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

Evaluación del desempeño del Modelo de Actor en los frameworks CAF y Akka

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

PRESENTA:
Hermilo Cortés González

Director de Tesina:
Dr.Jorge Luis Ortega Arjona
Departamento de Matemáticas, Facultad de Ciencias.

Ciudad Universitaria, Ciudad de México, 18 de enero de 2021.

Agradecimientos

Los meses entre la segunda mitad del 2019 y la primera de 2020 han sido los más retadores y de mayor aprendizaje de mi vida. No sólo por la pandemia que estamos enfrentando, sino también porque tuve la oportunidad de adentrarme en una ciencia increíble como es la computación. Fue descubrir nuevas preguntas y nuevas formas de pensar e imaginar el mundo. Pero sobre todo, fue conocer a profesores y compañeros que me contagiaron con su entrega la pasión por el cómputo.

Cada clase la disfruté y aprendí muchísimo y marcaron el rumbo de lo que quiero ser en el futuro. Muchas gracias a la Dra. Lárraga, Mtra. Yolanda Dr. Ernesto Rubio, Dr. José Luis Gordillo, Dr. Héctor Benítez, Ing. Adrián Chavesti, Dr. Lukas Nellen, quienes son excelentes académicos y sobre todo excelentes personas.

En especial al Dr. Jorge Ortega quien me guió en la elaboración de este proyecto. Dr. Ortega, mil gracias por confiar en mi y darme la oportunidad de trabajar este proyecto con usted, y por ayudarme a darle claridad y estructura a este trabajo. Su método de ordenar las ideas y pensar los problemas será algo enriquecedor en mi vida.

Estoy totalmente convencido que el pilar fundamental para mantenerme y no tirar la toalla fueron mis compañeros. Lety, Isra, Micke, Robert, Toño, mil gracias. Todos ustedes son seres humanos increíbles. Les estaré por siempre agradecido por compartir conmigo sus experiencias, conocimientos y sonrisas.

Sabri, mil gracias por apoyarme siempre.

Mi presente sería completamente distinto si no hubiese cursado la especialidad. Fue el motivo que me brindó la fortaleza mental para enfrentar estos meses tan complicados de pandemia. No tengo dudas que mi presente y mi futuro estará marcado por estos meses. Mil gracias a todos ustedes.

Resumen

El cómputo paralelo se ha vuelto indispensable para responder a retos y problematicas, así como también para aprovechar las tecnologías actuales. Distintas propuestas han sido presentadas para operar en ambientes multiprocesador. Así también, han surgido metodologías para la construcción de algortimos paralelos y de software paralelo.

El *Modelo de Actor* ha ganado relevancia para trabajar en ambientes mulitprocesador, distribuidos y paralelos. El modelo está basado en el concepto de *actor*, el cual es una unidad autónoma (*i.e.* tiene un hilo de control propio) que encapsula datos y métodos e interactúa con otros actores mediante mensajes. Un actor puede crear nuevos actores y de cambiar de estado.

Existen lenguajes de programación basados en el *Modelo de Actor* así como también frameworks o librerías que incorporan elementos del modelo a un lenguaje. Para el primer caso, algunos ejemplos son Erlang, Rebeca , Encore, etc. Por su parte, C++ Actor Framework (CAF) y Akka son frameworks disponibles para C++ y Java/Scala, respectivamente.

Ambos frameworks implementan de forma particular el modelo de actor y, en ocasiones, llegan a diferir de manera importante en determinados aspectos. Cada framework cuenta con características que los hacen más atractivos en función del hardware disponible. Tal es el caso del módulo *Cluster* de Akka que hace más manejable el escalamiento al hacer sencillo el proceso de incorporación de nuevos nodos a un cluster existente. Para el caso de CAF, facilita el manejo de dispositivos heterogéneos como GPU y otros aceleradores.

El fin de este trabajo es evaluar si CAF y Akka presentan un desempeño igual o distinto en un ambiente distribuido para un problema dado y un patrón arquitectónico de programación paralela. En estas condiciones, *¿Es posible determinar*

cuál de estos frameworks tiene un mejor desempeño, en términos de tiempo de ejecución, en un cluster?. Para responder a esta pregunta, se propone un análisis comparativo mediante la implementación del patrón arquitectónico *Manager-Workers* en un cluster para el cálculo del coeficiente de clustering de los nodos de una red no dirigida.

Se implementó la solución con los frameworks CAF y Akka. Para evaluar el desempeño de ambos, realizamos una prueba a nivel local y otra de forma distribuida. Para la primera obtenemos que la implementación en Akka tiene un mejor desempeño en términos de tiempo ejecución, además de presentar speedup y eficiencia superior a la implementación en CAF. Para la prueba distribuida los resultados muestran que la implementación en CAF presenta un mejor desempeño tanto en tiempo de ejecución, speedup y eficiencia cuando agregamos un nodo worker adicional, mientras que en Akka empeora los tres indicadores.

De acuerdo a nuestros resultados, se cuenta con evidencia para responder que CAF muestra un mejor desempeño en un cluster en términos de tiempo de ejecución para la propuesta de cálculo del coeficiente de clustering de una red no dirigida.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Contexto

Dadas la tecnologías actuales así como los volúmenes de datos cada vez mayores, la programación paralela se vuelve indispensable para dar respuesta a los problemas y retos.

Se han desarrollado tecnologías para operar en ambientes multiprocesador así como metodologías para la costrucción de algoritmos paralelos y de software paralelo.

Un enfoque que ha ganado relevancia para trabajar en ambientes multiprocesador ha sido el *Modelo de Actor*. Un *actor* representa una unidad autónoma (en el sentido que tiene un hilo de control propio) que encapsula datos y métodos y que interactúa con otros actores mediante mensajes, y tiene la capacidad de crear nuevos actores y de cambiar su estado.

Existen lenguajes de programación basados en el *Modelo de Actor*, tales como Erlang, Rebeca, Encore, entre otros [2]. A su vez, se han elaborado frameworks para incorporar elementos de este modelo a lenguajes de programación. Dos de estos son C++ Actor Framework (CAF) [1], el cual es enteramente implementado en C++, y Akka, que está disponible para Java y Scala.

1.2. Problemática

Evaluar si CAF y Akka presentan un desempeño igual o distinto en sistemas distribuidos para un problema dado.

1.3. Pregunta de investigación

Dada una problemática y un patrón arquitectónico de programación paralela, y considerando los frameworks CAF y Akka ¿Es posible determinar cuál de estos tiene un mejor desempeño en un cluster?

1.4. Aproximación

Se propone realizar un análisis comparativo entre CAF y Akka mediante la implementación del patrón arquitectónico *Manager-Workers* en un cluster para el cálculo del coeficiente de clustering de los nodos de una red no dirigida. La medida de desempeño se considera en términos de tiempo de ejecución en segundos.

Capítulo 2

Antecedentes

En esta sección detalla en qué consiste el modelo de actor. A su vez, se presenta la forma como éste es implementado en Akka y en CAF.

También se explica la metodología de Foster [3] para el diseño de programas paralelos.

A su vez, se analiza qué es un *Patrón Arquitectónico* y cómo se aplica a la *Programación Paralela*. Especialmente, detallamos el patrón *Manager-workers*.

Se detallan las métricas para la evaluación del desempeño de programas paralelos, como el *Speedup* y la *Eficiencia*, y se detalla las Leyes de Amdahl y Gustafson.

Por último, abordamos una problemática en particular que consiste en el cálculo de el coeficiente de clustering para una red no dirigida mediante su representación en matriz de adyacencia.

2.1. El Modelo de Actor

El modelo de actor fue propuesto por Hewitt et al [4] y formalizado por Agha [5]. Agha y Kim [6] señalan que los *actores* son objetos autónomos que encapsulan datos, métodos y una interface, mientras que su autonomía se define al encapsular un hilo de control. Los *actores* interactúan entre sí enviando y recibiendo mensajes. La recepción de estos mensajes inicia la ejecución del método especificado en los argumentos del mensaje. La ejecución de un método es atómica, *i.e.* una vez que un método es ejecutado, no puede ser interrumpido por el mensaje de otro *actor*.

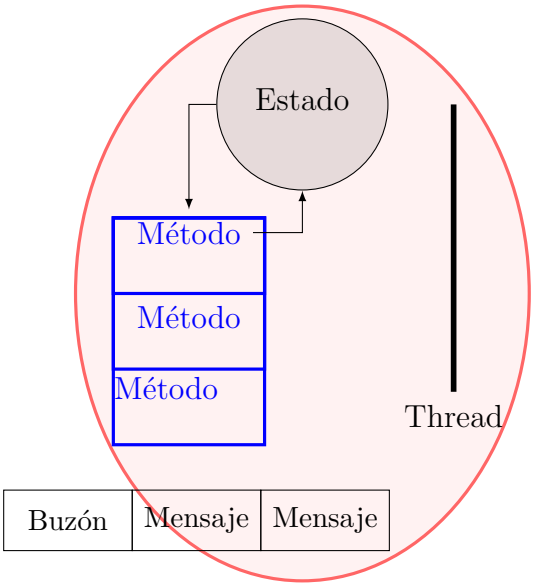


Figura 2.1: Anatomía del Actor
Un actor encapsula un estado, un conjunto de métodos y un thread. Los mensajes enviados a un actor son almacenados en una cola de correos que es identificada por una dirección de correo única.

