseño Manager-Workers (Gerente-Trabajadores) se utiliza para abordar problemas relacionados con la concurrencia y la paralelización de tareas. Este patrón, se basa en la idea de tener un “gerente” (Manager) que coordina y asigna tareas a varios trabajadores (Workers) para que las realicen en paralelo. La implementación concreta del patrón Manager-Workers puede variar dependiendo de los detalles específicos del problema que se esté abordando, de las herramientas y las bibliotecas utilizadas. Sin embargo, el uso de threads, sincronización y estructuras de datos compartidas son conceptos clave en la implementación de este patrón. Cada oponente trabaja de forma similar, cada uno cuenta con características que los hacen más atractivos en función al escenario que se presente.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1 Contexto

Dados los avances tecnológicos de hoy en día, así como la gran cantidad de datos que cada vez son mayores, la programación paralela se vuelve vital para dar respuesta a los problemas y retos que se presentan. Se han desarrollado tecnologías para operar en ambientes multiprocesador, así como metodologías para la construcción de algoritmos paralelos y de software paralelo. El poder del paralelismo se centra en la partición de un gran problema con el fin de hacer frente a la complejidad. Así, la partición se realiza con la finalidad de dividir un problema tan grande en subproblemas más pequeños que sean más fáciles de entender, y en los que se pueda trabajar por separado, a un nivel más "cómodo y eficiente computacionalmente". Para ello, diversos patrones de software (relaciones forma-función) se han desarrollado [3]. Entre estos, una solución que ha tenido relevancia para desempeñarse en este tipo de ambientes ha sido el modelo Fork-Join, una aproximación simple y eficiente para implementar programas paralelos. Este modelo consiste en dividir las tareas en subtareas más pequeñas para posteriormente unirlas en una sola y dar solución a un problema, particularmente, para el caso del lenguaje de programación Java, el framework fork-join ha estado presente desde la versión 7 como parte de la biblioteca estándar (java.util.concurrent). Uno de los patrones de software paralelo más usados es el Manager-Workers el cual no tiene una fecha de creación específica, ya que es un concepto que ha ido evolucionando a lo largo del tiempo. El patrón Manager-Workers es una variante del patrón Maestro-Esclavo [5] para sistemas paralelos, la variación se basa en que los componentes de este patrón en lugar de reactivos son proactivos. De tal manera que, cada uno de los componentes de procesamiento realiza simultáneamente las mismas operaciones, independientemente de la actividad de procesamiento de otros componentes. La característica más importante es preservar el orden de los datos.

1.2 Problemática

Aunque Fork-Join ha demostrado ser un modelo eficaz para problemas pequeños o medianos, pocos estudios existen de cómo se comporta cuando se trabaja con problemas que requieren gran cantidad de recursos y cómputo científico con datos en representación de punto flotante.

¿Fork-Join es adecuado para resolver problemas de cómputo científico bajo el lenguaje de programación Java? ¿Es posible que otro patrón de diseño mejore su funcionamiento?

1.3 Definición del problema

En esta tesina, se presenta una comparación del desempeño de usar el framework Fork-Join para la resolución de un problema, con respecto a una implementación empleando el patrón Manager-Workers bajo el lenguaje de programación Java, sobre las mismas circunstancias de hardware y problemática. La medida de desempeño se considerará en términos de tiempo de ejecución en milisegundos, segundos o minutos. Particularmente, el problema a resolver es la multiplicación de matrices, un problema que se sabe es adecuado para la realización de pruebas de desempeño de sistemas paralelos.

1.4 Objetivo

El objetivo de este trabajo que se presenta, es hacer un análisis del rendimiento que presenta el uso del framework Fork-Join de Java para realizar una tarea de cálculos numéricos de alta precisión(notación flotante), en comparación con otra implementación de la misma tarea, pero haciendo uso para su diseño e implementación del patrón Manager-Workers, con el fin de evaluar la robustez o limitante que el framework Fork-Join nos proporciona para realizar cálculos científicos computacionalmente costosos, cuando se utiliza para su implementación el lenguaje Java el caso de estudio será la multiplicación de matrices.

1.5. Hipótesis

Cuando se utiliza aritmética entera, nada va a superar a un paralelismo basado en Fork-Join, sin embargo, en el momento en que se utiliza notación científica, creemos que el desempeño de la multiplicación de matrices puede mejorarse al hacer uso de un patrón de diseño como Manager-Worker, donde el programador tiene el control del paralelismo. Para llevar a cabo los objetivos la metodología que se propone son tres escenarios:

- Escenario secuencial.
- Escenario multiproceso, el cual se divide en dos, uno utilizando Fork-Join y otro con un diseño propio utilizando el patrón de software Manager-Workers.
- Escenarios de matrices con diferentes tamaños.

1.6 Objetivos particulares

De esta manera, los objetivos particulares son:

1) Realizar una implementación de la multiplicación de matrices usando el patrón Fork-Join en el lenguaje de programación Java.
2) Realizar una implementación de la multiplicación de matrices usando el patrón Manager-workers en el lenguaje de programación Java.
3) Evaluar el desempeño de ambas implementaciones considerando diferentes tamaños de las matrices y tipos de sus datos. Particularmente, realizar una evaluación de los resultados considerando diferentes números de procesadores y tamaños de las matrices; así como de los tipos de datos que las componen.

1.7 Contribución

El trabajo de investigación que aquí se pretende contribuirá a comprender las ventajas y limitaciones del modelo Fork-Join cuando se implementa en el lenguaje de programación Java, particularmente, cuando se utilizan componentes de matrices con notación en punto flotante. Se pretende este estudio sirva como base para evaluar de mejor manera el diseño de software paralelo en el lenguaje de programación Java y la importancia de evaluar los patrones de diseño a usar.

1.8 Organización del trabajo

El resto de este trabajo de tesina está organizado de la siguiente manera. Con finalidad de que el lector comprenda mejor esta tesina, en el capítulo 1 se introducen conceptos requeridos para ello. En el capítulo 2, se presentan los antecedentes de este trabajo de tesina. En el capítulo 3, se presenta el análisis comparativo de la multiplicación de matrices con diferentes implementaciones y tamaños del problema bajo estudio. En el capítulo 4, se presentan los resultados de comparar diferentes tamaños del problema cuando se evalúan diferentes métricas para ello. Finalmente, en el capítulo 5, se presentan las conclusiones de este trabajo.

Capítulo 2

Marco teórico

En este capítulo, se introducen aquellos conceptos que el lector puede requerir para un mejor entendimiento y comprensión del trabajo que se desarrolla en esta tesina. Así, se presentan conceptos relacionados, con el lenguaje Java, el paralelismo y los patrones de diseño paralelo Fork-Join y Manager Workers. También, se presentan conceptos relacionados a las métricas requeridas para evaluar el desempeño de las implementaciones que se realizan.

2.1 El lenguaje Java

Java es un lenguaje de programación que surgió en 1991 cuando un grupo de ingenieros de Sun Microsystems trataron de diseñar un nuevo lenguaje de programación adscrito a electrodomésticos. La mínima potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo apto de generar código de tamaño muy reducido. Se buscaba diseñar un lenguaje que permitiera programar una aplicación una sola vez que luego pudiera ejecutarse en distintas máquinas y sistemas operativos. En ese momento, ni siquiera se llamó como actualmente lo conocemos “Java”; se llamó Oak.

El proyecto de televisión inteligente e interactiva nunca se materializo. De modo simultaneo, a principios de la década de los 90 surgió Internet y con ella, la aparición de los primeros navegadores web. Los lideres del Green Project fueron sensatos de la importancia que iba a tener Internet y orientaron su lenguaje de programación Oak para que programas escritos en este lenguaje de programación se pudiesen ejecutar dentro del navegador web Mozilla. Y este fue el inicio de Java, así llamado porque cuando se intentó registrar el nombre Oak este ya estaba registrado [4].

El lenguaje de Java fue introducido públicamente en 1995 y fue rápidamente adoptado por la comunidad de desarrolladores debido a sus características de seguridad, portabilidad y facilidad de uso. En 1996, Sun Microsystems lanzó la primera versión de Java Development Kit (JDK).

Las nuevas características generales que se añadieron son:

- Seguridad, ya que los programas que se ejecutan en un navegador se descargan desde Internet.

- Potencia, ya no se tenía la delimitación de la ejecución en dispositivos de electrónica de consumo.

La plataforma Java se compone de dos partes: el lenguaje de programación Java y la máquina virtual de Java (JVM). El lenguaje de programación se utiliza para escribir el código fuente de una aplicación, mientras que la JVM es responsable de ejecutar ese código en cualquier plataforma que soporte la plataforma Java.

2.1.1 Características de Java

Java representa un lenguaje de programación de propósito general, centrado en la orientación a objetos, que deriva conceptos esenciales de otros lenguajes. Específicamente, incorpora de Smalltalk la noción de ejecutar programas en una máquina virtual, mientras que adopta la sintaxis proveniente del lenguaje de programación C++. La utilización de la máquina virtual en Java asegura la independencia de la plataforma. En caso de contar con una máquina virtual compatible con nuestra plataforma, será posible ejecutar el mismo programa desarrollado en Java sin requerir una nueva compilación.

Algunas de las principales características de Java incluyen:

- Portabilidad: Los programas escritos en Java se pueden ejecutar en cualquier plataforma que tenga una máquina virtual de Java (JVM) instalada.
- Seguridad: La seguridad en Java se logra a través de características como la comprobación de tipos de datos, la administración de memoria, la gestión de excepciones y el control de acceso.
- Orientado a objetos: Java es un lenguaje de programación orientado a objetos, lo que significa que se basa en el concepto de objetos y clases. Los objetos son instancias de una clase y las clases son plantillas para crear objetos.
- Facilidad de uso: Java es un lenguaje fácil de aprender y utilizar, especialmente para aquellos que tienen experiencia en otros lenguajes de programación orientados a objetos.
- Robusto: el compilador de Java detecta muchos errores que otros compiladores solo detectarían en tiempo de ejecución o incluso nunca.

En resumen, Java es un lenguaje de programación versátil, seguro, orientado a objetos, fácil de usar y de alto rendimiento que se utiliza en una amplia variedad de aplicaciones[7].

2.1.2 Programación orientada a objetos en Java

La Programación Orientada a Objetos (POO) corresponde a un paradigma de programación que se basa en la noción de clases y objetos para estructurar un software en componentes sencillos y reutilizables. A través de la creación de clases, se forman instancias individuales de objetos que interactúan por medio de mensajes, simulando situaciones del mundo real. Estos objetos contienen atributos y métodos que los definen. Un beneficio destacado de la POO es su capacidad para generar programas y módulos más legibles, mantenibles y reutilizables, facilitando la comprensión y el mantenimiento de códigos, así como promoviendo la reutilización de código.

La POO se basa en cuatro conceptos fundamentales: encapsulamiento, herencia, polimorfismo y abstracción (ver Tabla 1 Fundamentos de la POO.

).

Encapsulamiento:	**Es la capacidad de ocultar la complejidad de un objeto y exponer solo las partes relevantes para su uso.**
Herencia:	Es un mecanismo que permite crear una nueva clase a partir de una clase existente, pero con nuevas características.
Polimorfismo:	Permite que una misma clase tenga múltiples comportamientos, dependiendo del contexto en el que se utilice.
Abstracción:	Es la capacidad de definir un objeto solo por sus características esenciales, sin tener en cuenta los detalles de implementación.

Tabla 1 Fundamentos de la POO.

2.1.3 Programación multihilo en Java

La programación multihilo se refiere a un enfoque de programación en el que diferentes secciones del código de un mismo programa se intercalan durante su ejecución. Estas secciones individuales se denominan hilos (o threads en inglés). Los hilos, también conocidos como hebras o contextos de ejecución, son fragmentos de código que la JVM (Máquina Virtual de Java) ejecuta simultáneamente en sistemas multiprocesador y de manera intercalada en sistemas monoprocesador. Esto implica que la ejecución de los hilos se lleva a cabo mediante la asignación de intervalos de tiempo específicos para cada uno de ellos.[8].

En Java un hilo es una clase que desciende de la clase java.lang.Thread o bien que extiende a la interfaz Runnable (útil en los casos de que la clase ya formé parte de una jerarquía de clases esto es que ya derivé de otra clase). Es una secuencia de instrucciones ejecutables. Así, la programación multihilo es el proceso de utilizar dos o más hilos de control en un mismo programa. En Java un hilo puede encontrarse en cualquiera de los siguientes estados:

Nace	**Se ha declarado el hilo, pero no se ha sido ejecutado (método start()).**
Listo	El hilo está preparado para ser ejecutado, pero el planificador aún no ha decidido su marcha.
Ejecutándose	El hilo está ejecutándose en la CPU.
Dormido	El hilo se ha detenido durante un instante de tiempo, por el método sleep().
Bloqueado	El hilo está pendiente de una operación y no volverá a un estado listo hasta que esta termine.
Suspendido	El hilo ha sido temporalmente detenido por el método suspend(), para poder seguirse ejecutando habrá que llamar a la función resume().
Esperando	El hilo ha detenido su ejecución debido a una condición que puede ser interna o externa, mediante la llamada wait() y solo se podrá reanudar hasta la llamada al método notify() o notifyAll().
Muerto	El hilo ha terminado su ejecución con éxito.