Esto no impite que multiples hilos se mantengan activos en un actor, quien, como respuesta a un mensaje, puede enviar mensajes, crear actores o realizar cambios a sus datos locales (Ver Figura 2.1 y Figura 2.2). Cabe señalar que tanto los hilos de control como el estado están encapsulados en el propio *actor*.

Cada *actor* cuenta con una *dirección de correo* que representa el espacio lógico en el cual se encuentra. Los actores pueden enviar mensajes a aquellos que conocen su *dirección de correo*, pero esta puede ser enviada en mensajes. El paso de mensajes entre actores es *asíncrono* y *no bloqueante*, mientras que la recepción de mensajes de eventos se realiza de forma desordenada. Los mensajes son almacenados en una cola de recepción de correos, y son procesados uno a la vez, respetando el orden de llegada[7].

Como se mencionó, los *actores* pueden crear una cantidad finita de *actores*, lo cuales se denominan *hijo*, mientras que el *actor* creador se denomina *padre* o *raíz*. El *padre* o *raíz* conoce la dirección de los *hijos*, de manera que pueden enviar mensajes a estos y viceversa. A su vez, el *actor padre* puede enviar a otros *actores*, mediante mensajes, la dirección de sus *hijos*.

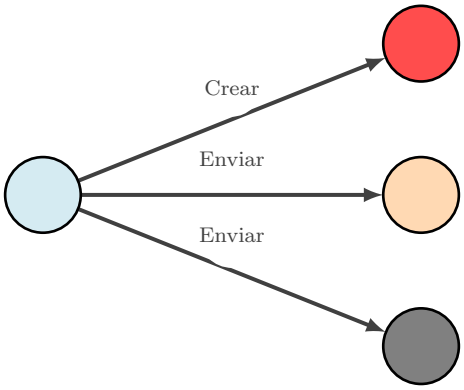


Figura 2.2: Operaciones Primitivas de un Actor
Los actores pueden enviar mensajes, crear nuevos actores y cambiar su estado local.

La creación de *actores* define una jerarquía de *actores* en la cual, a excepción del *actor raíz*, todos tendrán un padre y cualquier actor podrá tener uno o más hijos. La colección de estos acotres, comenzándo por el actor raíz y bajando por la descendencia, se le denomina un *sistema de actor*.

2.1.1. El modelo de actor en Akka

Akka es una biblioteca escalable disponible para Java y Scala que implementa actores utilizando el patrón de objeto activo [2]. Akka está escrito en Scala y hace uso de una de las características clave de este lenguaje: las estructuras de datos inmutables. Estas estructuras son utilizadas por Akka para sus protocolos de mensajes.

En Akka, las los actores pueden ser *locales* o *remotos*. La localidad se entiende si la recepción y envío de mensajes entre actores se realiza en la misma JVM. Independientemente si el actor es local o remoto, los mecanismos de comunicación permanecen sin cambios, de manera que el código no difiere en ambos escenarios. De manera que el código es *distribuido por diseño*, puesto que no hay cambios entre operaciones locales y distribuidas.

Los principales componentes de Akka son los siguientes:

Akka Actor

El Modelo de Actor se implementa con paso de mensajes asíncronos y evitando compartir el estado de los actores.

Akka limita el acceso a los actores. Dado que la comunicación entre actores puede ocurrir únicamente mediante paso de mensajes, no es posible hacer llamados a métodos de los actores. Cuando se crea una instancia de un actor, se regresa un `ActorRef`, el cual representa un *wrapper* de este actor, aislándolo del resto de código. Toda la comunicación con el actor debe realizarse a través de `ActorRef`. Los mensajes a un actor son enviados vía `ActorRef`. El actor receptor procesa los mensajes que le fueron enviados uno a la vez, lo cual genera la ilusión de un *single-thread*, encapsulando el estado del actor de otro *thread*.

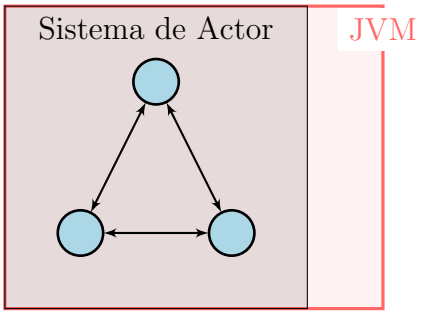
Los cambios de comportamiento de los actores son implementados mediante `become`.

Sistema de actor en Akka

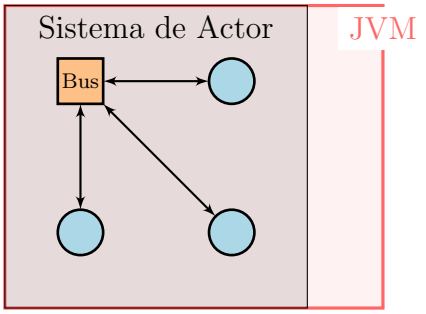
Akka implementa el sistema de actor de manera local y remota. Para el primer caso, los actores se encuentran en una misma Java Virtual Machine (JVM). La muestra un sistema de actor local compuesto por 3 actores en el que la comunicación se realiza punto a punto, de manera que los actores pueden enviar y recibir mensajes entre sí (Ver Figura 2.3a). Akka también proporciona un mecanismo de mensajes uno-a-muchos, mediante la incorporación de un *bus de eventos* (Ver Figura 2.3b). Este *bus de eventos* permite a un actor publicar un mensaje y al resto de los actores suscribirse para recibir este mensaje. Este mecanismo permite desacoplar al actor que envía del que recibe.

Para los sistemas de actor remotos, Akka permite la comunicación de estos sobre una red y la serialización de mensajes, de manera que actores en distintas JVM pueden enviar y recibir mensajes. Esta comunicación se puede realizar por la dirección única de los actores. Dicha dirección puede incluir, además de la *trayectoria* de la jerarquía del actor, la dirección de red del actor, haciendo posible la identificación de cualquier sistema de actor, incluso cuando este se encuentre en una JVM de otro equipo.

Un detalle es que cada nodo debe tener conocimiento de la existencia de otros



(a) Punto-a-Punto



(b) Bus de Eventos

Figura 2.3: Comunicación de Sistemas de Actor locales

nodos, y debe conocer su dirección de red dado que el mecanismo de comunicación es punto a punto. Hasta este punto, el mecanismo remoto de los sistemas de actor en Akka no incluyen mecanismos de descubrimiento como el implementado en los clusters y que detallaremos en la siguiente sección.

La estructura jerárquica implementada en el sistema de actor permite contar con un mecanismo de supervisión en el que los actores padres supervisan a los actores hijos. Dicho mecanismo facilita el manejo de excepciones, puesto que los errores tienen una trayectoria de propagación separada. Cuando un actor encuentra un error, este lo comunica al actor *supervisor* (el actor que inició la comunicación en un primer momento). Estos supervisores cuentan con métodos para manejar estas excepciones...

Cluster Akka

Para un conjunto grande de nodos, el sistema de actor remoto se vuelve complicado, pues agregar un nodo nuevo significa que el resto de los nodos del grupo

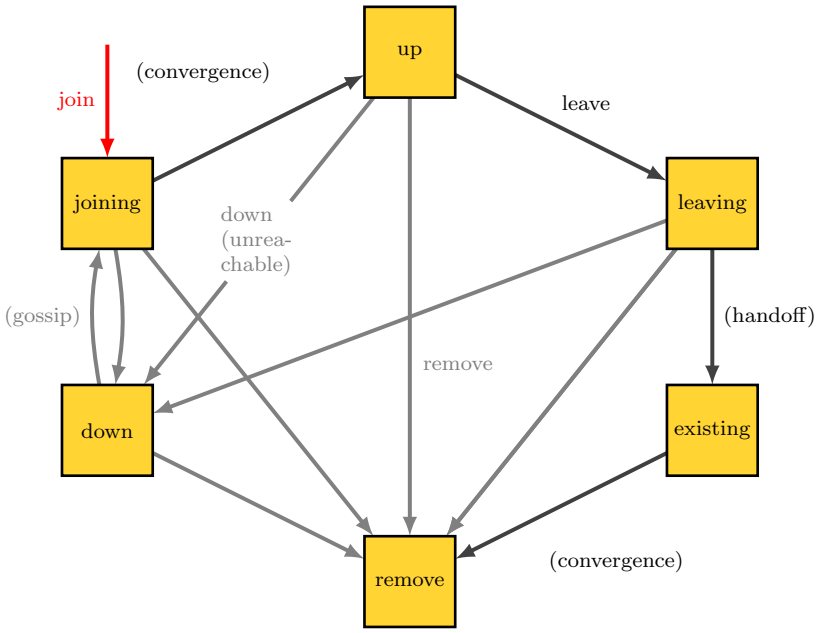


Figura 2.4: Estados del Cluster

deben estar informados de este evento, generando que crezca el ruteo de los mesajes.

Akka cuenta con un módulo de *clustering* que simplifica este proceso. Este módulo proporciona la capacidad para que los sistemas de actor, que están corriendo en múltiples JVM, puedan interactuar entre sí y comportarse como si fueran parte del mismo sistema de actor.

Sin el módulo de *clustering*, es necesario incorporar en el código la ubicación remota de los actores y, como consecuencia, construir las direcciones con las cuales se enviarán mensajes entre los actores. Con el módulo de *clustering*, en lugar que un nodo conozca la ubicación precisa del resto de los nodos directamente en la red, basta que conozcan uno o más *seed nodes*. Estos *seed nodes* son nodos designados para encargarse de conectar al cluster a los nodos que desean unirse.

El nuevo nodo que se unirá transita por una serie de pasos hasta que se convierte en un integrante pleno del cluster. La Figura 2.4 presenta ese *ciclo de vida*.

La administración del cluster, a través del *ciclo de vida*, se maneja de forma separada y se detalla a continuación. Para la administración de los miembros del cluster, se utiliza una variante al protocolo *gossip*. Monitoreos aleatorios hacen

comunicar un nodo a otro nodo la información que este tienen acerca de los miembros del cluster, mientras que el nodo que es informado hace saber al otro nodo la información que tiene. Después de varios pasos, se presenta una *convergencia*, esto es, cuando todos los nodos tienen una imagen completa de los miembros del cluster, de la misma vesion de la estructura del cluster. En este punto, no existe un nodo central que concentra esta información.

Para determinar aquellos nodos que han sido perdidos en el cluster, Akka clustering utiliza el detector de fallas Phi Accrual para detectar a aquellos nodos que son inalcanzables, y que eventualmente son retirados del cluster.

Cada cluster tiene un *líder*. Cada nodo puede determinar de forma algorítmica qué nodo debe ser el lider actual, basándose en la información recibida por *gossip* y prescindiendo que el lider sea elegido en una votación. El lider tiene dos tareas en la administración del cluster:

- Incorpora nuevos miembros al cluster cuando estos son elegibles a ser agregados y los retira cuando es necesario.
- Organiza el *balanceo* del cluster (*i.e.* los movimientos de actores entre los miembros del cluster).

2.1.2. El modelo de actor en CAF

C++ Actor Framework (CAF) [1] el cual es enteramente implementado en C++, y no utiliza la abstracción de una máquina virtual como Erlang o Akka [8].

De acuerdo a Charousset et al [1], el diseño de CAF está dirigido a cumplir los siguientes objetivos:

- Fiabilidad
- Escalabilidad
- Eficiencia de recursos
- Transparencia de la distribución

A su vez, para la fiabilidad, se necesita *type safety* para proporcionar un ambiente de programación robusto. Para la mejorar la eficiciencia de la demanda de recursos, son necesarios tres elementos: *1)* un procesamiento eficiente de mensajes que minimice los costos de la abstracción basada en mensajes, *2)* una *footprint* de muy bajo nivel para los actores y *3)* una liberación de asignación de memoria tan pronto sea posible para su reutilización. Para el caso de la escalabilidad, se requiere el uso eficiente de muchos CPU's.

En CAF, los actores se encuentran hospedados en un *ambiente de tiempo de ejecución* que proporciona el envío de mensajes, el manejo de las colas, etc. Los actores se comunican de forma remota con otros actores vía el ambiente de tiempo de ejecución de una forma transparente, en el sentido que, como los actores ven colas individuales, no deben estar enterados del lugar físico donde se encuentran otros actores. Esto se logra por la capa de paso de mensajes global de CAF (Ver Figura 2.5). Esta capa implementa componentes que implementan servicios individuales de los actores, así como múltiples instancias del intercambio de mensajes en tiempo de ejecución dado el *Binary Actor System Protocol* (BASP).

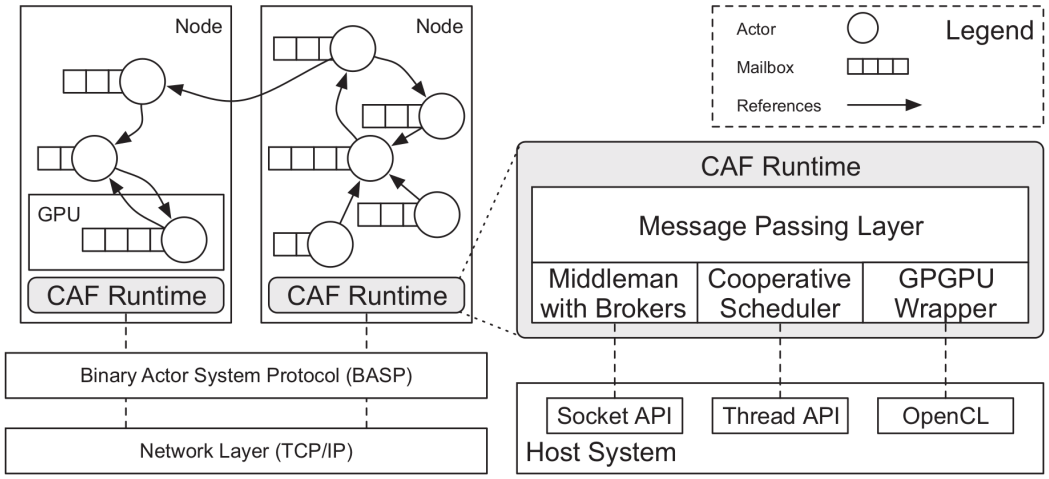


Figura 2.5: Arquitectura de CAF para una configuración distribuida en dos nodos, cada uno corriendo actores de forma local (Imagen tomada de [1])

En ambientes distribuidos se establece una capa de paso de mensajes común via un intermediario. La principal función de un intermediario es organizar el intercambio de mensajes vía interfaces de red como los sockets. Este encapsula

la API de red del sistema huesped para ocultar las primitivas de comunicación. Los torrentes de paquetes y bytes son convertidos a mensajes que son entregados a los *brokers*. Un broker es un actor que lleva acabo operaciones asíncronas de entrada y salida (I/O) y que viven en el loop de eventos del intermediario. Cuando una aplicación comienza, la ejecución de CAF instancia un actor que implementa BASP. El protocola transporta los mensajes del actor y propaga los errores de los actores que fallan hacia los actores remotos.

El scheduler cooperativo organiza una ejecución justa y concurrente de los actores a un nivel de sub-thread a nivel local del host. Este gestiona los threads de trabajo usando la librería estándar de C++ y distribuye la carga de trabajo entre estos.

CAF tiene la capacidad de integrar componentes heterogeneos usando fachadas que manejan la comunicación entre el huesped y dispositivos de aceleradores. CAF permite la creación de actores desde el kernel de OpenCL.

Buzón de mensajes en CAF

De acuerdo a Charousset et al [1], para la implementación del concepto de *buzón* del modelo de actor se identifican dos categorías:

- En la primera categoría, los actores iteran sobre los mensajes en su buzón. En cada recepción de una llamada, los actores comienzan con el primero en fila, pero son libres de saltarse mensajes. Como un actor puede cambiar su comportamiento como respuesta a un mensaje, un comportamiento recién definido puede aplicarse a mensajes previamente omitidos. Un mensaje permanece en el buzón hasta que finalmente es procesado y eliminado como parte de su consumo.

- La segunda categoría sigue un esquema de procesamiento de mensajes más restrictivo. Un manejador de mensajes es invocado una vez por mensaje con el comportamiento específico del actor. Un mensaje que no fue atendido no puede recuperarse en un tiempo posterior, incluso aunque algunos sistemas nos permiten cambiar el manejador de mensajes en tiempo de ejecución. En consecuencia, los actores son forzados a tratar los mensajes siguiendo el orden de llegada. Akka sigue este mecanismo de procesamiento de mensajes.

CAF sigue el primer enfoque. El argumento para esta selección es que este mecanismo permite priorizar mensajes y esperar respuestas antes de regresar un comportamiento por defecto. Saltarse mensajes para su posterior atención es una característica importante para aquellos sistemas de actores que permiten patrones de comunicación que requieren que un actor procese la respuesta de mensajes de multiples actores en un orden particular. La desventaja de este enfoque es que el desarrollador necesita indicar explicitamente cuales mensajes serán eliminados y, en consecuencia, que nunca serán procesados. Sin esta eliminación, estos mensajes se acumularán en el buzón del actor, ralentizando las operaciones de la cola, ya que estarán revisando repetidamente después de cada invocación exitosa de mensaje.

En este contexto, el *patrón de coincidencia*(*pattern matching*) ha probado ser útil y efectivo para la definición del manejador de mensajes. Por esta razón, CAF presenta un patrón de coincidencia para el manejo de mensajes como un *dominio específico del lenguaje* (*a domain-specific language*). Además, CAF implementa un algoritmo concurrente para el manejo de colas el cual está diseñado para usarse como buzón.

Algoritmo de buzón sin bloqueo

Todos los mensajes enviados a un actor son entregados a su buzón, que actúa como un recurso compartido si un actor recibe mensajes de múltiples actores en paralelo.