Tabla 2 Estados de un hilo en java

2.2 Organización de la memoria

En [3], Jorge L. Ortega Arjona menciona que la memoria y los periféricos son recursos comunes o compartidos que emplean los procesadores para poder comunicarse entre sí durante la ejecución de un programa paralelo o distribuido, y dependiendo del grado de interacción (entre los procesos y los periféricos) se

clasifican en dos tipos: fuertemente acoplados(tightly-coupled) o débilmente acoplados(loosely-coupled) para grados de interacción alto y bajo respectivamente.

Los sistemas paralelos MIMD presentan dos arquitecturas diferenciadas: memoria compartida y memoria distribuida.

El modelo de memoria utilizado hace que la programación de aplicaciones paralelas para cada caso sea esencialmente diferente.

2.2.1 Modelo de memoria compartida

El modelo de memoria compartida se refiere a un modelo de programación en el que varios procesos pueden acceder a la misma región de memoria compartida en un sistema operativo para compartir datos. Por lo tanto, múltiples procesadores normalmente funcionan de forma independiente, compartiendo los mismos recursos de memoria. Cada ubicación de memoria es única e idéntica para cualquier procesador del sistema. Los procesos pueden escribir y leer datos en esta región compartida, lo que permite la comunicación entre ellos (ver Figura 1).

Ventajas:

- Al tener un espacio de direcciones global simplifica la programación
- Compartir datos entre procesos es rápido y uniforme

Desventajas:

- Difícil escalamiento de cantidad de memoria y procesadores en las computadoras con este tipo de memoria.
- Los sistemas de este tipo suelen representar altos costos.

Por otra parte, aunque el modelo de memoria compartida es muy útil para compartir datos entre procesos, ya que es mucho más rápida que las técnicas alternativas de intercomunicación, también requiere que los procesos compartan el mismo espacio de direcciones virtuales. Como consecuencia, puede representar un problema en sistemas operativos que no tienen un mecanismo de protección conveniente para evitar que los procesos accedan a memoria que no les pertenece.

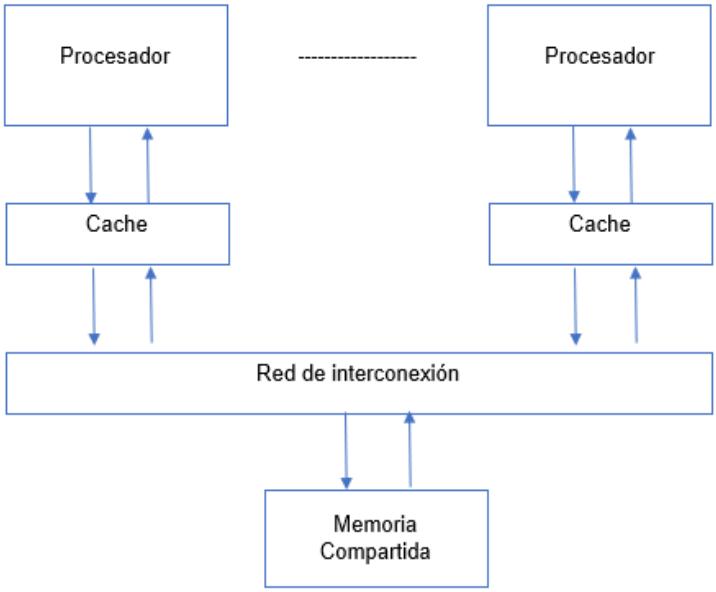


Figura 1 Memoria Compartida

2.2.2 Modelo de memoria distribuida

Un sistema de este tipo, sólo permite que cada procesador tenga acceso directo a su propia memoria local. La comunicación con otros procesadores se realiza utilizando operaciones de entrada y salida a través de mecanismos de comunicación entre procesos, proporcionados por una red de interconexión. De tal manera que, todos los datos se almacenan y se procesan en diferentes dispositivos, lo que permite que el trabajo sea distribuido entre ellos, disminuyendo el tiempo de procesamiento de grandes volúmenes de datos y ampliando la escalabilidad del sistema (ver Figura 2). La red de interconexión está compuesta por un conjunto de enlaces entre procesadores, basados en una topología lineal, anillo, estrella, etc. La red es el medio utilizado para llevar a cabo el intercambio de datos y esta durante la ejecución de un programa puede permanecer estática o dinámica acorde a las necesidades del programa[3].

La comunicación entre procesos en estos sistemas se realiza a través de paso de mensajes.

Ventajas:

- La memoria normalmente escala con la cantidad de procesadores.
- Cada procesador puede acceder a su propia memoria, sin ninguna interferencia o sobrecarga.

- Rentabilidad
- Permite que los sistemas informáticos se adapten a diferentes requisitos y entornos

Desventajas

- La implementación de sistemas de memoria distribuida puede ser compleja y requiere conocimientos especializados en programación distribuida y sistemas de red.
- La comunicación entre nodos en un sistema de memoria distribuida puede ser lenta debido a la latencia y al ancho de banda limitado de las redes de comunicación.

La memoria distribuida puede ser más compleja y difícil de implementar que otros modelos de computación, puede tener algunas delimitaciones en términos de latencia, ancho de banda, confiabilidad y seguridad.

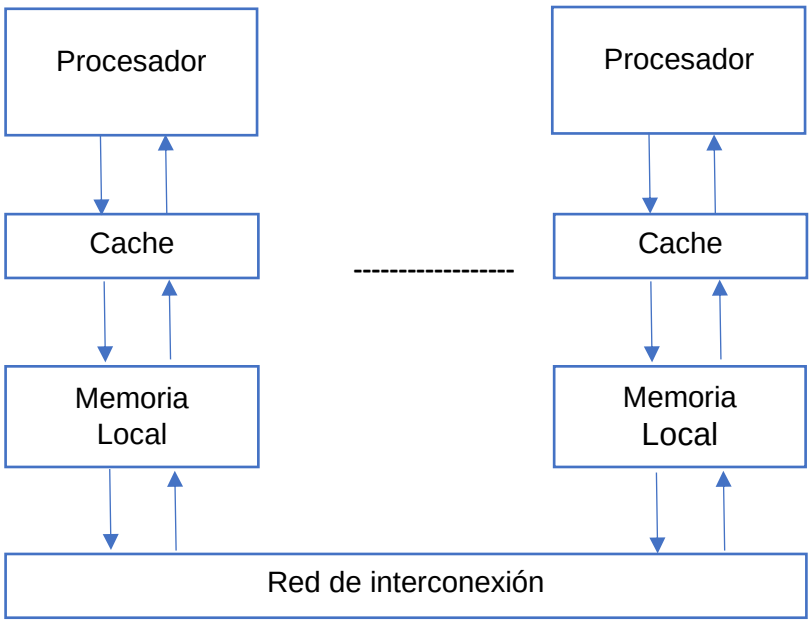


Figura 2 Memoria distribuida

2.3 Procesamiento paralelo

El procesamiento paralelo implica la fragmentación de un problema en múltiples partes que son procesadas simultáneamente por varios componentes de ejecución. Este enfoque tiene como objetivo lograr una mayor eficacia en el rendimiento. Actualmente, se identifica con términos como hilos, threads o subprocesos. Es fundamental resaltar que la planificación de un programa bajo esta perspectiva de cómputo debe considerar factores como la estructura de ejecución, las necesidades

temporales y espaciales, junto con otros elementos. La partición es necesaria para dividir un problema tan grande en subproblemas más pequeños que son más fáciles de resolver. La programación paralela es una habilidad valiosa para los desarrolladores y científicos de datos que trabajan en aplicaciones de alta demanda computacional, ya que permite mejorar el rendimiento y la eficiencia de los sistemas informáticos [5].

Ventajas

- Mejora el rendimiento: múltiples tareas ejecutadas simultáneamente.
- Eficiencia energética: al trabajar varios procesadores al mismo tiempo, lo que puede mejorar la eficiencia energética por lo que puede reducir los costos operativos.
- Mayor capacidad de procesamiento: procesamiento de grandes datos o cálculos complejos más rápidamente.
- Respuesta rápida: puede mejorar la capacidad de respuesta de una aplicación, esencialmente en aplicaciones en tiempo real como por ejemplo la automatización industrial.

Desventajas

- Complejidad: la complejidad puede dificultar la programación.
- Problemas de concurrencia: se pueden generar problemas como condiciones de carrera, bloqueos y deadlock, que pueden ser difíciles de detectar.
- Dependencia del hardware: totalmente dependiente del número de procesadores y múltiples núcleos he ahí el problema, ya que puede generar más gastos en ello.

2.4 Métricas de desempeño en paralelismo

Como ya se mencionó, un programa paralelo es aquel que lleva a cabo operaciones de forma simultánea [8]. Existen varias razones principales para realizar programas paralelos, una de las principales es que pueden realizar una ejecución más rápida que la versión secuencial del programa. Se cuentan con distintas métricas para evaluar si efectivamente el programa paralelo tiene un mejor desempeño que el programa serial.

2.4.1 Tiempo de ejecución

El tiempo de ejecución en se refiere al tiempo que tarda un programa en ejecutarse desde el inicio hasta la finalización. Para poder medirlo, es importante tomar como parámetro la media de un número considerable de mediciones, para contrarrestar la imprecisión debido al no determinismo de la elección de los procesos por el calendarizador de tareas del sistema operativo. El tiempo de ejecución puede variar según diversos factores, como el hardware, el sistema operativo y la carga del sistema, por lo que los resultados pueden no ser siempre precisos [8].

2.4.2 SpeedUp (aceleración)

Métrica para evaluar el beneficio potencial de un programa consiste en medir el tiempo que un solo procesador tarda en realiza una tarea contra en tiempo que toma completarse la misma tarea con n procesadores paralelos. El factor de SpeedUp al usar n procesadores se define como en la ecuación (1):

$$\mathcal{S}\mathrm{p}(\mathrm{n}) = \frac{tiempo\ de\ ejecución\ serial}{tiempo\ de\ ejecución\ paralela} = \frac{T_s(n)}{T_p(n)} \qquad (1)$$

donde p representa al número de procesadores que se usaron para resolver un problema paralelo con tamaño de entrada n, y $T_S(n)$ es la aceleración absoluta, es decir, el tiempo de ejecución correspondiente a la mejor implementación secuencial para resolver el mismo problema.
El SpeedUp es un concepto importante en la optimización de programas y algoritmos, ya que permite hacer una evaluación de las mejoras realizadas y comparar diferentes enfoques para lograr un rendimiento óptimo. Sin embargo, es importante tener en cuenta que el SpeedUp puede estar limitado por varios determinantes, como el tamaño del problema, la eficiencia algorítmica y las limitaciones del hardware [9].

2.4.3 Eficiencia

La eficiencia en cómputo se refiere a la virtud de un sistema o programa para utilizar los recursos disponibles de manera óptima y ejecutar las tareas de manera rápida y efectiva. Sean $\mathrm{T_s(n)}$, $T_p(n)$, que denotan el mejor tiempo de ejecución serial y el tiempo de ejecución paralela cuando se usan *p* procesadores [9] y se considera un tamaño de entrada *n* entonces, entonces la eficiencia $E(n)$se define como en (2).

$$E(n) = \frac{tiempo\ de\ jecución\ serial}{(tiempo\ de\ ejecución\ paralela)*(número\ de\ procesadores)} \quad (2)$$

También, se puede puntualizar de la siguiente manera:

$$E(n) = \frac{T_S(n)}{T_p(n)*p} * 100 \quad (3)$$

2.4.4 Ley de Amdahl

Si suponemos que en todo programa hay una fracción no paralelizable, obtendremos una cota superior al SpeedUp que como máximo puede alcanzar un programa ejecutándose en paralelo. La ley establece que el rendimiento máximo que se puede obtener al mejorar una parte de un sistema está limitado por la fracción de tiempo que se dedica a esa parte. En otras palabras, si se mejora una parte de un sistema que representa un cierto porcentaje del tiempo total de ejecución, el rendimiento general del sistema no puede mejorar en más de ese porcentaje.

Determina una cota superior al SpeedUp que como máximo puede alcanzar un programa ejecutándose en paralelo, es decir, la aceleración máxima Sp_M de un algoritmo en una plataforma paralela con un tamaño fijo de su capacidad de trabajo. Asimismo, se establece que la aceleración tiende a la proporción serial que no se puede paralelizar cuando se incrementa el número de procesadores p.