Un buzón es una cola de un único lector y múltiples escritores. El buzón está expuesto a un acceso de escritura paralelo, pero sólo al actor que es dueño de él le es permitido sacar mensajes de la cola. De manera que esta operación no necesita permitir accesos paralelos[1]

CAF logra mejorar de forma rápida y concurrente la operación de poner en cola, mientras que la operación de quitar de la cola mejora la rapidez, pero no de manera concurrente. Esto lo logra combinando una implementación de una pila sin

[1]CAF realiza una estrategia de optimización *copy-on-write* para minimizar los gastos generales de copiar un mensaje en tiempo de ejecución. Un mensaje puede ser compartido por varios actores en la misma instancia tan pronto como todos los participantes sólo demanden accesos de lectura. Un actor implicitamente copia el mensaje compartido cuando este requiere acceso de escritura y sólo se le permite modificar su propia copia. De manera que la *condición* de carrea no puede ocurrir por diseño y cada mensaje es copiado sólo si se necesita.

bloqueo (*lock-free stack*) con una cola ordenada FIFO como caché intermedio. La pila sin bloqueo es implementada usando una operación atómica CAS (*compare-and-swap*)[2]. Los autores [1] señalan que esta operación atómica no padece del problema de acceso concurrente conocido como ABA el cual puede corromper los estados en sistemas basados en CAS, ya que la operación de poner en cola ya que sólo necesita manipular el apuntador de la cola. Sin necesidad de ordenar, la operación de quitar de la cola tendría que atravesar la pila con el objetivo de encontrar el elemento más viejo.

Definición de patrones y actores en CAF

Un actor se define en términos de los mensajes que envía y recibe. De manera que su comportamiento se especifica como un conjunto de manejadores de mensajes que despachan la extracción de datos de funciones asociadas. La definición de estos manejadores es una tarea común y recurrente en la programación de actores. El patrón de pareo (*pattern matching*) de los lenguajes de programación funcionales ha probado ser una vía adecuada para la definición de tales manejadores de mensajes. En la comunidad de C++ se le reconoce como un mecanismo de abstracción poderoso. Sin embargo, no existe aún una API estándar ni soporte dentro del propio lenguaje. Esta es una carencia importante dado que el patrón de pareo es un ingrediente clave para la definición de actores de una manera natural. Por este motivo, CAF proporciona un dominio específico del lenguaje (DSL, *domain-specific language*) para este propósito.

El DSL de CAF está limitada a los mensajes de actores que mantienen la interface ligera y enfocada en la definición de los manejadores de mensajes. Contrario a otros mecanismos de despacho en tiempo de ejecución, la implementación de patrón de pareo revela al compilador todos los tipos de mensajes entrantes así como todos los tipos de mensajes salientes. En este sentido, el compilador puede derivar la interface de un actor de la definición de su comportamiento para realizar la comprobación de los tipos estáticos.

Un patrón en CAF es una lista de *casos de coincidencia* (*match cases*). Cada caso es:

[2]Una operación CAS es una operación atómica usada en el diseño de algoritmos concurrentes para permitir la sincronización de hilos de ejecucións sin recurrir a bloqueos de los mismos

- **Trivial**. Un caso trivial es generado por callbacks, generalmente expresiones lambda.Los tipos de entrada y salida son derivados simplemente de la signatura del callback.
- **Una regla general(*a catch-all rule*)**. Una regla general comienza con `others`, seguido de un callback sin argumentos que retorna un `void`. Un caso de este tipo siempre empareja y no produce ningún mensaje de respuesta.
- **Una expresión avanzada que permite guardas y proyecciones**. Un caso de coincidencia avanzada comienza con una llamada a la función `on` que devuelve un objeto intermedio que proporciona el operador $\gg$. El lado derecho de este operador denota un callback que debe ser invokado después de la coincidencia de un mensaje de los tipos derivados de `on`. Cada argumento de `on` es un objeto función con signatura `T->optional <U>` o un valor.

Manejo de los actores en CAF

De acuerdo a Charousset et al [1], en sistemas basados en paso de mensajes, las entidades de software tienen dos atributos característicos: identificadores e interfaces. El primero direcciona las entidades de software (actores, en este caso) en una red, mientras que el segundo codifica y valida las entradas.

En el modelo original de actor se usan las direcciones de correo como identificadores con una interfaz implicita. La interface acepta cualquier intput y el receptor es responsable de despachar de forma dinámica los datos recibidos, usualmente a través de un patrón de matching.

Muchas implementaciones del modelo de actor siguen este diseño, en que al remitente se le permiten recibir entradas arbitrarias. Tal es el caso de Akka con `ActorRef` el cual es usado directamente para pasar mensajes y para aceptar entradas de tipo `Any` en Scala y tipo `Object` en Java. Estas son las raíces de las jerarquías de clase en ambos lenguajes. Los autores [1] sugieren que esto representa un problema, pues se esconde información al compilador y hace imposible un análisis estático de las interfaces.

Algunas implementaciones de modelo de actor (SALSA o Charm++) utilizan un enfoque Orientado a Objetos (OO) para definir las interfaces, las cuales generan dependencias entre remitentes y receptores.

CAF proporciona un diseño alternativo que permite comprobar los tipos estáticos sin introducir dependencias entre remitentes y receptores. Este diseño revela todo tipo de información al compilador para habilitar una verificación global de los tipos de los mensajes sin recurrir a una jerarquía de herencia tipo OO. Al usar una definición de interface específica del dominio, se le permite a los actores enviar mensajes con sólo información de tipo parcial del receptor. Por otro lado, diseño distingue explicitamente entre identificadores e interfaces, haciendo a ambos accesibles al programador.

Sin embargo, CAF también integra interfaces como las usadas por Akka las cuales pueden ser manejadas mediante el tipo `actor`. Ya sea en el diseño original de CAF o la alternativa descrita, ambas interfaces pueden ser consultadas para regresar el identificador de tipo `actor addr`. Este identificador es usado en tiempo de ejecución para dirigir y monitorear de forma única a los actores en sistemas distribuidos.

Tipado estático y dinámico de actores

CAF permite el tipado estático y dinámico de los actores. El tipado dinámico tiene la capacidad de proporcionar un único tipo encargado para manejar a los actores. Este enfoque está más cercano al modelo original de Hewitt et al que no especifica cómo (e incluso sí) los actores especifican la interfaz para los mensajes entrantes y salientes. En su lugar, los actores son definidos en términos de los *nombres* que ellos utilizan, *derechos de acceso* que otorgan a *conocidos* y *patrones* que especifican para despachar el contenido de los datos entrantes.

Por su parte, con el tipado estático de actores, el compilador es capaz de verificar los protocolos entre los actores. Por lo tanto, el compilador es capaz de descartar toda una categoría de errores en tiempo de ejecución, puesto que la violación del protocolo no puede ocurrir una vez que el programa ha sido compilado. EL compilador no sólo verifica el envío correcto de un mensaje sino también el manejo del resultado cuando utiliza `sync_send`.

Cuando se usa `sync_send`, se asigna un único ID al mensaje. El actor remitente puede usar `.then` para instalar una continuación para el mensaje de respuesta. La continuación de mensajes es en sí misma un manejador de mensajes que es usado

para responder el mensaje con un ID particular. El remitente se sincroniza con el receptor omitiendo cualquier otro mensaje entrante hasta que recibe el mensaje de respuesta o se produce un tiempo de espera. Cualquier error, por ejemplo, si el remitente ya no existe o no es alcanzable, generará que el remitente salga con un motivo de salida anormal a menos que proporcione un manejador de errores personalizado. Por lo tanto, usando `sync_send-even` sin tiempo de espera, dará al programador un manejo garantizado de los errores. Además, en tiempo de ejecución se generará un mensaje de respuesta vacío si el receptor controla el mensaje, pero no responde con un mensaje por sí mismo. De manera que `sync_send` captura los errores dentro de la lógica del programa. Combinado con el tipado estático de los actores, esto también elimina la posibilidad que el receptor nunca responda. Así, `sync_send` en combinación con el tipado estático de los actores, elimina escenarios de deadlock accidentales donde un remitente espera por siempre porque falló en utilizar el tiempo de espera y el monitoreo de la recepción.

Brokers: actores para I/O asíncronos

Al comunicarse con otros servicios en la red, es frecuentemente inevitable manejar manualmente paquetes de datos o flujos de bytes. Por esta razón, CAF proporciona *brokers* como un mecanismo abstracto basado en actores sobre primitivas de red. Este es comparable con el mecanismo existente en Erlang y en Akka. Un broker es un actor basado en eventos corriendo en el llamado *intermediario* (MM). El intermediario es una entidad de software que "multiplexea.ºperaciones I/O de bajo nivel (soket) y permite la unión tardía de primitivas de comunicación específicas de la plataforma. El intermediario traduce los eventos de la capa de red y lo flujos de bytes a mensajes de CAF.

Los actores operan con cualquier número de tipos de manejadores. Manejadores del tipo `accept_handle` identifica un punto final de conexión al cual otros pueden conectarse. Un `connection_handle` identifica un flujo de bytes punto a punto, por ejemplo, una conexión TCP. Si se establece una nueva conexión, el intermediario envía un `new_connection_msg` al broker asociado. Mensajes de este tipo contienen el identificador que aceptó la conexión(`source`) y un identificador a la nueva conexión (`handle`). Si arriban nuevos datos, el intermediario envía un `new_data_msg`

al broker asociado que contiene la fuente de este evento (`handle`) y un buffer que contiene los bytes recibidos (`buf`).

2.1.3. Comparación de los frameworks

La Tabla 2.1 presenta un comparativo de las características de los frameworks.