Por lo tanto, la aceleración máxima de un programa paralelo queda compuesta del tiempo de ejecución de la fracción serial, f y el tiempo de ejecución de la fracción paralela $(1-f)/\mathrm{p}$, quedando expresada como en (4):

$$\mathcal{S}\mathfrak{p}(\mathrm{n}) = \frac{Ts\ (n)}{[f * Ts\ (n) + ((\frac{1-f}{P} * Ts\ (n)]} = \frac{1}{(f + \frac{(1-f)}{P})} \leq \frac{1}{f} \quad (4)$$

donde p representa al número de procesadores utilizados para resolver un problema de tamaño n [10]. En la Figura 3 Speedup correspondiente a la ley de Amdahl.

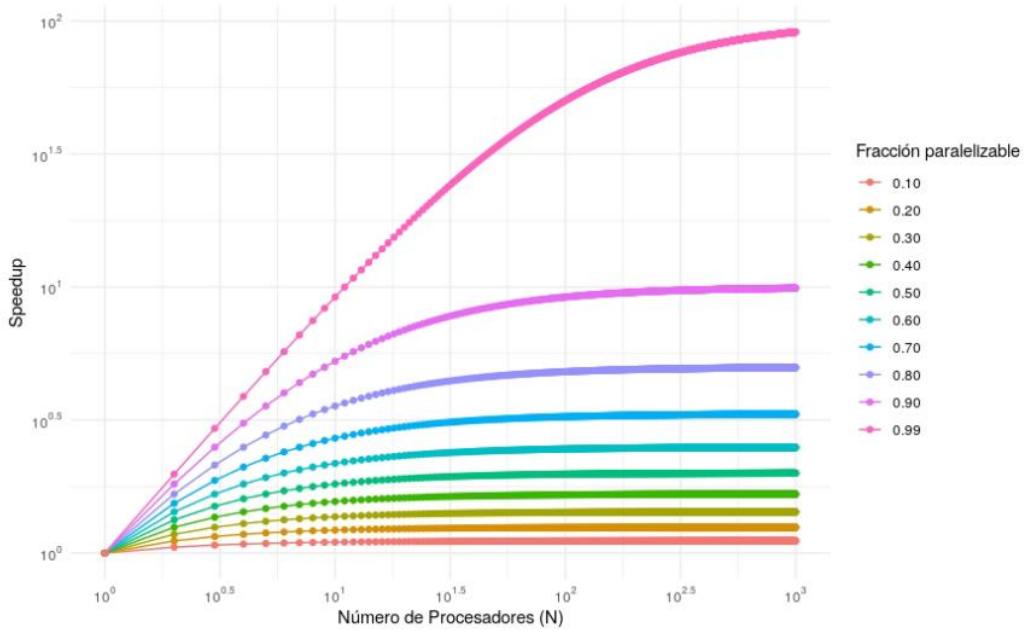


Figura 3 Speedup correspondiente a la ley de Amdahl.

2.4.5 Ley de Gustafson

De la Ley de Amdahl subyace cierto pesimismo sobre el SpeedUp. El principio de este resultado surge del supuesto que el problema a atender (la tara) es fijo. Es decir, que la fracción paralelizable del código es fijo y no depende del tamaño de la tarea. Gustafson planteó la discusión considerando que el paralelismo aumenta cuando el tamaño del problema también se incrementa. Supongamos N procesos paralelos. El tiempo tomado en procesar la tarea en N procesadores esta dado por:

$$\mathrm{Tp(N)} = (1 - \mathrm{f})Tp + fTp = Tp \qquad (5)$$

Cuando la tarea es ejecutada en un solo procesador, la parte serial no cambia, pero la parte paralela se incrementa entorno a:

$$S(N) = \frac{Tp\ (1)}{Tp\ (N)} = (1 - f) + Nf = 1 + (N - 1)f \qquad (6)$$

En la ecuación (6), se observa que el speedup es mayor que cero incluso para valores pequeños de *f*, en tanto que la situación mejora en la medida que aumenta *N* [9]. Para obtener el valor de un speedup, necesita cumplirse la siguiente desigualdad:

$$f(N-1) \gg 1 \quad (7)$$

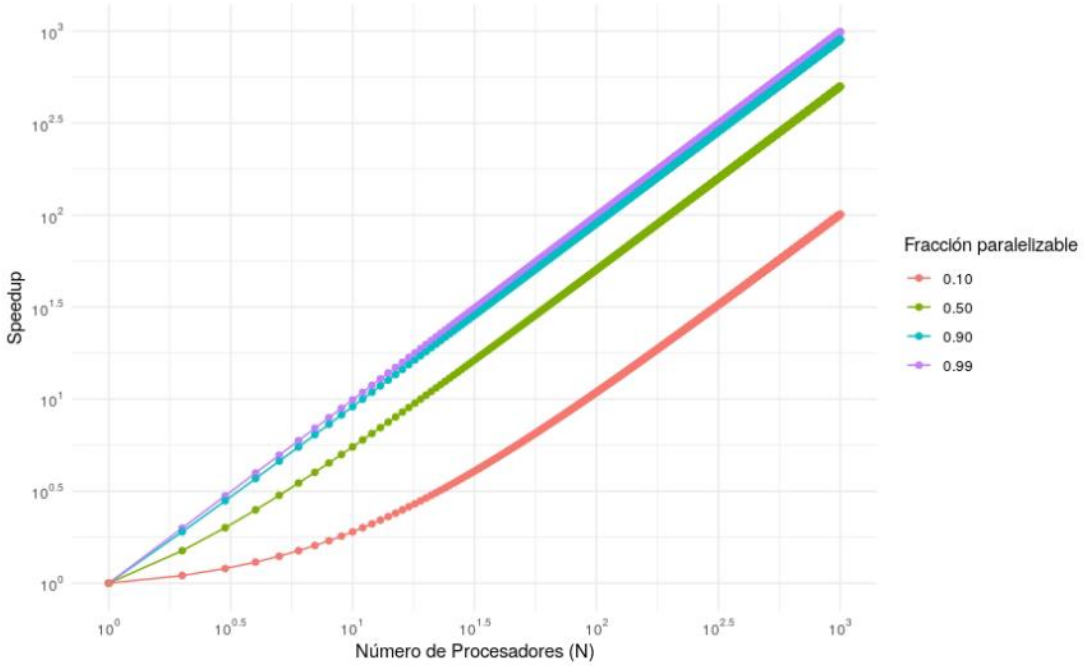


Figura 4 SpeedUp correspondiente a la ley de Gustafson

2.5 Patrón de software

Los patrones de software se centran en capturar, sistematizar experiencias y técnicas que resultan ser exitosas en el desarrollo de software, describen soluciones exitosas a problemas de software con la intencion de crear buenas prácticas de diseño y programación, su principal objetivo es recopilar experiencia y técnicas de diseño para software. Además, ayudan a los desarrolladores impartir mejor su experiencia, razonar sobre lo que hacen y el por qué lo hacen. Un patrón se define como una solución recurrente a un problema estándar [4] [10], es una forma de capturar y sistematizar la práctica comprobada en cualquier disciplina [11] [12].

A continuación la descripción de un patrón acorde a The POSA Form [5], muestra en la Tabla 3.

Nombre	Palabra o nombre que esencialmente describe el patrón.
Breve	Descripción del patrón indicando lo que hace.
Ejemplo	Una problemática del mundo real
Contexto	Situación en la que se aplica el patrón.
Problema	Descripción del conflicto que resuelve el patrón.
Solución	Principio fundamental de la solución esto en una breve descripción.
Estructura	Una descripción de los aspectos estructurales del patrón.
Dinamica	Escenarios que describen el comportamiento de los participantes del patrón por medio del tiempo.
Implementación	Pautas para implementar el patrón.
Ejemplo resuelto	Representa una discusión sobre aspectos importantes para resolver el problema propuesto como ejemplo.
Usos conocidos	Ejemplos de usos del patrón, al menos tres.
Consecuencias	Beneficios que se producen al aplicar el patrón.
Consultas	Referenciarse con otros patrónes que resuelvan problematicas similares, para asi poder refinar el patrón que se esta definiendo.

Tabla 3 Descripción de un patrón de diseño acorde a The POSA Form.

2.5.1 Categorías de patrones de software

Los patrones de software cubren varios niveles de escala y abstracción. Van desde los que estructuran un sistema de software en subsistemas, hasta aquellos que se utilizan para implementar aspectos de diseños particulares en un lenguaje de programación definido.

A continuacion las tres categorías en las que se dividen los patrones de software:

- Patrones arquitectónicos. Esquematización de organización estructural esencial para los sistemas de software, incluye reglas y pautas para organizar la relación de ellos.

- Idioms o modismos. Patrón de bajo nivel especifico de un lenguaje de programación, usando las caracteristicas de los lenguajes dados implementa aspectos particulares.
- Patrones de diseño. Refina los subsistemas o componentes de un sistema de software, dentro de un contexto en particular resuelve un problema de diseño [13] [14].

2.5.2 Patrón de software Manager-Workers

En [3], se menciona que el patrón Manager-Workers es una variante de patrón Master-Slave de Buschmann [15] para el caso de sistemas paralelos, considerando un planteamiento de paralelismo de actividad en el que se dividen tanto algoritmos como datos son y las mismas operaciones son llevadas a cabo sobre datos ordenados. La variación radica en el hecho que los componentes del patrón son proactivos en lugar de reactivos. Esto quiere decir que, cada componente de procesamiento lleva a cabo simultáneamente las mismas operaciones e independientemente del procesamiento de los otros componentes, como puede apreciarse de la Figura 5 . Es importante que el orden de los datos se preserve.,

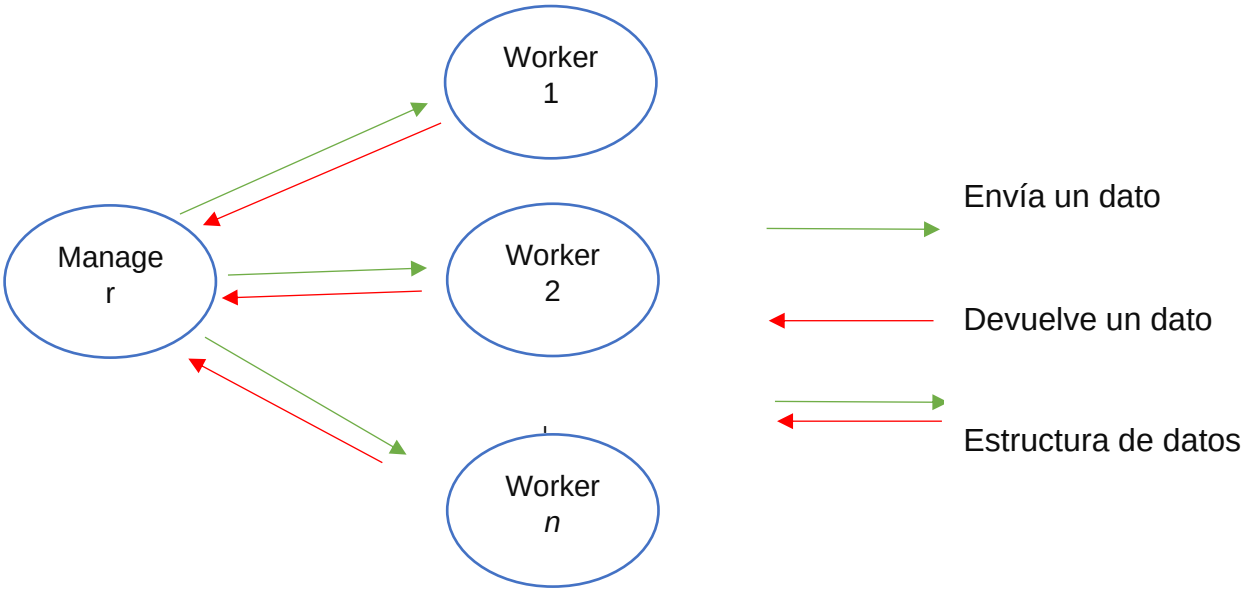


Figura 5 Esquema Manager-Workers

Planificación del Manager-Workers

Contexto

- El problema introduce tareas de una escala que sería no realista o poco eficiente para que otros sistemas manejen y faciliten una solución que implica paralelismo.

- La plataforma paralela, así como el entorno de programación a utilizarse ofrecen un ajuste moderado para el problema y un adecuado nivel de paralelismo en términos del número de procesadores o ciclos paralelos disponibles.

- El lenguaje de programación está determinado y el compilador está disponible para la plataforma paralela.

Problema

- La misma operación debe ser llevaba a cabo reiteradamente sobre todos los elementos de un conjunto de datos ordenado. A pesar de esto, los datos pueden ser procesados sin un orden propio. Sin embargo, es crucial preservar el orden de los datos.

Requerimientos

- El orden de los datos debe preservarse.
- La operación puede llevarse a cabo de forma independiente sobre diferentes agrupaciones de datos.
- La solución debe escalar sobre la cantidad de elementos de procesamiento.
- El mapeo de los elementos de procesamiento a procesadores debe tomar en cuenta la interconexión de la plataforma de hardware.