Cuadro 2.1: Comparación de los frameworks Akka y CAF

	Akka	**CAF**
Aislamiento de actores	• Cuando se crea una instancia de un actor, se regresa un `ActorRef`, el cual representa un *wrapper* de este actor, aislándolo del resto de código.	• CAF se abstiene de generadores de código o herramientas similares para el aislamiento de los actores, proponiendo en su lugar usar únicamente el compilador estándar de C++.
Buzón de mensajes	• Los actores atienden los mensajes siguiendo el orden de llegada.	• En cada recepción de una llamada, los actores comienzan con el primero en fila, pero son libres de saltarse mensajes. Este mecanismo es implementado mediante un patrón de coincidencia para el manejo de mensajes como un *dominio específico del lenguaje* (*a domain-specific language*). Además, CAF implementa un algoritmo concurrente para el manejo de colas el cual está diseñado para usarse como buzón.
Manejo de actores	• El remitente se le permiten recibir entradas arbitrarias. En Akka, `ActorRef` es usado directamente para pasar mensajes y para aceptar entradas de tipo `Any` en Scala, la cual es la raíz de las jerarquía de clase. En suma, la interfaz implícita en los actores realiza un análisis dinámico de los tipos recibidos.	• CAF integra interfaces como las usada por Akka y una alternativa. Para el primer caso, la interfaz puede ser manejada mediante el tipo `actor`. Para la segunda, la interfaz permite comprobar los tipos estáticos sin introducir dependencias entre remitentes y receptores. Al usar una definición de interface específica del dominio, se le permite a los actores enviar mensajes con sólo información de tipo parcial del receptor.

Cuadro 2.1: Comparación de los frameworks Akka y CAF

Sistema de actor	• Akka implementa el sistema de actor de manera local y remota. Para el primer caso, los actores se encuentran en una misma Java Virtual Machine (JVM). Para los sistemas de actor remotos, Akka permite la comunicación de estos sobre una red y la serialización de mensajes, de manera que actores en distintas JVM pueden enviar y recibir mensajes.	•
Comunicación remota	• Para los sistemas de actor remotos, Akka permite la comunicación de estos sobre una red y la serialización de mensajes, de manera que actores en distintas JVM pueden enviar y recibir mensajes. Esta comunicación se puede realizar por la dirección única de los actores. Dicha dirección puede incluir, además de la *trayectoria* de la jerarquía del actor, la dirección de red del actor, haciendo posible la identificación de cualquier sistema de actor, incluso cuando este se encuentre en una JVM de otro equipo.	• CAF proporciona *brokers* como un mecanismo abstracto basado en actores sobre primitivas de red. Un broker es un actor basado en eventos corriendo en el llamado *intermediario* (MM). El intermediario es una entidad de software que *multiplexea* operaciones I/O de bajo nivel (soket) y permite la unión de primitivas de comunicación específicas de la plataforma. El intermediario traduce los eventos de la capa de red y lo flujos de bytes a mensajes de CAF.
Particularidades	• **Cluster Akka**: Para un conjunto grande de nodos, el sistema de actor remoto se vuelve complicado, pues agregar un nodo nuevo significa que el resto de los nodos del grupo deben estar informados de este evento, generando que crezca el ruteo de los mesajes. Con el módulo de *clustering*, en lugar que un nodo conozca la ubicación precisa del resto de los nodos directamente en la red, basta que conozcan uno o más *seed nodes*. Estos *seed nodes* son nodos designados para encargarse de conectar al cluster a los nodos que desean unirse.	• **Dispositivos heterogeneos**: CAF tiene la capacidad de integrar componentes heterogeneos usando fachadas que manejan la comunicación entre el huesped y dispositivos de aceleradores. CAF permite la creación de actores desde el kernel de OpenCL.

2.2. Patrones arquitectónicos para programación paralela

Buschmann et al [9] define un *patrón arquitectónico* como un esquema de organización estructural para sistemas de software. El patrón proporciona un conjunto de subsistemas predefinidos, para los cuales se especifican sus responsabilidades e incluyen reglas y guías para organizar la relación entre estos. Los patrones arquitectónicos pueden pensarse como *templates* para arquitecturas de software concretas.

Los patrones arquitectónicos para programación paralela pueden ser clasificados siguiendo las características de los sistemas paralelos como criterio de clasificación [10]. Dichos patrones están definidos y clasificados de acuerdo a requerimientos del orden de los datos así como de las operaciones, y de la naturaleza de sus componentes de procesamiento. En este sentido, Ortega [10] indica que, considerando requerimientos del orden de los datos así como de las operaciones, las aplicaciones paralelas se pueden clasificar en tres grupos:

- *Paralelismo Funcional.*
- *Paralelismo de Dominio.*
- *Paralelismo de actividad*

El paralelismo funcional es aplicado a problemas que involucran una serie de operaciones ordenadas por pasos de tiempo. El paralelismo de dominio se relaciona con problemas en los que un conjunto de operaciones casi independientes se aplican a datos locales ordenados. Por último, el paralelismo de actividad involucra problemas a los que se les aplican cálculos independientes a valores de estructuras de datos ordenados o desordenados.

Por otra parte, se considera como criterio de clasificación de sistemas paralelos a la naturaleza de los componentes de procesamiento. Estos componentes llevan acabo actividades de coordinación y procesamiento. Tomando este criterio, los sistemas paralelos son clasificados en dos grupos[10]:

- *Sistemas homogéneos.*

- *Sistemas heterogéneos.*

Los sistemas homogéneos se basan en la iteracción de componentes idénticos en torno a un conjunto de reglas simples de comportamiento. Por su parte, los sistemas heterogeneos involucran componentes diferentes con reglas de comportamiento y relaciones especializadas.

En términos de los criterios anteriores, Ortega [10] presenta cinco patrones arquitectónicos para programación de sistemas paralelos:

- *Pipes and Filters*
- *Parallel Hierarchies*
- *Communicating Sequencial Elements*
- *Manager-Workers*
- *Shared Resources*

2.2.1. Patrón arquitectónico Manager-Workers

Ortega [10] señala que el patrón *Manager-Workers* es una variante de patrón *Master-Slave* de Buschmann et al [9] para el caso de sistemas paralelos, considerando un enfoque de paralelismo de actividad en el que se dividen tanto algoritmos como datos son y las mismas operaciones son llevadas acabo 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 acabo simultaneamente las mismas operaciones e independientemente del procesamiento de los otros componentes. Es importante que el orden de los datos se preserve. En la Tabla 2.2 se presenta un resumen de las características del patrón *Manager-Workers* con base en [10].

Cuadro 2.2: Resumen del patrón Manager-Workers

Concepto	**Descripción**
Contexto	• El problema involucra tareas de una escala que sería no realista o poco eficiente para que otros sistemas manejen y presten una solución que implica paralelismo. • La plataforma paralela así como el ambiente de programación a utilizarse ofrecen un ajuste razonable 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. Muchos de los lenguajes tienen extensiones paralelas o bibliotecas para plataformas paralelas. El objetivo principal es que se ejecute la tarea en el tiempo más eficiente.
Problema	• La misma operación debe ser llevaba acabo repetidamente sobre todos los elementos de un conjunto de datos ordenado. A pesar de esto, los datos pueden ser procesados sin un orden específico. Sin embargo, es importante preservar el orden de los datos.
Requerimientos	Deben de considerarse las siguientes condiciones : • El orden de los datos debe preservarse. Sin embargo, no es fijo el orden de las operaciones aplicadas a cada conjunto de datos. • La operación puede realizarse de forma independiente sobre diferentes conjuntos de datos. • Los conjuntos de datos pueden tener distinto tamaño. Esto significa que los cómputos independientes a cada conjunto de datos deben adaptarse al tamaño de los datos a procesar, esto para obtener un balanceo de carga automático. • La solución debe escalar sobre la cantidad de elementos de procesamiento. Los cambios en número de elementos de procesamiento deben verse reflejados en el tiempo de ejecución. • El mapeo de los elementos de procesamiento a procesadores debe tomar en cuenta la interconexión de la plataforma de hardware.

Cuadro 2.2: Resumen del patrón Manager-Workers

Solución	Introducir paralelismo de actividad para procesar múltiples conjuntos de datos al mismo tiempo. El patrón *Manager-Workes* 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 acabo 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. Es ahí cuando el programa termina. El modelo de ejecución es el mismo independientemente del número de workers. 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 estár 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 tamaño. El manager se ocupa de preservar la integridad de los datos, pues monitorea qué 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 recive las solicitudes de datos de los workers. De manera que las operaciones del manager necesitan capacidades de sincronización y bloqueo.
Estructura	El patrón *Manager-Workers* está compuesto de un manager y uno o más workers. Los workers actuan como el elemento de procesamiento. Usualmente, sólo existe un manager y varios workers identicos 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.

Cuadro 2.2: Resumen del patrón Manager-Workers

Dinámica	Los pasos para realizar un conjunto de cálculos es: • Todos los participantes son creados y esperan hasta que un cómputo es requerido por el manager. Cuando el dato esta 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.

2.3. Diseño de programas paralelos

Implementar una solución paralela puede realizarse detectando partes o subtareas de un algoritmo secuencial latentes a ser paralelizadas. Por otra parte, es posible definir el diseño del algoritmo a ser paralelo desde el inicio. Foster [3] propuso una metodología para el diseño de algoritmos paralelos basada en cuatro puntos :

- **Particionamiento**: Aquí ya sean los datos, las tareas, o ambas, se dividen en un número de procesadores. El particionamiento de datos se le denomina *descomposición de dominio* y al de tareas *descomposición funcional*.
- **Comunicación**: En este punto se analizan la cantidad de subtareas paralelas de envío, recepción y de datos.
- **Aglomeración**: Partiendo de los dos puntos anteriores, en esta etapa del diseño se analiza la forma de acomodar los resultados anteriores para formar grupos relativamente grandes con el propósito de reducir la comunicación entre tareas.
- **Mapeo**: En este punto, se define una correspondencia entre los procesadores y los grupos formados en el punto anterior.