Solución del problema

Introducir paralelismo de actividad para procesar múltiples conjuntos de datos al mismo tiempo. El patrón Manager-Workers es una representación flexible a esta solución. La estructura del patrón está compuesta por un componente de administración (manager) y un grupo de componentes idénticos de trabajo (Workers). El manager es el responsable de preservar el orden de los datos. Cada worker lleva a cabo el mismo procesamiento en distintos conjuntos de datos independientes. Repetidamente los workers buscan una tarea a realizar, lo cual

repiten hasta que no hay más tareas. Entonces, el programa termina. Si las tareas son distribuidas en tiempo de ejecución, la estructura presenta naturalmente balanceo de carga, puesto que mientras que un worker está ocupado con una tarea grande, otros pueden estar realizando tareas más pequeñas. La distribución de las tareas en tiempo de ejecución hace frente al hecho que los conjuntos de datos pueden presentar distintos tamaños. Por otra parte, el manager, se ocupa de preservar la integridad de los datos, pues monitorea que partes de los datos ha sido han sido procesados y cuales permanecen a la espera de ser procesados por los workers. De forma opcional, el manager puede ser un componente activo al lidiar con el particionamiento y recolección de los datos, permitiendo que dichas tareas se realicen de forma concurrente mientras recibe las solicitudes de datos de los workers. De manera que las operaciones del manager necesitan capacidades de sincronización y bloqueo [3].

Estructura

El patrón Manager-Workers está compuesto de un manager y uno o más workers. Los workers actúan como el elemento de procesamiento. Usualmente, solo existe un manager y varios workers idénticos que realizan procesamiento de forma simultánea en tiempo de ejecución. Como se ha mencionado, en este patrón la misma operación es aplicada por los workers de forma simultánea a distintos conjuntos de datos. Conceptualmente, los workers tienen acceso a diferentes conjuntos de datos y operaciones en cada componente de procesamiento.

La solución radica en considerar una estructura en la que el manager centraliza la distribución de datos entre los workers, logrando con ello la preservación del orden de los datos y los resultados. De manera que la solución se presenta como red centralizada, donde el manager es el componente común central.

Participantes

- Manager. Las responsabilidades del manager son crear una cantidad de workers, particionar trabajo entre ellos, echar a andar la ejecución, calcular el resultado general a partir de los sub resultados obtenidos por los workers.
- Workers. La responsabilidad del worker es buscar una tarea y realizar su procesamiento bajo la forma del conjunto de operaciones requeridas.

Dinámica

Los pasos para realizar un conjunto de cálculos son:

- Todos los participantes son creados y esperan hasta que un cómputo es requerido por el manager. Cuando el dato está disponible, el manager lo divide y envía cada conjunto a la solicitud de cada worker.
- Cada worker recibe el dato y comienza el procesamiento. Esta operación es independiente de las operaciones del resto de los workers. Cuando el worker finaliza el procesamiento, este regresa el resultado al manager y solicita más datos. Si aún permanecen datos a ser procesados, el proceso se repite.
- El manager usualmente responde las solicitudes de datos de los workers y recibe sus resultados parciales. Una vez que todos los conjuntos de datos han sido procesados, el manager ensambla un resultado total a partir de los resultados parciales, y es entonces cuando el programa finaliza [3].

2.6 Frameworks en Java

Un framework es un agregado de herramientas, librerías y componentes que proporcionan una estructura y abstracciones de alto nivel para favorecer el desarrollo de aplicaciones o sistemas. En otras palabras, es un marco de trabajo que define la arquitectura básica y las pautas para el desarrollo de software.

Los frameworks proporcionan un cimiento sobre la cual los desarrolladores pueden construir sus aplicaciones, ya que ofrecen una estructura predefinida, un conjunto de funcionalidades comunes y una forma de interactuar con el entorno de desarrollo. Esto ayuda a agilizar el proceso de desarrollo, promueve las mejores prácticas y permite una mayor reutilización de código. A continuación, se describen sus principales características en la Tabla 4.

Abstracción y reutilización de código	Brindan componentes y librerías predefinidas, que se pueden emplear para desarrollar aplicaciones sin tener que escribir todo el código desde cero. Esto ahorra tiempo y promueve la reutilización de código.
Automatización de tareas comunes	Proporcionan herramientas y utilidades para automatizar tareas repetitivas o tediosas.
Manejo del flujo de trabajo	Ofrecen funcionalidades como enrutamiento de solicitudes, manejo de sesiones, autenticación, autorización, entre otros.

Extensibilidad	Los desarrolladores pueden agregar o personalizar funcionalidades según las necesidades de su aplicación.

Tabla 4 Características de un framework.

2.7 Fork-Join

Fork-Join es un framework para facilitar la programación paralela y concurrente en lenguajes de programación como Java [15]. Fue introducido en Java en la versión 7 como parte del paquete java.util.concurrent. El concepto detrás del modelo Fork-Join se basa en la idea de dividir un problema grande en subproblemas más pequeños y luego combinar los resultados de esos subproblemas para obtener el resultado final. Esto se conoce como el principio de "dividir y vencerás". Se basa en los siguientes elementos:

1. Tarea (Task): Es la unidad básica de trabajo. Una tarea representa un subproblema que se puede ejecutar de forma independiente.
2. Trabajo (Work): El trabajo se divide en tareas más pequeñas, de manera recursiva, hasta que las tareas sean lo suficientemente pequeñas para ser ejecutadas directamente. Esto se conoce como "división" (forking) del trabajo.
3. Combinación (Join): Una vez que todas las tareas han sido ejecutadas, los resultados se combinan o se sincronizan para obtener el resultado final. Esto se conoce como "unión" (joining) de los resultados.

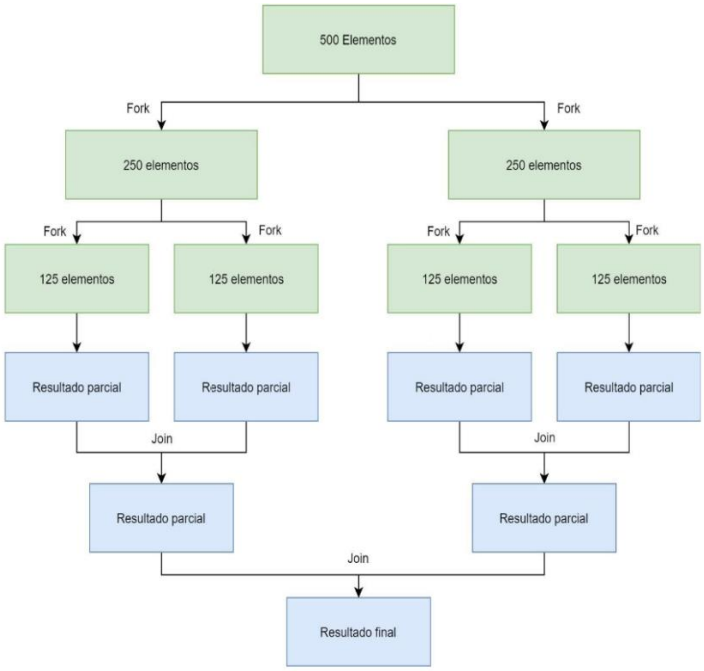


Figura 6 Ejemplo del funcionamiento de Fork-Join

Pasos de trabajo que realizan:

1. Se divide el trabajo inicial en tareas más pequeñas y se programa la "división" del trabajo. Si el trabajo es lo suficientemente pequeño, se ejecuta de primera mano.
2. Las tareas se distribuyen a través de un grupo de hilos (pool de hilos) llamado "ForkJoinPool", que es el encargado de administrar la ejecución de las tareas.
3. Cada hilo del pool de hilos toma una tarea libre y la ejecuta de forma concurrente.
4. Si una tarea se divide en tareas más pequeñas, estas nuevas tareas se añaden al pool de hilos para su ejecución.
5. A medida que las tareas se finalizan, los resultados se combinan (ya sea mediante sumas, concatenación, etc.) hasta obtener el resultado final.

Aprovecha la habilidad de los procesadores contemporáneos para ejecutar múltiples hilos al mismo tiempo y hace uso de la recursión para descomponer las tareas en partes más diminutas. Esto conlleva a una mayor utilización de los recursos y puede acelerar la resolución de problemas que son adecuados para el enfoque en paralelo.

Java proporciona la clase ForkJoinPool y la interfaz ForkJoinTask para implementar el modelo Fork/Join. Además, la clase RecursiveTask<T> se utiliza para tareas que devuelven un resultado y la clase RecursiveAction se utiliza para tareas que no devuelven un resultado. Algunos otros lenguajes de programación donde se introdujo fork-join:

- C++: A partir de la versión C++11, el estándar de C++ introdujo la biblioteca <future> que incluye una interfaz para la programación fork-join utilizando la clase std::async y std::future.
- C#: El lenguaje C# también proporciona soporte para la programación fork-join a través de la biblioteca Task Parallel Library (TPL). La clase System.Threading.Tasks.Parallel en C# permite paralelizar tareas utilizando la técnica fork-join.
- Scala: Scala es un lenguaje de programación que se ejecuta en la plataforma Java Virtual Machine (JVM) y ofrece soporte nativo para la programación fork-join. La biblioteca estándar de Scala proporciona la clase scala.concurrent.forkjoin.ForkJoinTask para implementar este modelo.

La adopción de la técnica fork-join puede resultar beneficiosa en situaciones donde una tarea puede fragmentarse en subtareas más reducidas que tienen la capacidad de operar de manera aislada. Esta técnica gestiona automáticamente la distribución de estas subtareas a través de hilos y la coordinación de los resultados. Además, el marco de trabajo puede ajustarse automáticamente al número de hilos disponibles y a la cantidad de núcleos presentes en el sistema.

2.7.1 Evolución de Fork-Join

La evolución del modelo Fork-Join ha estado estrechamente vinculada con el desarrollo de lenguajes y bibliotecas de programación que admiten la concurrencia y la paralelización. A continuación, se destacan algunas etapas importantes en la evolución de Fork-Join en un cuadro comparativo mostrado en la Tabla 5.

Número de versión Java	Características
7	Esta versión pone en uso el paquete java.util.concurrent que incluía la clase ForkJoinPool y la interfaz ForkJoinTask. Estas clases proporcionaron la base para la programación Fork-Join en Java.
8	Esta versión agregó mejoras significativas al framework Fork-Join en Java. Se introdujeron nuevos métodos en la clase ForkJoinTask, como join(), fork(), invokeAll(), que favorecieron la creación y ejecución de tareas Fork-Join. Asimismo, se agregaron métodos de fábrica para crear instancias de tareas.
9	En esta versión, se realizaron mejoras suplementarias en la implementación de Fork-Join. Se introdujo la clase ForkJoinPool.CommanPool() para obtener una instancia compartida de ForkJoinPool con configuraciones predefinidas. También se agregaron métodos como newWorkStealingPool() y newWorkStealingPool(int

	parallelism) para crear instancias de ForkJoinPool optimizadas para el robo de trabajo.

Tabla 5 Evolución de Fork-Join

Esas son algunas de las versiones clave de Java que han influido en la implementación y las capacidades de Fork-Join. Sin embargo, hay que tener en cuenta que la evolución y mejora de Fork-Join en otros lenguajes o entornos de programación pueden seguir diferentes cronologías y tener características diversas.

Es importante evaluar y ajustar el uso de Fork-Join en función de las características de la aplicación y el entorno de ejecución para obtener el mejor rendimiento posible.

2.8 Comparación Fork-Join versus Manager-Workers

En la *Tabla* 6, se presenta una comparación de las características del patrón de diseño paralelo Fork-Join con respecto al patrón Manager-Workers.

	FORK-JOIN	**MANAGER-WORKERS**
Dependencia de tareas	○ Las tareas suelen ser interdependientes, la tarea principal se divide en sub-tareas y la ejecución de cada sub-tarea puede depender de los resultados de otras sub-tareas.	○ Las tareas suelen ser independientes entre sí, el gerente asigna tareas individuales a los trabajadores y pueden ejecutarlas simultáneamente sin requerir comunicación o coordinación entre ellos.
Granuralidad del trabajo	○ Especialmente útil para problemas que exigen un alto nivel de paralelismo en un nivel de grano fino.	○ Especialmente útil para problemas que se pueden dividir en partes relativamente grandes e independientes.
Control de flujo	○ Sigue un control de flujo más explícito, donde se	○ El gerente decide que tareas asignar a los

	maneja automáticamente la división recursiva de las tareas, administra la sincronización y unión de sub-tareas.	trabajadores y los trabajadores ejecutan las tareas a medida que se les asignan.
Comunicación y coordinación	○ Es posible que las sub-tareas deban comunicarse y sincronizar sus resultados durante la fase de unión. Esta comunicación puede agregar gastos generales, pero es necesaria para combinar los resultados parciales en el resultado final.	○ Los trabajadores no se comunican entre sí durante la ejecución, el gerente es el responsable de recoger los resultados de los trabajadores.

Tabla 6 Manager-Workers vs Fork-Join comparativa

Una vez que se han descrito el marco teórico para un mejor entendimiento de este trabajo de tesina, en el siguiente capítulo se introduce el trabajo que se propone.

Capítulo 3

Implementación paralela de la multiplicación de matrices

El uso de la multiplicación de matrices en aplicaciones tales como la resolución de sistemas de ecuaciones de muchas variables, el cálculo numérico y el cálculo de microarreglos en diversas áreas de estudio es de gran importancia. Cuando se manejan matrices de dimensiones grandes, la cantidad de cómputo que se requiere suele ser muy amplia. Adicionalmente, cuando se usan datos en notación de punto flotante, puede comprometer recursos como la memoria y el manejo interno de los mismos en función del lenguaje que se utiliza para su implementación. Por lo que un diseño adecuado de la implementación de algoritmos paralelos para su resolución es fundamental.

En este trabajo de tesina, nos enfocamos en la implementación en paralelo de la multiplicación de matrices haciendo uso del lenguaje de programación Java y conseguir mayor velocidad en su proceso de resolución cuando se manejan datos en notación de punto flotante. Aunque existen diversos diseños, aquí nos enfocamos en el uso del framework Fork-Join (FJ) y el patrón de diseño Manager-Workers (MW) para la implementación en paralelo. Así, se desarrollan tres implementaciones: una secuencial, una implementación paralela usando FJ y otra usando MW.

Con base en estas implementaciones, se presenta un análisis de los resultados obtenidos cuando se consideran diferentes dimensiones de matrices. Para ellos se consideran diferentes números de procesadores y dos métricas de medición, aceleración (SpeedUp) y eficiencia, con la finalidad de aplicar el concepto de paralelismo en el lenguaje Java y así, conseguir mayores velocidades en el proceso de resolución del problema.

3.1 Implementaciones

En esta sección, se presentan los pasos a seguir para la construcción de una solución paralela y de esta manera, realizar la comparativa entre una implementación en java del framework Fork-Join y una respectiva utilizando Manager-Workers, con el fin de analizar la eficiencia de ambos casos dentro de un mismo escenario.

Se tiene como problema, la multiplicación de matrices cuadrada, compuesta por números de notación flotantes, con una dimensión “nxn”, definida por el usuario; por otra parte, tenemos dos métodos de solución, por un lado, una implementación del patrón de software Manager-Workers y por el otro una implementación del framework Fork-Join, que como se logra apreciar en el cuadro 4, trabajan de distinta manera el problema presentado.

Para llevar a cabo los objetivos planteados en este trabajo de tesis, la metodología que se propone son tres implementaciones:

- Una implementación secuencial o serial.
- Una implementación paralela haciendo uso del framework Fork-Join.
- Una implementación paralela con base en el patrón de diseño de software Manager-Workers.
- Para todas las implementaciones se realizan estudios considerando matrices de diferentes dimensiones.

3.2 Implementación utilizando Fork-Join

Retomando un poco el framework Fork-Join en Java proporciona una forma conveniente de realizar programación paralela, se basa en dos conceptos clave: Fork (división) y Join (unión). Funciona de la siguiente forma:

División (Fork):

- La tarea principal se divide en subtareas más pequeñas hasta que alcanzan un tamaño que pueda resolverse de manera directa.
- Cada subtarea es ejecutada por un hilo diferente.

Ejecución Paralela:

- Las subtareas se envían a un conjunto de hilos conocido como ForkJoinPool.
- El ForkJoinPool distribuye las subtareas entre los hilos disponibles.

Unión (Join):

- Una vez que todas las subtareas han sido completadas, se espera que los resultados de estas subtareas se unan para producir el resultado final.

La forma en que funciona Fork-Join para este caso en específico de matrices es la siguiente:

El usuario asigna la dimensión de la matriz cuadrada nxn

```
1 System.out.println("Dimensión de la matriz: ");
2 int size = Integer.parseInt(scanner.nextLine());
```

División de la tarea principal:

Lo que hace Fork-Join posteriormente que se le asigno una dimensión de matriz es dividir la dimensión inicial en subtareas o pedazos más pequeños, si al hacer la primera división la tarea no es lo suficientemente de granularidad fina vuelve a dividir la tarea hasta que lo sea. A esta tarea principal se le conoce como tarea padre. Si la dimensión de la matriz es relativamente pequeña hace el cálculo de forma directa (línea 1).

Fork:

La matriz se divide en subtareas más pequeñas a cada subtarea se le asigna un hilo (línea 14).

Ejecución de la multiplicación:

Las subtareas ya asignadas al hilo correspondiente se ejecutan de forma paralela, las partes se ejecutan de forma paralela por medio de un pool de hilos, donde una parte espera a la otra a que termine.

Join:

Cuando todos los procesos terminan y los resultados están listos ocurre la unión y con ello el resultado final. Si una subtarea se divide en más subtareas, el proceso de división y combinación se repite hasta que todas las tareas estén completas.

Finalización de la multiplicación:

Una vez que todas las subtareas han terminado y los resultados se han combinado, para obtener el resultado de la matriz, la tarea padre puede continuar con su ejecución o terminar si no hay más trabajo que hacer (línea 22).

Finalización del Proceso:

Con la tarea completada, el proceso puede finalizar, en este caso en específico finaliza mostrando el tiempo de ejecución, mas no el resultado de la multiplicación, que, aunque pueden mostrarse, no es del interés saber el resultado de esta.

En lo siguiente, se presenta la implementación en Java del proceso descrito.

```
protected void compute() {

if (rowEnd - rowStart <= THRESHOLD)
for (int i = rowStart; i < rowEnd; i++)
for (int j = colStart; j < y[0].length; j++)  {
             result[i] = new BigDecimal[x.length];
             BigDecimal tmp = BigDecimal.ZERO;
for (int k = colStart; k < y[0].length; k++)
             tmp = tmp.add(x[i][k].multiply(y[k][j]));
             result[i][j] = tmp;
        }
else  {

     // divide la dimensión en dos partes

     int midRow = (rowStart + rowEnd) / 2;
     RecursiveAction topHalf = new Task(x, y, result,
     rowStart, midRow, colStart, colEnd);
     RecursiveAction bottomHalf = new Task(x, y, result,
     midRow, rowEnd, colStart, colEnd);

     // combinación de resultados

     invokeAll(topHalf, bottomHalf);
         }
         }

```

3.3 Implementación utilizando Manager-Workers

Retomando un poco el funcionamiento del patrón de software Manager-Workers es una técnica de programación que se utiliza para distribuir y coordinar la ejecución de un conjunto de tareas en un sistema. Funciona de la siguiente manera:

Manager:

- Es responsable de coordinar y controlar el flujo de trabajo.
- Recibe la tarea o conjunto de tareas a realizar y las divide en unidades manejables.
- Asigna estas unidades a los Workers para su procesamiento.
- Gestiona la comunicación y sincronización entre los Workers y toma decisiones sobre cómo asignar y distribuir tareas.

Workers:

- Los Workers son los encargados de realizar las tareas asignadas por el Manager.
- Cada Worker ejecuta su tarea de manera independiente y concurrente.
- Una vez que han completado su tarea, informan al Manager sobre el resultado.

Finalización y Resultados:

- Una vez que todos los Workers han completado sus tareas, el Manager recopila y combina los resultados según sea necesario.

La forma en que funciona Manager-Workers para este caso en específico de matrices es la siguiente:

El usuario asigna la dimensión de la matriz cuadrada nxn

```
1 System.out.println("Dimensión de la matriz: ");
2 int size = Integer.parseInt(scanner.nextLine());
```

División de Tareas:

El Manager recibe la tarea en este caso la dimensión de la matriz a multiplicar (línea 3).

Creación de los Workers:

El Manager crea un grupo de Workers. Estos serán los encargados de realizar las multiplicaciones de la matriz por secciones (línea 7).

Asignación de Tareas:

El Manager divide la dimensión de la multiplicación en partes más pequeñas y asigna a cada Workers una parte.

Ejecución Concurrente:

Los Workers ejecutan su sección de la matriz a multiplicar de manera concurrente e independiente. Cada Worker puede ejecutar su tarea en su propio hilo (línea 41).

Comunicación:

Durante la ejecución, los Workers pueden comunicarse con el Manager si es necesario, por ejemplo, para informar sobre el progreso, solicitar más trabajo (en caso de que hayan terminado la tarea asignada previamente) o reportar errores.

Sincronización y Espera:

El Manager espera a que todos los Workers completen su sección de la matriz a multiplicar. Esto implica un mecanismo de sincronización para asegurarse de que todos los Workers han terminado antes de continuar.

Recolección de Resultados:

Una vez que todos los Workers han completado su sección, el Manager recopila los resultados producidos por cada Worker.

Combinación y Presentación de Resultados:

El Manager combina los resultados obtenidos por cada Worker, para obtener el resultado final de la multiplicación de la matriz (línea 17).

Finalización del Proceso:

Con la tarea completada, el proceso puede finalizar, en este caso en específico finaliza mostrando el tiempo de ejecución, mas no el resultado de la multiplicación, que, aunque pueden mostrarse, no es del interés saber el resultado de esta.

En el código siguiente se muestra la implementación en Java del proceso descrito.

```
public class ManagerWorker {

    public BigDecimal[][] multiply(BigDecimal[][] x, BigDecimal[][] y)
{
        int filasA = x.length;
        int columnasB = y[0].length;
        BigDecimal result[][] = new BigDecimal[filasA][columnasB];

        Thread [] poolWorkers = new Thread[filasA];

        for (int i = 0; i < filasA; i++) {
            poolWorkers[i] = new Thread(new Worker(x, y, result, i));
            poolWorkers[i].start();
        }


        for (int i = 0; i < filasA; i++) {
            try {
                poolWorkers[i].join();
            } catch (InterruptedException e) {
            }
        }
        return result;
    }

    private class Worker implements Runnable {

        private final BigDecimal [][] x;
        private final BigDecimal [][] y;
        private final BigDecimal [][] result;
        private final int row;

public Worker(BigDecimal [][] x, BigDecimal [][] y, BigDecimal [][]
              result, int row) {
            this.x = x;
            this.y = y;
            this.result = result;
```

```
36              this.row = row;
37          }
38
39          // Realiza tarea parcial
40          @Override
41          public void run() {
42              for (int i = 0; i < x[0].length; i++) {
43                  result[row][i] = BigDecimal.ZERO;
44                  for (int j = 0; j < y[row].length; j++) {
45                   result[row][i] =
46                   result  [row][i].add(x[j][i].multiply(y[row][j]));
47                  }
48              }
49          }
50
51      }
52
    }
```

Una vez descritas las implementaciones propuestas. En el siguiente capítulo, se presentan los resultados obtenidos de analizar el desempeño de estos bajo diferentes casos de estudio.

Capítulo 4 Resultados

En este capítulo, se presentan y analizan los resultados del desempeño del framework Fork-Join y el patrón de diseño de software Manager-Workers de la propuesta de multiplicación de matrices con notación científica, con el objetivo de conocer que implementación tiene mejor desempeño en tiempo de ejecución. Se realizan una evaluación serial y otras más de forma paralelas. En ambos casos obtenemos que Fork-Join tiene mejor desempeño en términos de tiempo de ejecución, SpeedUp y eficiencia para matrices pequeñas. Por su parte Manager-Workers presenta mejor desempeño en términos de tiempo y ejecución, SpeedUp y eficiencia para matrices de mayor tamaño. Tratándose de matrices de gran tamaño Fork-Join empeora sus tiempos de ejecución, haciéndolo no factible para este tipo de problemas.

4.1 Características de la plataforma

Para llevar a cabo la simulación de las implementaciones se utilizó un equipo con las siguientes características:

Hardware	Software
Lenovo Y720 Procesador: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz RAM: Samsung 16 DDR4-2400 MHz GPU: NVIDIA GeForce GTX 1060 6 GB	Sistema operativo: Windows 10 Home Single Language versión: 22H2 JDK 8

Tabla 7 Características del equipo de cómputo empleado.