2.4. Métricas de desempeño

Un programa paralelo es aquel que lleva acabo operaciones de forma simultánea [11]. Existen varias razones principales para realizar programas paralelos. Una de los principales es 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. Speedup y eficiencia

Una métrica para evaluar el beneficio potencial de un programa paralelo es 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*,$S(N)$, de usar N procesadores se define como:

$$S(N) = \frac{T_p(1)}{T_p(N)}, \tag{2.1}$$

Donde $T_p(1)$ es el tiempo de procesamiento del algoritmo en un sólo procesador y $T_p(N)$ es el tiempo de procesamiento de los procesadores paralelos. En un situación ideal, para un algoritmo completamente paralelizable, y cuando no se considera el tiempo de comunicación entre los procesadores y la memoria, tenemos que $S(N) = \frac{T_p(1)}{N}$, de manera que de la ecuación anterior tenemos:

$$S(N) = N, \tag{2.2}$$

La métrica de eficiencia de un sistema paralelo es un indicador de la fracción de tiempo utilizado por los N procesadores en un cálculo dado. Se calcula de la siguiente manera:

$$E(N) = \frac{S(N)}{N} \tag{2.3}$$

2.4.2. Ley de Amdahl

Asumiendo que un algoritmo o una tarea está compuesta por una fracción paralelizable f y una fracción serial $1 - f$. Suponiendo que el tiempo necesario

para procesar esta tarea en un sólo procesador está dado por:

$$T_p(1) = N(1-f)\tau_p + Nf\tau_p = N\tau_p, \tag{2.4}$$

donde el primer término del lado derecho de la primera igualdad es el tiempo que necesita el procesador para procesar la parte serial, mientras que el segundo término es el tiempo que el procesador necesita para procesar la parte paralela. Cuando esta tarea es ejecutada en N procesadores paralelos, el tiempo de procesamiento estará dado por:

$$T_p(N) = N(1-f)\tau_p + f\tau_p, \tag{2.5}$$

Donde el speedup se presenta solamente en la parte paralela que ahora está distribuida en N procesadores. La ley de Amdahl para el speedup $S(N)$ esta dada por:

$$\begin{aligned} S(N) &= \frac{T_p(1)}{T_p(N)} \\ &= \frac{N}{(1-f)N + f} \\ &= \frac{1}{(1-f) + \frac{f}{N}}, \end{aligned} \tag{2.6}$$

Para valores extremos de f, la ecuación anterior presenta :

- $S(N) = 1$ cuando $f = 0$, es decir, el código es completamente serial.
- $S(N) = N$ cuando $f = 1$, es decir, el código es completamente paralelizable.

Para valores grandes de N, el speedup anterior se aproxima a:

$$S(N) \approx \frac{1}{1-f} \tag{2.7}$$

La Figura 2.6 muestra los Speedup de la Ley de Amdahl correspondientes a distintos valores de la fracción paralelizable.

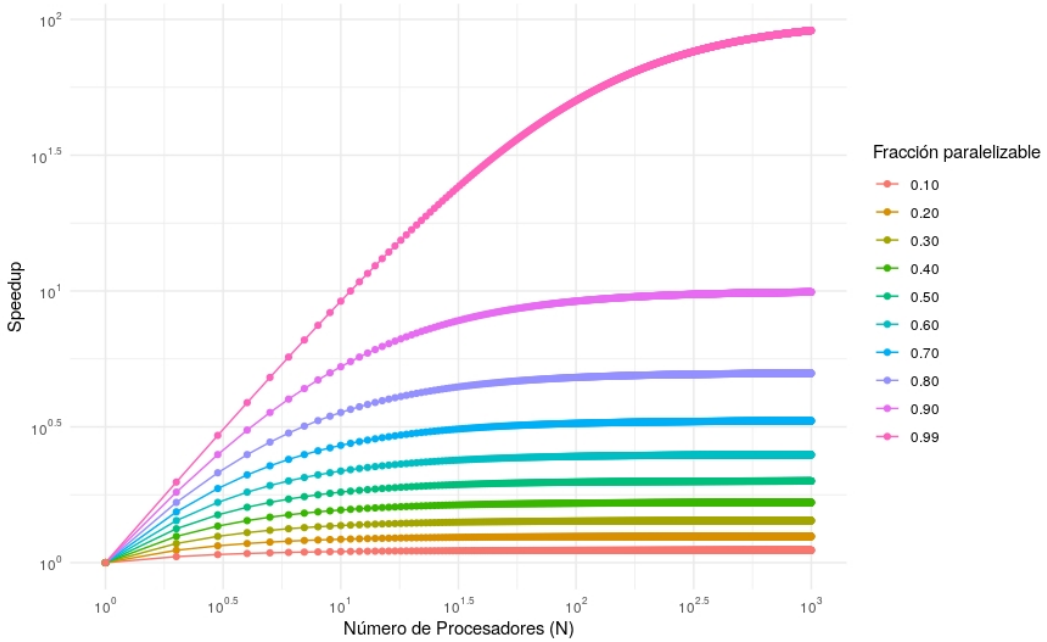


Figura 2.6: Speedup correspondientes a la Ley de Amdhal

2.4.3. 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 abordó 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 está dado por:

$$T_p(N) = (1 - f)\tau_p + f\tau_p = \tau_p, \tag{2.8}$$

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

$$\begin{aligned} S(N) &= \frac{T_p(1)}{T_p(N)} \\ &= (1-f) + Nf \\ &= 1 + (N-1)f, \end{aligned} \tag{2.9}$$

La Figura 2.7 muestra el speedup para distintos valors de N y f. Se observa que hay speedup incluso para valores pequeños de f, en tanto que la situación mejora en la medida que aumenta N.

Para obtener un speedup, necesita cumplirse la siguiente desigualdad:

$$f(N-1) \gg 1 \tag{2.10}$$

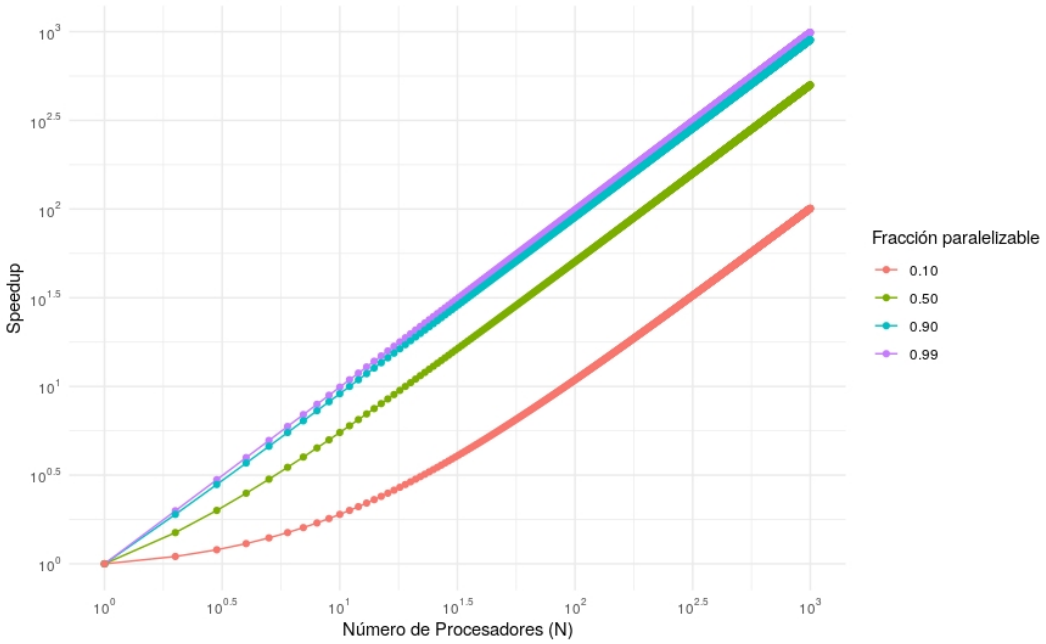


Figura 2.7: Speedup correspondientes a la Ley de Gustafson

2.5. Descripción del problema

Se pide obtener el coeficiente de clustering de los nodos de una red no dirigida.

La red se encuentra en su representación de matriz de adyacencia. Una matriz de adyacencia se define como una matriz $A[n, n]$ donde cada elemento $a_{ij} = 1$ si hay una arista que une el vértice (nodo) i con el j y $a_{ij} = 0$ en otro caso. La matriz de adyacencia es simétrica para el caso de una gráfica no dirigida y asímétrica para digráficas en general [12, pp 30].

Puesto que se nos pide obtener el *coeficiente clustering por nodo* (c_i), podemos usar la siguiente expresión propuesta por Latora et al [13, pp 117-118]:

$$c_i = \begin{cases} \dfrac{K[G_i]}{k_i(k_i-1)/2} & \text{para } k_i \geq 2 \\ 0 & \text{para } k_i = 0, 1 \end{cases} \tag{2.11}$$

donde $K[G_i]$ es el número de conexiones en G_i, la subgráfica inducida por la vecindad de i.