4.2 Dimensiones de las matrices utilizadas

Para llevar a cabo el comparativo que se realiza en este trabajo de tesis, se utilizaron matrices de diferentes dimensiones. Las dimensiones que se utilizaron se muestran en la siguiente tabla:

	Matrices pequeñas		
250	500	750	1000
	Matrices medianas		
2000	3000	4000	5000
	Matrices grandes		
10000	15000	20000	25000

Tabla 8 Dimensiones de las matrices

4.3 Análisis de resultados

Nosotros realizamos pruebas con distintas matrices, como ya se mencionó previamente, las matrices tienen dimensiones desde 250 hasta 25000, para cada una de ellas se hizo una medición de la eficiencia y de la aceleración, entonces, lo que hacemos es comparar los tres tipos de implementaciones para cada una de estas combinaciones de parámetros cuando se utilizan diferentes números de procesadores desde 1 a 8, los resultados de ello se muestran a continuación:

Matriz 250

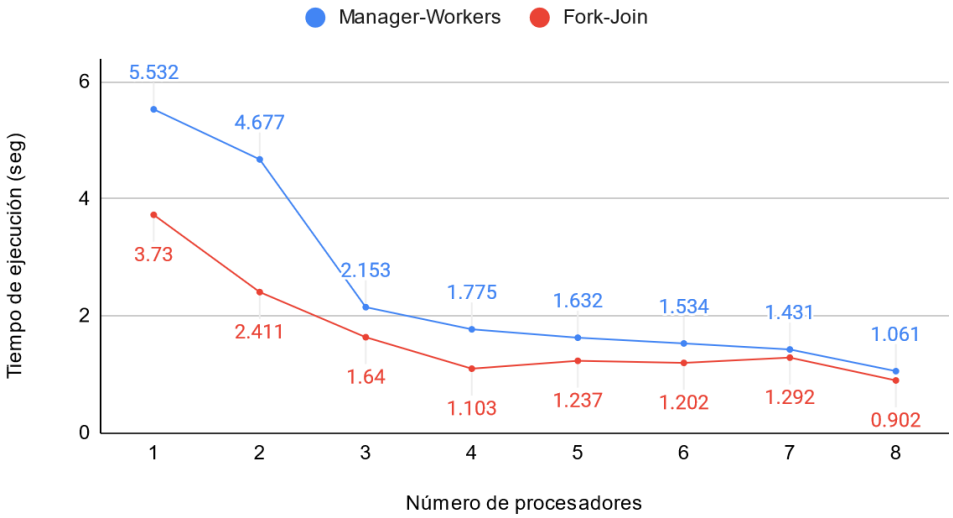


Figura 7 MW vs FJ matriz 250

Segundos de Ejecución		
Procesador	Manager-Workers	Fork-Join
1	5.532	3.730
2	4.677	2.411
3	2.153	1.640
4	1.775	1.103
5	1.632	1.237
6	1.534	1.202
7	1.431	1.292
8	1.061	0.902

Tabla 9 Segundos de ejecución matriz 250

Matriz 500

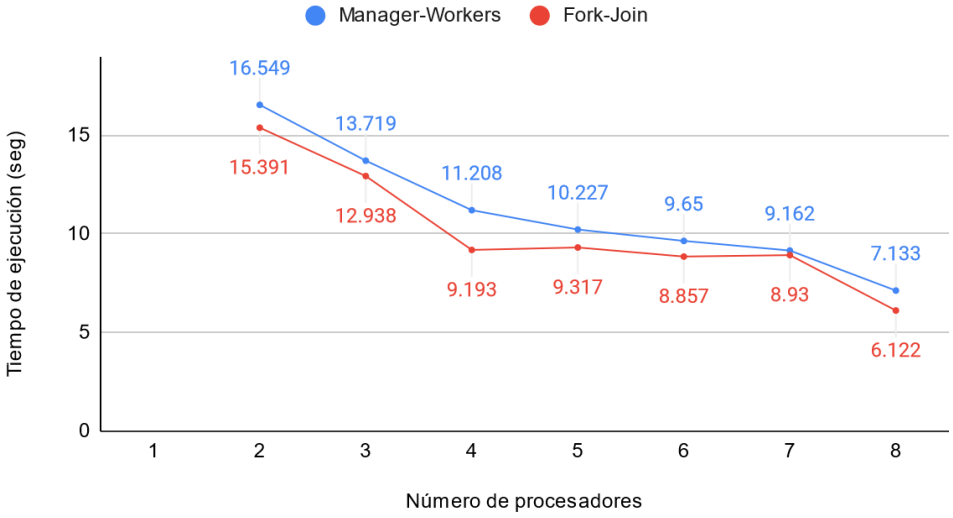


Figura 8 MW vs FJ matriz 500

Segundos de Ejecución		
Procesador	Manager-Workers	Fork-Join
1	29.956	27.811
2	16.549	15.391
3	13.719	12.938
4	11.208	9.193
5	10.227	9.317
6	9.65	8.857
7	9.162	8.93
8	7.133	6.122

Tabla 10 Segundos de ejecución matriz 500

Matriz 750

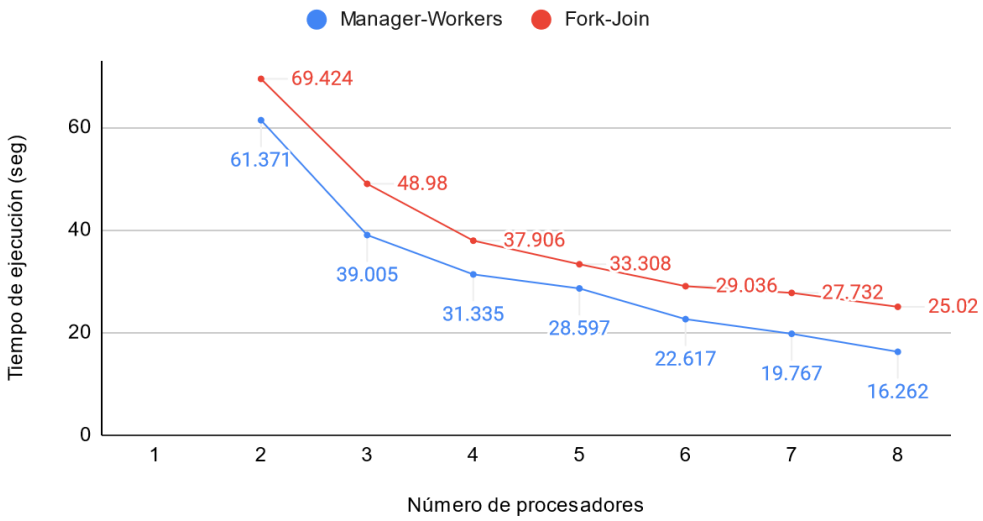


Figura 9 MW vs FJ matriz 750

Segundos de Ejecución		
Procesador	Manager-Workers	Fork-Join
1	89.724	100.523
2	61.371	69.424
3	39.005	48.98
4	31.335	37.906
5	28.597	33.308
6	22.617	29.036
7	19.767	27.732
8	16.262	25.02

Tabla 11 Segundos de ejecución matriz 750

A partir de la matriz con dimensión 1000 se nota una diferencia en los tiempos de ejecución, un mejor desempeño por parte de la implementación de Managers-Workers sobre la implementación basada en Fork-Join. También cabe mencionar que desde el procesador numero 4 hasta el 8 comienza a notarse que el tiempo de ejecución mejora respecto uno del otro.

Matriz 1000

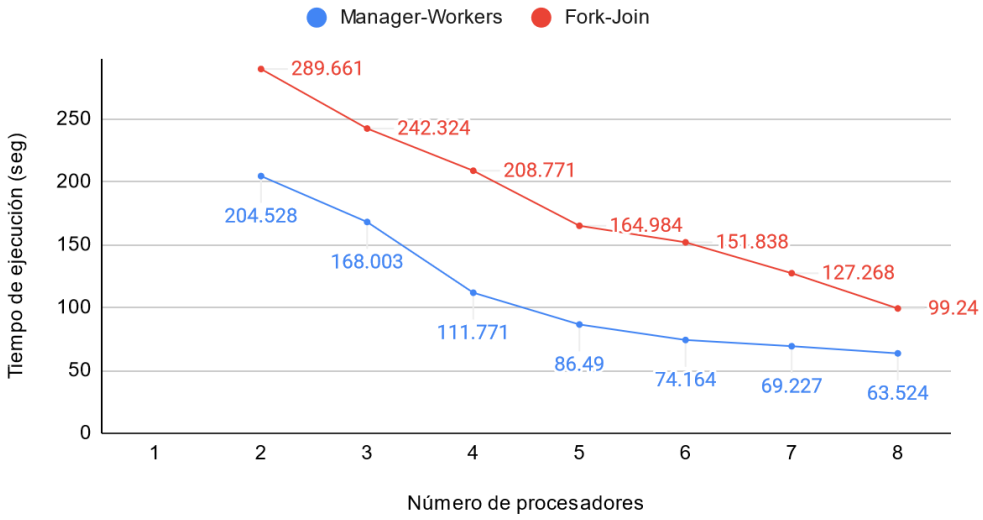


Figura 10 MW vs FJ matriz 1000

Segundos de Ejecución		
Procesador	Manager-Workers	Fork-Join
1	255.626	311.508
2	204.528	289.661
3	168.003	242.324
4	111.771	208.771
5	86.49	164.984
6	74.164	151.838
7	69.227	127.268
8	63.524	99.24

Tabla 12 Segundos de ejecución matriz 1000

A continuación, se muestran los cálculos correspondientes al SpeedUp para las matrices con dimensión 250 hasta 1000.

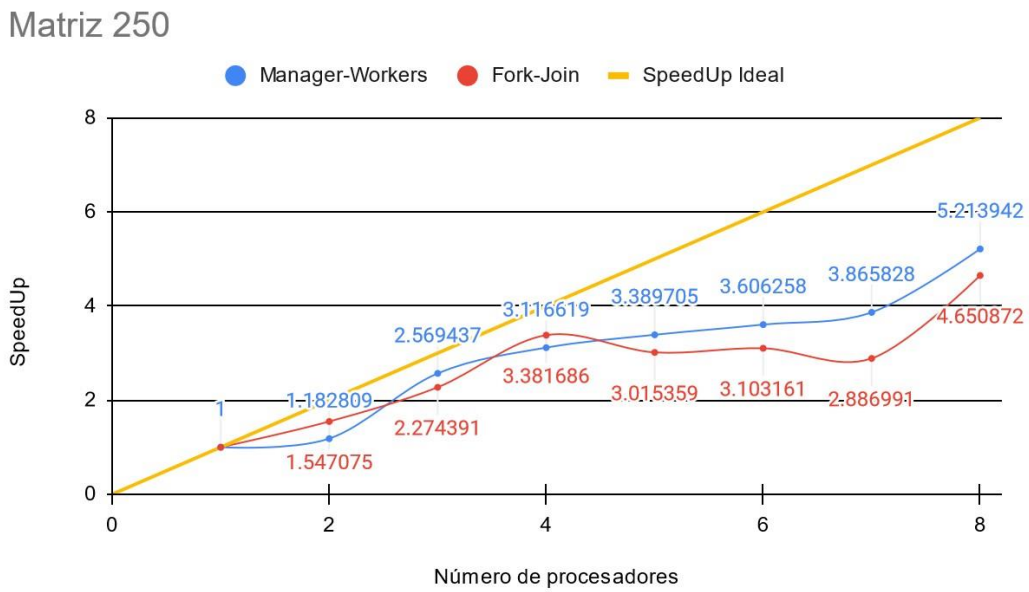


Figura 11 MW vs FJ matriz 250 SpeedUp

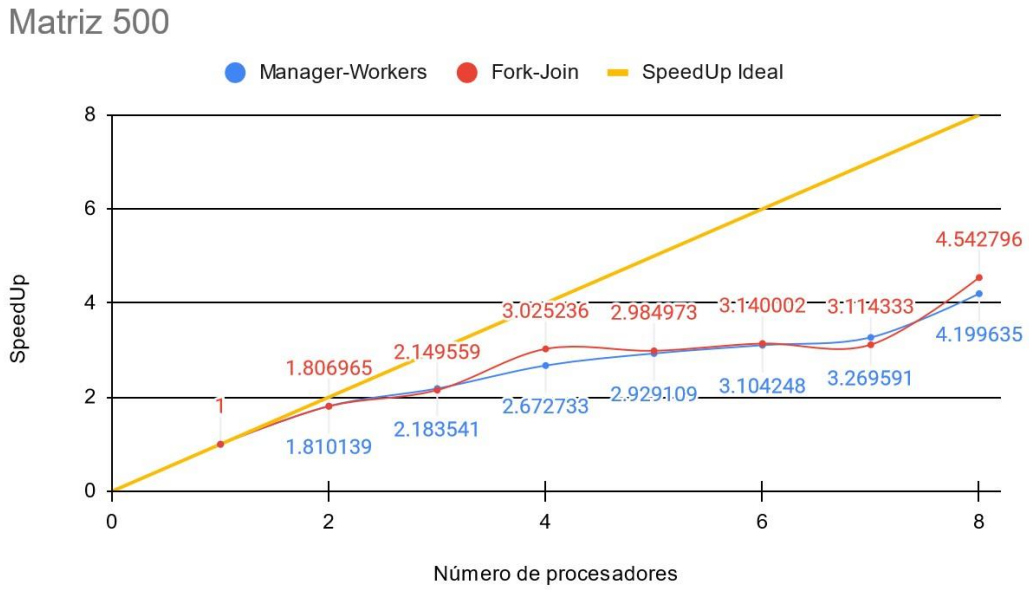


Figura 12 MW vs FJ matriz 500 SpeedUp

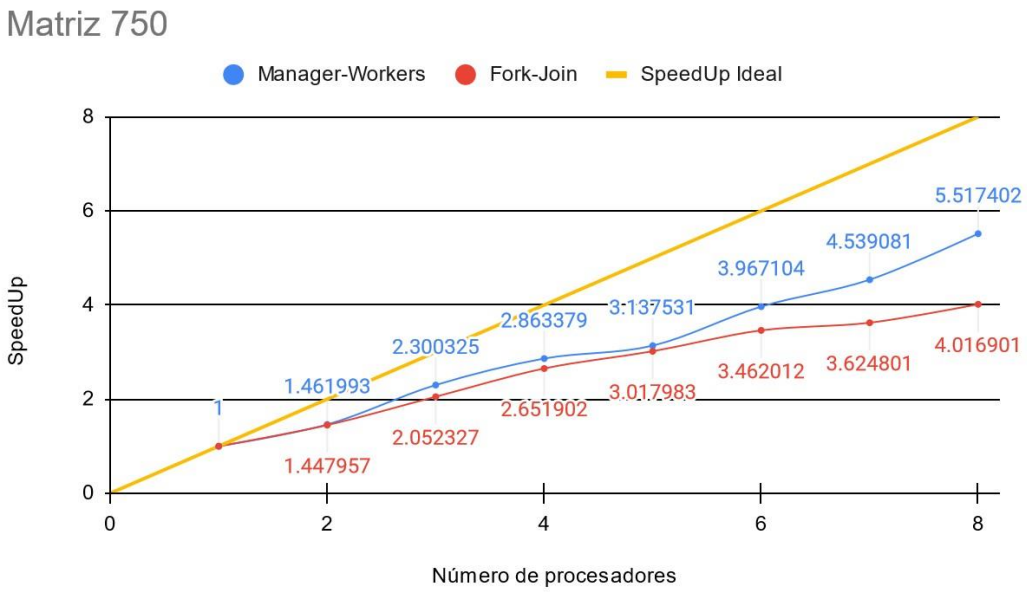


Figura 13 MW vs FJ matriz 750 SpeedUp

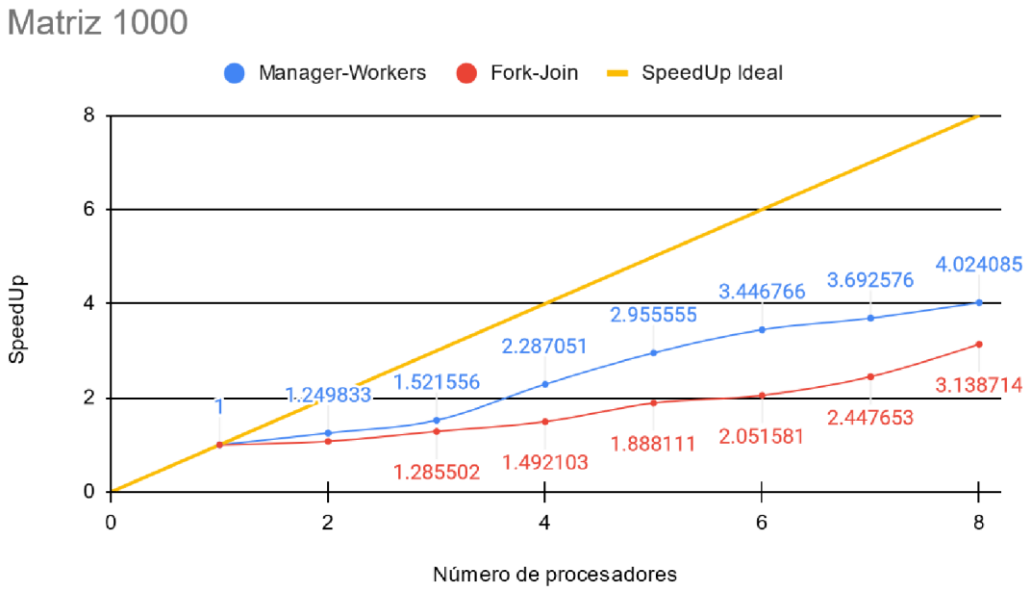


Figura 14 MW vs FJ matriz 1000 SpeedUp

Claramente la implementación de Managers-Workers obtiene mejores resultados sobre la implementación de Fork-Join, se acerca más a lo que es la aceleración ideal, mientras la implementación de Fork-Join medianamente se va alejando.

A continuación, se muestran los cálculos correspondientes a la Eficiencia para las matrices con dimensión 250 hasta 1000.

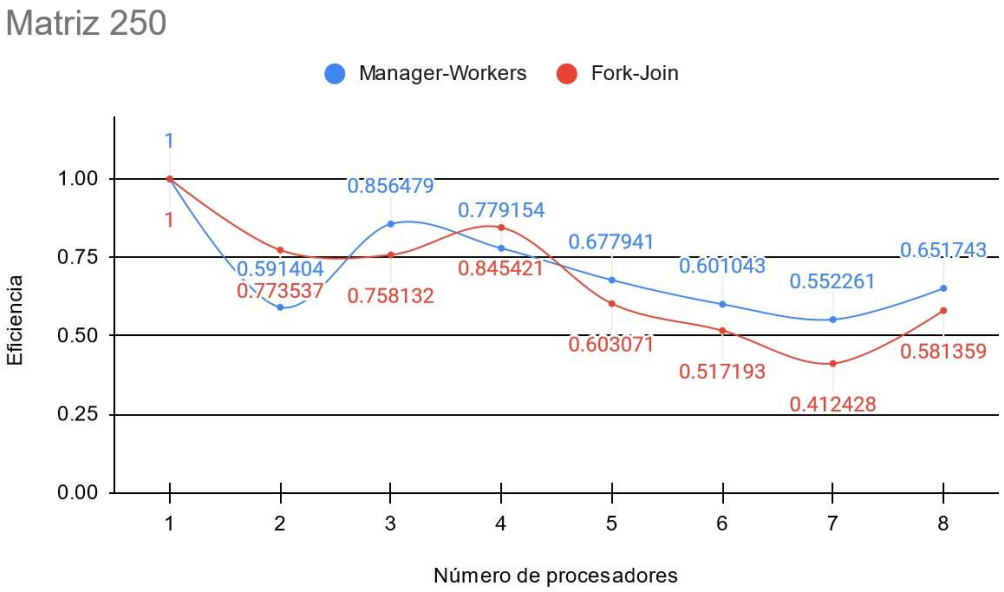


Figura 15 MW vs FJ matriz 250 Eficiencia

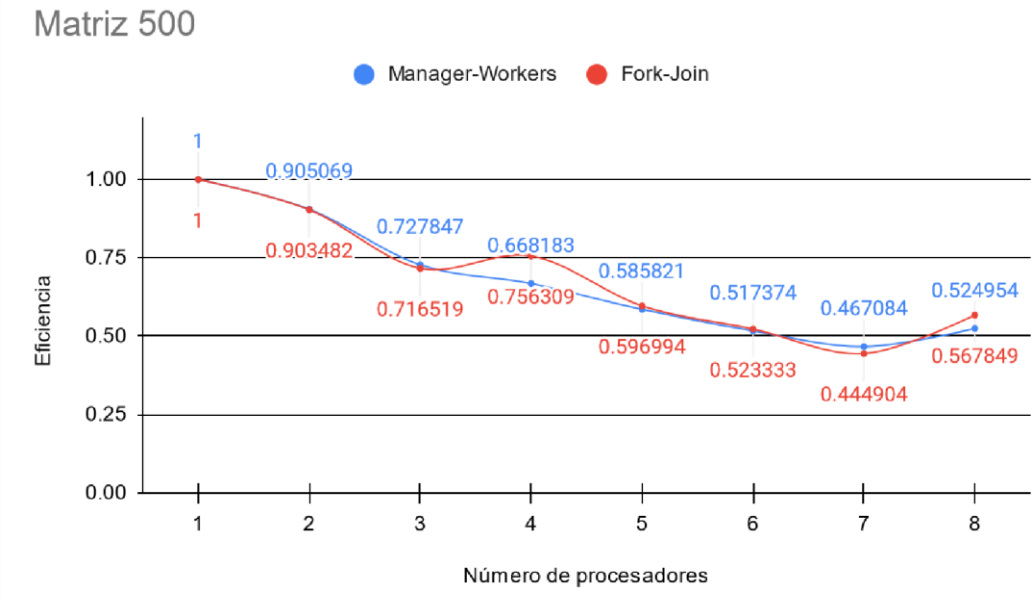


Figura 16 MW vs FJ matriz 500 Eficiencia

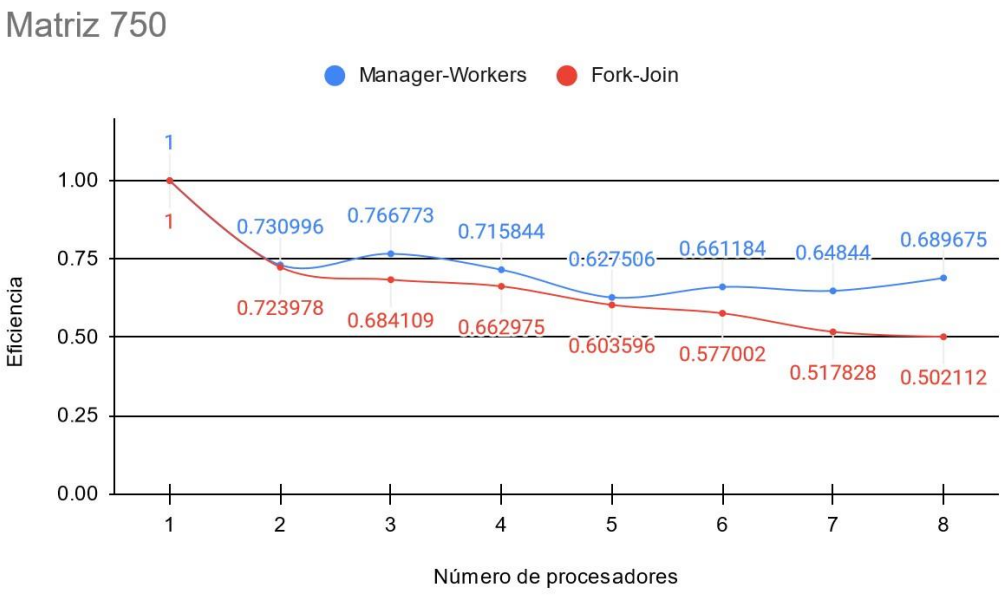


Figura 17 MW vs FJ matriz 750 Eficiencia

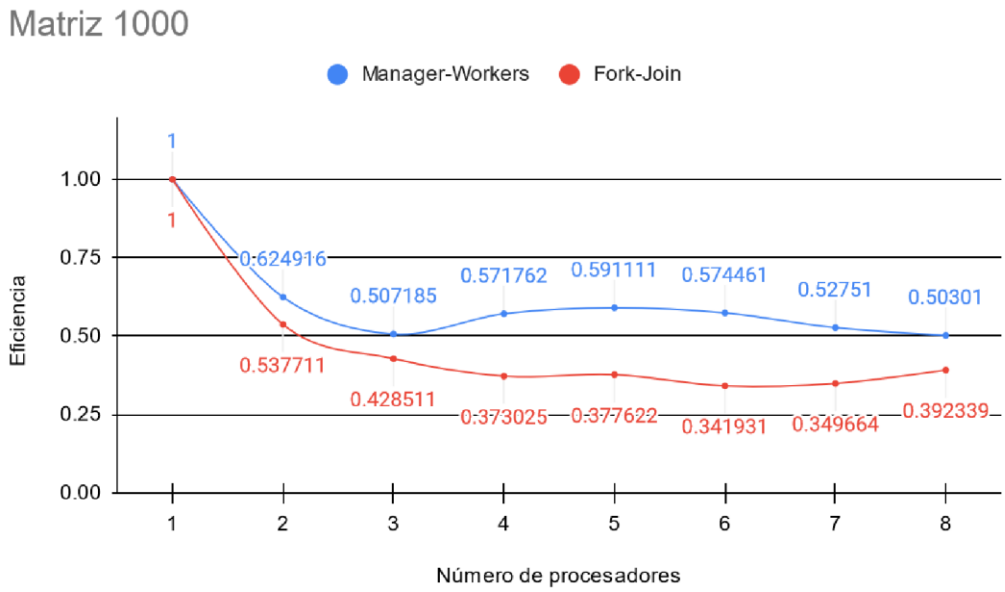


Figura 18 MW vs FJ matriz 1000 Eficiencia

De igual manera que en el SpeedUp, la implementación de Managers-Workers obtiene mejores resultados sobre la implementación de Fork-Join, se acerca más a lo que es el valor 1 en este caso Eficiencia e igualmente a partir de procesador numero 4 las diferencias son más notables.

A continuación, se muestra en la siguiente tabla un resumen de cada una de las implementaciones con sus respectivos tiempos de ejecución, también un promedio del SpeedUp y Eficiencia para cada dimensión.