Latora et al [13] indican que podemos obtener c_i a partir de la matriz de adyacencia de la siguiente manera:

$$c_i = \begin{cases} \dfrac{\sum_{l,m} a_{il}a_{lm}a_{mi}}{k_i(k_i-1)} = \dfrac{\left(A\right)^3_{ij}}{k_i(k_i-1)} & \text{para } k_i \geq 2 \\ 0 & \text{para } k_i = 0, 1 \end{cases} \tag{2.12}$$

El número de aristas $K[G_i]$ en la gráfica G_i puede ser obtenida en términos de la matriz de adyacencia de la gráfica. De acuerdo a los autores, $\left(A\right)^3_{ij} = \sum_{l,m} a_{il}a_{lm}a_{mj}$ es igual al número de recorridos de 3 pasos que conectan al nodo i con el nodo j. En particular, para $i = j$, la cantidad $\left(A\right)^3_{ii} = \sum_{l,m} a_{il}a_{lm}a_{mi}$ denota el número de trayectorias de 3 pasos que llevan del nodo i a si mismo. Esto es dos veces la cantidad de triángulos generados en la vecindad del nodo i. El triángulo que contiene al nodo i y a los dos nodos l y m está compuesto por dos aristas conectadas al nodo i, llámese (i, l) y (m, i), y por la arista (l, m) que pertenece a la gráfica G_i inducida por los primeros vecinos de i. Dado que la arista (l, m) aparece dos veces (en las trayectorias (i, l, m, i) y (i, m, l, i)), el número de aristas $K[G_i]$ está dado por:

$$K[G_i] = \frac{1}{2}(A)^3_{ii} = \frac{1}{2}\sum_{l,m} a_{il}a_{lm}a_{mi}$$

De manera que sustituyento esta expresión en (2.11) obtenemos (2.12).

Así, para el cálculo de c_i tenemos que calcular el cuadrado de la matriz de adyacencia(A^2). No es necesario calcular por completo el cubo de la matriz de adyacencia, pues los valores que necesitamos de esta son los de la diagonal principal, los cuales lo obtenemos de realizar la multiplicación de las i filas de A^2 por las de la matriz de adyacencia.

Supongamos la siguiente red no dirigida (Figura 2.8).

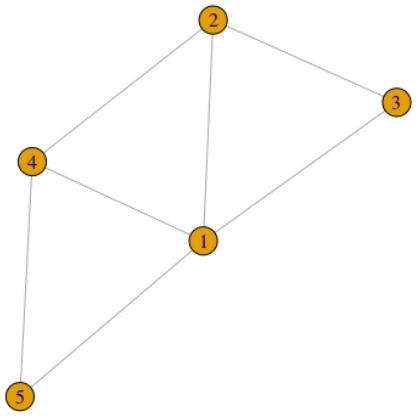


Figura 2.8: Gráfica no dirigida

La cual tiene la siguiente representación en una matriz de adyacencia:

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

A^2 y A^3 son las siguientes:

$$A^2 = \begin{bmatrix} 4 & 2 & 1 & 2 & 1 \\ 2 & 3 & 1 & 1 & 2 \\ 1 & 1 & 2 & 2 & 1 \\ 2 & 1 & 2 & 3 & 1 \\ 1 & 2 & 1 & 1 & 2 \end{bmatrix} \qquad A^3 = \begin{bmatrix} 6 & 7 & 6 & 7 & 6 \\ 7 & 4 & 5 & 7 & 3 \\ 6 & 5 & 2 & 3 & 3 \\ 7 & 7 & 3 & 4 & 5 \\ 6 & 3 & 3 & 5 & 2 \end{bmatrix}$$

La diagonal principal de la matriz A^3 indica el doble de las trayectorias de 3 pasos que llevan al nodo i a si mismo. En otras palabras, es dos veces la cantidad de triángulos que generados por la vecindad del nodo i.

De manera que el coeficiente de clustering por nodo (c_i) se muestra en la Tabla 2.3:

i	k_i	$A_i^3 i$	c_i
1	4	6	$\frac{6}{4(4-1)} = 0{,}5$
2	3	4	$\frac{4}{3(3-1)} = 0{,}666$
3	2	2	$\frac{2}{2(2-1)} = 1$
4	3	4	$\frac{4}{3(3-1)} = 0{,}666$
5	2	2	$\frac{2}{2(2-1)} = 1$

Cuadro 2.3: Coeficientes de clustering por nodo (c_i)

2.6. Resumen

En este capítulo se presentó el modelo de Actor, así como su implementación en Akka y CAF. Se mostraron las diferencias en implementación de los elementos distintivos del modelo de actor.

Se explicó a grandes rasgos la metodología de Foster [3] para el diseño de programas paralelos.

Abordamos qué es un *Patrón Arquitectónico* y cómo se aplica a la *Programación Paralela*. Entramos en detalle al patrón *Manager-workers*.

Se presentaron métricas para la evaluación del desempeño de programas paralelos, como el *Speedup* y la *Eficiencia*, y las Leyes de Amdahl y Gustafson.

Por último, abordamos una problemática en particular que consiste en el cálculo de el coeficiente de clustering para una red no dirigida mediante su representación en matriz de adyacencia.

Capítulo 3

Trabajo Relacionado

3.1. Evaluación de los frameworks CAF y Akka

De acuerdo a Boer et al [2], dentro de la familia de lenguajes basados en el modelo de actor, se encuentran los lenguajes de *objeto activo* que se caracterizan por utilizar como paradigma de comunicación *llamadas a métodos asíncronos*, los cuales restringen la comunicación entre objetos activos a mensajes que provocan la activación de métodos. Estos mismos autores comparan los lenguajes de *objeto activo* Rebeca, ABS, ASP, and Encore, basándose en el *objetivo del lenguaje*, *grados de sincronización*, *grados de transparencia*, *grados de compartimiento de datos*, *semántica* e *implementación y soporte*.

CAF ha sido implementado en arquitecturas heterogéneas al incorporarlo con OpenCL[14]. Además, se han realizado pruebas usando el patrón *Map*[8] y CAF.

La escalabilidad de Akka ha sido puesta a prueba por Google Compute Engine (GCE) en plataformas en la nube corriendo un cluster con 2400 nodos.

Charousset et al [1] realizaron una evaluación del desempeño de distintos lenguajes y API's que implementan el modelo de actor, enfocándose en el escalamiento de estos en términos de la utilización del CPU y eficiencia de memoria. Estos autores realizan las siguientes pruebas:

- Micro benchmark de la creación de actores y la eficiencia del buzón de mensajes.
- Ejecución de escenarios que involucran operaciones mixtas.

- Cálculo del Mandelbrot set del Computer Language Benchmarks Game.
- Evaluación en GPU.
- Evaluación en un ambiente distribuido para el cálculo del Mandelbrot set.

Para éste último punto, los autores utilizan 16 nodos Intel i7 quad core de 3.4 GHz, para los cuales definen 4 actores en cada uno.

En un primer momento, los autores presentan los resultados de la misma prueba pero comparando CAF y OpenMPI (Ver Figura 3.1). Esta prueba la realizan también en un ambiente distribuido virtual con hasta 64 maquinas virtuales como nodos. Para el caso virtual, los autores indican que se ejecuta 4 % más rápido en 64 nodos, lo cual indica una ligera mejor escalabilidad que OpenMPI. Para la evaluación en el ambiente físico, encuentran que CAF es 3 % más rápido que OpenMPI en 16 nodos físicos, lo cual aporta evidencia a favor de una mejor escalabilidad.

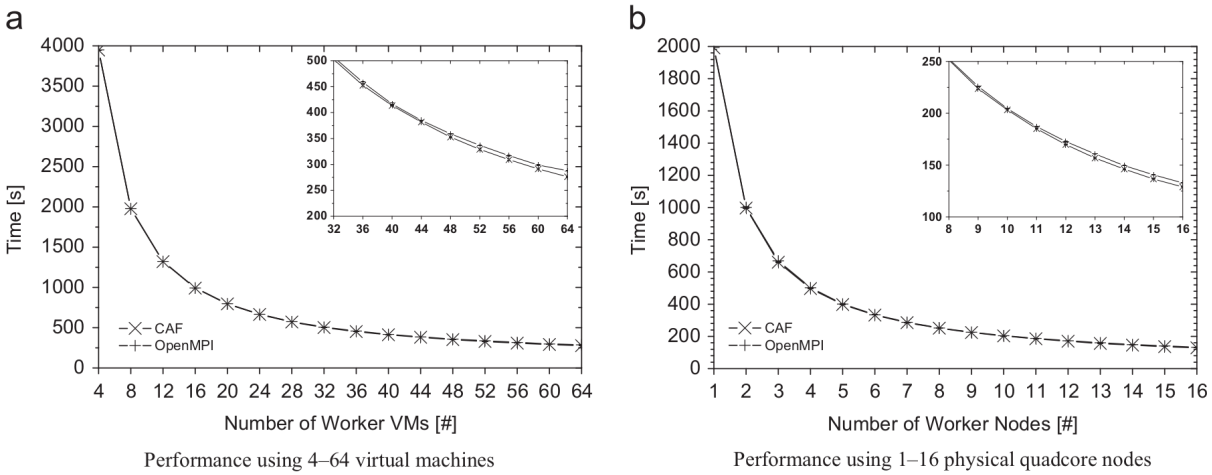


Figura 3.1: Evaluación del desempeño del cálculo del Mandelbrot set de CAF y MPI para ambientes distribuidos virtualizados y físicos. (Imagen tomada de [1])

Los autores evalúan CAF, Charm++, Erlang y Scala (que implementa Akka) en el ambiente distribuido físico. En términos generales, todas las implementaciones tienen un comportamiento similar, acercándose al *speed up* ideal para el cual el duplicar el número de nodos reduce a la mitad el tiempo de ejecución. Sin embargo, cuando realizan un análisis más detallado para los nodos 12-16, los autores señalan que sólo CAF fue más rápido que la implementación de OpenMPI, mientras que

Scala fue aproximadamente 1 % más lenta (Ver Figura 3.2). Dado que los autores no presentan los resultados de más pruebas, no es posible determinar si estas diferencias son estadísticamente distintas.