	Matriz 250		Matriz 500		Matriz 750		Matriz 1000	
Tiempo Serial (seg)	5.532	3.73	29.956	27.811	89.724	100.523	255.626	311.508
Tiempo Paralelo (seg)	1.061	0.802	7.133	6.122	16.262	25.025	63.524	99.247
Hilos Activos	8	8	8	8	8	8	8	8
SpeedUp (promedio)	2.99307	2.73244	2.646124	2.720483	3.098351	2.65924	2.52218	1.79739
Eficiencia (promedio)	0.713735	0.68639	0.674541	0.68867	0.730052	0.65895	0.61249	0.4751

Manager-Workers Fork-Join

Tabla 13 Resumen MW vs FJ matrices pequeñas

En los siguientes resultados, se usaron 8 procesadores, con el fin de ver la efectividad de cada implementación Manager-Workers y Fork-Join con problemas de matrices con dimensión medianas de 2000 a 5000. En estos resultados no se toman en cuenta los tiempos seriales, ya que los tiempos de cómputo que se llevaban eran muy grandes y el tiempo del que se disponía muy corto.

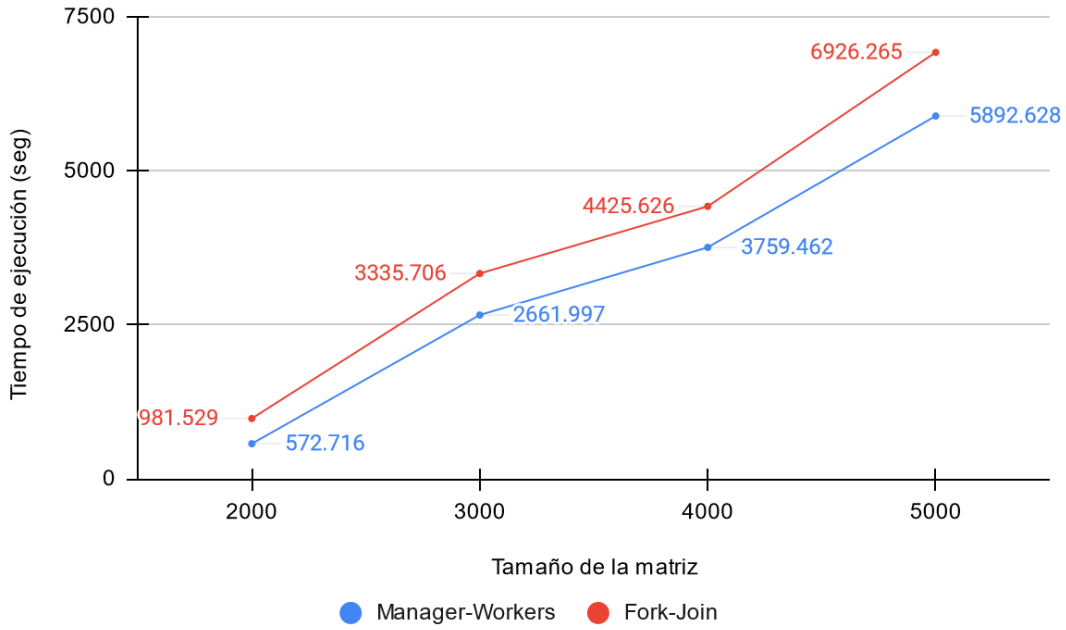


Figura 19 MW vs FJ matrices medianas

En esta grafica la implementación de Managers-Workers obtiene mejores tiempos de ejecución que la implementación de Fork-Join, podemos a preciar tomando como referencia la dimensión 5000 Managers-Workers aventaja poco más de 17 minutos de ejecución, en estas instancias apreciamos que la forma en la que Fork-Join realiza la comunicación comienza a afectar en los tiempos. A continuación, una tabla con un breve resumen.

Tamaño	Manager-Workers	Fork-Join
	Ejecución en segundos	
2000	572.716	981.529
3000	2661.997	3335.706
4000	3759.462	4425.626
5000	5892.628	6926.265
	Ejecución en minutos	
2000	9.545266667	16.3588167
3000	44.36661667	55.5951
4000	62.6577	73.7604333
5000	98.21046667	115.43775

Tabla 14 Resumen MW vs FJ matrices medianas

De igual manera en los siguientes resultados, se usaron 8 procesadores para todos los resultados que se presentan con el fin de ver la efectividad de cada implementación Manager-Workers y Fork-Join con problemas de matrices con dimensión grandes de 10000 a 25000. En estos resultados no se toman en cuenta los tiempos seriales, ya que los tiempos de cómputo que se llevaban eran muy grandes y el tiempo del que se disponía muy corto.

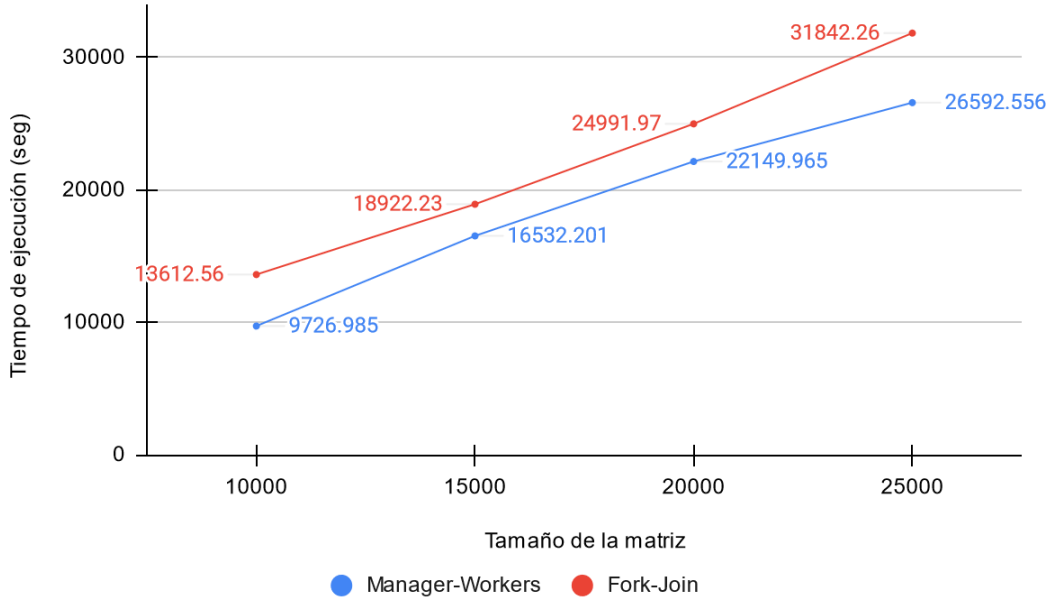


Figura 20 MW vs FJ matrices grandes

En estos resultados podemos ver que los resultados casi son rectos, siempre manteniendo una distancia considerable el uno del otro de nuevo favorecidos para la implementación del Manager-Workers quien en la dimensión más grande el tiempo de ejecución le es favorable y con ello reafirmamos una vez más la hipótesis planteada en un principio.

Tamaño	Manager-Workers	Fork-Join
	Ejecución en segundos	
10000	9726.985	13612.56
15000	16532.201	18922.23
20000	22149.965	24991.97
25000	26592.556	31842.26
	Ejecución en minutos	
10000	162.1164167	226.876
15000	275.5366833	315.3705
20000	369.1660833	416.532833
25000	443.2092667	530.704333

Tabla 15 Resumen MW vs FJ matrices grandes

4.4 Conclusiones de los resultados

En esta sección describimos los resultados de realizar diferentes ejecuciones de las soluciones Manager-Workers y Fork-Join para la implementación de la propuesta de multiplicación de matrices con notación científica.

En tiempo serial y paralelo con matrices pequeñas el framework Fork-Join es mejor, tiene mejores tiempos de ejecución, mejor SpeedUp y eficiencia, en una problemática más grande, sucede todo lo contrario, el patrón de software Manager-Workers es mejor en todos los aspectos. Ya cuando las matrices comienzan a ser de un tamaño considerablemente más grandes, es muy notable la decadencia de Fork-Join.

En la siguiente tabla tenemos un resumen de todos los resultados, agregando una columna donde se maneja el porcentaje donde muestra que tanto es superior uno del otro en cada dimensión de matriz. En la mayoría de los casos la implementación de Manager-Workers es muy favorable en cuestión de tiempo de ejecución.

Tamaño	Manager-Workers	Fork-Join	Porcentajes	
	Ejecución en segundos		Manager-Workers	Fork-Join
250	1.061	0.802		13%
500	7.133	6.122		11%
750	16.262	25.025	15%	
1000	63.524	99.247	15%	
2000	572.716	981.529	17%	
3000	2661.997	3335.706	12%	
4000	3759.462	4425.626	11%	
5000	5892.628	6926.265	11%	
10000	9726.985	13612.56	13%	
15000	16532.201	18922.23	11%	
20000	22149.965	24991.97	11%	
25000	26592.556	31842.26	11%	

Tabla 16 MW vs FJ resumen final

Capítulo 5

Conclusiones

Se realizó un análisis comparativo entre el patrón de software Manager-Workers y el framework Fork-Join para poder conocer cuál de ellos tiene un mejor desempeño en términos de tiempo de ejecución, para la multiplicación de matrices.

Concretamente, el cuestionamiento a responder es cual es mejor bajo un mismo escenario matricial considerando como medida el tiempo de ejecución.

Para responder este cuestionamiento se propuso una solución paralela, donde ambas soluciones se enfrentarían a una misma problemática que es la multiplicación de matrices cuadrada, compuesta por números de notación flotantes random´s, con una dimensión “n”, definida por el usuario.

De acuerdo con nuestros resultados obtenidos , se cuenta con la evidencia para responder que Manager-Workers al ser homogéneo (uniforme en todas sus partes) y liberar memoria, es mejor para matrices grandes, en cambio Fork-Join que está pensado para problemas heterogéneos (componentes que no están uniformemente distribuidos) y este al no liberar memoria, tiene problemas para trabajar con matrices grandes, por eso este es mejor para matrices pequeñas, sin duda alguna nadie es mejor que Fork-Join para problemas pequeños.

Referencias

[1] L. Lemay y C. L. Perkins, Aprendiendo Java en 21 días,, Sams.net, 1996.

[2] G. Hager y G. Wellein, Introduction to High Performance Computing for Scientists and Engineers,, Chapman & Hall/CRC Computational Science, 2011.

[3] J. L. Ortega Arjona, Patterns for Parallel Software Design, 1/E, Wiley Publishing, 2010.

[4] J. O. Ccoplien, «Patterns of Software: Tales from the Software Community,» vol. 6, nº 6, p. 18, 1994.

[5] F. Buschmann, R. Meunier y H. Rohnert, «Pattern-Oriented SoftwareArchitecture.,» *John Wiley & Sons, Ltd., 1996,* pp. A4-1-A4-6, 1996.

[6] H. Schildt, The Complete Reference Java J2SE. 5th edition, McGraw Hill, 2004.

[7] M. C. Chan, S. W. Griffith y A. F. Lasi, 1001 Tips para programas con Java., McGrawHill, 1998.

[8] P. Wilkinson, Parallel programming: techniques and applications using networked workstations and parallel computers, 2/E, Pearson Education India, 2da. Edición, 2006.

[9] J. W. McCormick, F. Singhoff y J. Hugues, Building parallel, embedded, and real-time applications with Ada., Cambridge University Press, 2011.

[10] R. P. Gabriel, Patterns of Software: Tales from the Software Community, Oxford University Press , 1998.

[11] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King y A. Shlomo, A Pattern Language: Towns, Buildings, Construction, New York: Oxford University Press, 1997.

[12] C. Alexander, The Timeless Way of Building, New York: Oxford University Press, 1979.

[13] M. Nash y W. Waldron, Applied Akka patterns: a hands-on guide to designing distributed applications, O'Reilly Media, Inc., 2016.

[14] R. Hiesgen, D. Charousset y T. C. Schmidt, «OpenCL Actors–Adding Data Parallelism to Actor-Based Programming with CAF,» *Programming with Actors: State-of-the-Art and Research Perspectives,* pp. 59-93, 2018.

[15] L. Vito, V. Nicosia y G. Russo, Complex Networks: Principles, Methodsand Applications, Cambridge University Press, 2017.

[16] H. 2. U. C. o. O.-O. T. (. 9. 1. Rohnert, «(Pattern-Oriented} Software Architecture.,» de *2nd USENIX Conference on Object-Oriented Technologies (COOTS 96)*, Toronto Ontario, Canada., 1996.

[17] I. Foster, Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering, Bosto, MA, USA.: Addison-Wesley, 1995.

[18] H. Mostafa, Advanced Computer Architecture and Parallel Processing, John Willey and Sons, 2011.

[19] P. Pacheco, An introduction To Parallel Programming, Morgan Kaufmann, 2011.