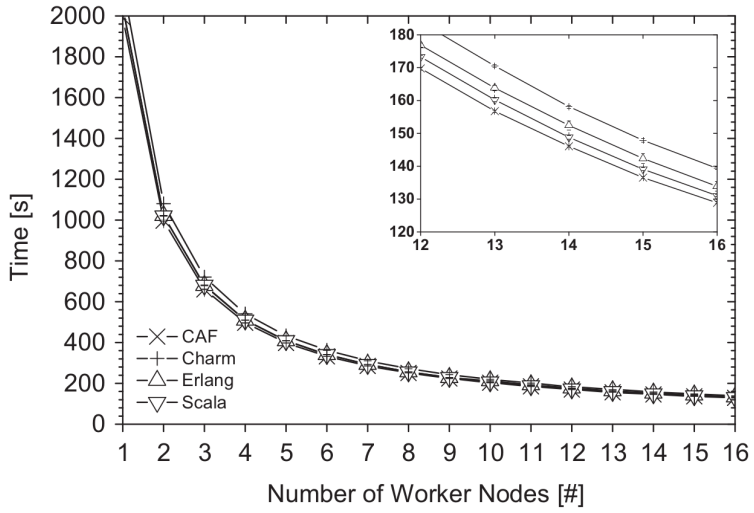


Figura 3.2: Evaluación del desempeño del cálculo del Mandelbrot set en CAF, Scala, Erlang y Charm, en un ambiente distribuido físicos. (Imagen tomada de [1])

3.1.1. Resumen

En este capítulo se detalló la evaluación del desempeño realizada por Charousset et al [1] de CAF y un conjunto de lenguajes y API's que implemenentan el modelo de actor. En primer lugar, se destaca que, para las condiciones en las que se realizó la evaluación, los autores indican que CAF obtuvo mejor desempeño en términos de escalabilidad, tanto en un ambiente distribuido virtual como físico, comparado con un enfoque de paso de mensajes de bajo nivel como el OpenMPI.

Por otra parte, se señala que en la evaluación anterior, Scala fue más lenta que la implementación en OpenMPI, y, como consecuencia, que CAF. Sin embargo, los autores no presentan que estas diferencias sean estadísticamente significativas. Lo cual deja la discusion abierta para evaluaciones futuras.

Capítulo 4

Propuesta

4.1. Diseño de la solución paralela

En esta sección se presentarán los pasos seguidos para la construcción de una solución parallela para el cálculo del coeficiente de clustering de nodos de grado k de una red no dirigida.

Se presentará como se incorpora el patrón *Manager-Workers* bajo el modelo de actor.Posteriormente, explicamos las decisiones tomadas en cada punto de los propuestos por Foster [3] para la solución al problema planteado. Un supuesto esencial es que trabajaremos con una red no dirigida. Dada la propiedad de este tipo de red, señalada en el Capítulo 2, la matriz de adyacencia es simétrica, lo cual hace menos laboriosos los cálculos de algunas métricas.

4.1.1. Patrón Manager-Workers en el Modelo de Actor

Como se mencionó en el Capítulo 2, el patrón *Manager-Workers* se compone de dos participantes: un *manager* y uno o más *workers*. Las responsabilidades del manager son crear crear workers, distribuir trabajo entre ellos, iniciar la ejecución y obtener el resultado general a apartir de los resultados parciales de los workers, logrando con ello la preservación del orden de los datos. Los workers son el elemento de procesamiento y tienen un comportamiento proactivo: cuando no están realizando una tarea de procesamiento, solicitan al manager una tarea a realizar, lo cual repiten hasa que no hay más tareas a realizar.

La dinámica del patrón es el siguiente:

- Los paricipantes son creados y están a la espera de realizar cómputos. Cuando el manager dispone de los datos, los distribuye de acuerdo a la solicitud de cada worker.
- Los workers reciben el dato y comienzan a procesarlo. Cuando terminá la operación, el worker envía el resultado al manager y solicita más datos. Este proceso se repite hasta que ya no existan datos para repartir.
- El manager responde las solicitudes de datos de los workers y recibe sus resultados parciales. Cuando todos los datos han sido procesados, el manager obtiene el resultado total a partir de los resultados parciales.

En el Sección 1.4 se señaló que las operaciones primitivas de un actor son las de crear nuevos actores, enviar mensajes y cambiar su estado. Estas primitivas se acoplan de forma importante con el patrón *Manager-Workers*. Los actores pueden crear una cantidad finita de actores, definiendo con ello una jerarquía de actores en la que, a excepción del actor raíz, todos tendrán un padre y podrán tener uno o más hijos. Esta jerarquía de actores que comienza con el actor raíz se le denomina un *sistema de actor*. El sistema de actor tiene una correspondencia fuerte con el patrón *Manager-Workers* en términos que una de las responsabilidades del manager es la de crear workers.

El patrón *Manager-Workers* aplicado a un sistema de actor ofrece la ventaja que evita el problema de desbordamiento del buzón de los workers, dado que el control de la velocidad del flujo de trabajo está administrada por las solicitudes de los workers y de la respuesta del manager. De manera que el buzón de los workers tendrá siempre una cantidad pequeña de mensajes [15]. Sin embargo, el manager necesita ser capaz de controlar el flujo de trabajo, de lo contrario el problema de desbordamiento del buzón se trasladaría de los workers al manager, generando un cuello de botella que se propaga en todo el sitema de actor.

Para nuestra solución paralela del cálculo del coeficiente de clustering por nodo (c_i) de una red no dirigida, utilizaremos el patrón *Manager-Workers*. Cómo se explicó en el Capítulo 2, el coeficiente de clustering por nodo el puede ser calculado

con la expresión $c_i = \frac{(A)^3_{ij}}{k_i(k_i-1)}$. De manera que necesitamos calcular los elementos de la diagonal principal de A^3 y el número de conexiones por cada nodo.

A continuación explicamos las decisiones tomadas en cada punto de los propuestos por Foster [3] para la solución al problema planteado.

4.1.2. Particionamiento

El manager realiza un particionamiento doble de la matriz de adyacencia por fila (*Row-wise Array Partitioning*) y por columna (*Row-wise Array Partitioning*) [1] . El manager proporciona una fila de A por cada solicitud de los workers.

4.1.3. Comunicación

Los workers establecen comunicación con el manager mediante envíos de mensajes. Inmediatamente que se crea el sistema de actor, los workers comienzan a solicitar filas de A al manager. Para que los workers puedan calcular $A^2_{i\bullet}$, estos necesitan las columnas de la matriz A. Por tal motivo, el manager proporciona una columna de A por cada solicitud de los workers.

Cuando el worker ha calculado toda la fila completa $A^2_{i\bullet}$, este calcula A^3_{ii} con la multiplicación de $A^2_{i\bullet}$ por $A_{i\bullet}$. Con $A_{i\bullet}$ el worker calcula también la cantidad de conexiones por nodo, que, para el caso del nodo i, es igual a la cantidad de unos en $A_{i\bullet}$.

El worker calcula c_i, lo envía al manager y le solicita una nueva fila de A.

[1]Estas decisión está basada en la propuesta de Foster [3] de la multiplicación paralela de matrices mediante una descomposición en dos dimensiones en la que una tarea encapsula elementos correspondientes de la matrices que pre y pos multiplican (A y B), así como de la matriz resultante (C), y que cada tarea es responsable del cálculo asociado con los elementos que le corresponden de C_{ij}. Como el cálculo de un elemento de C_{ij} requiere la fila completa $A_{i\bullet}$ y la columna completa $B\bullet j$, de A y de B, necesitamos que el manager esté suministrando las respectivas columnas de B a las tareas para realizar los cálculos.

La implementación se realizará de la siguiente manera:

- El manager proporciona una fila de A por cada solicitud de los workers.
- El manager proporciona una columna de A por cada solicitud de los workers para que realicen el cálculo de $A^2_{i\bullet}$.
- El worker calcula A^3_{ii} con la multiplicación de $A^2_{i\bullet}$ por $A_{i\bullet}$.
- El worker calcula la cantidad de conexiones por nodo, que, para el caso del nodo i, es igual a la cantidad de unos en $A_{i\bullet}$.
- El worker calcula c_i, lo envía al manager y le solicita una nueva fila de A.
- Si ya no hay filas que distribuir a los workers, el programa termina.

Se presenta el proceso anterior en un diagrama de objetos (Ver Figura 4.1).

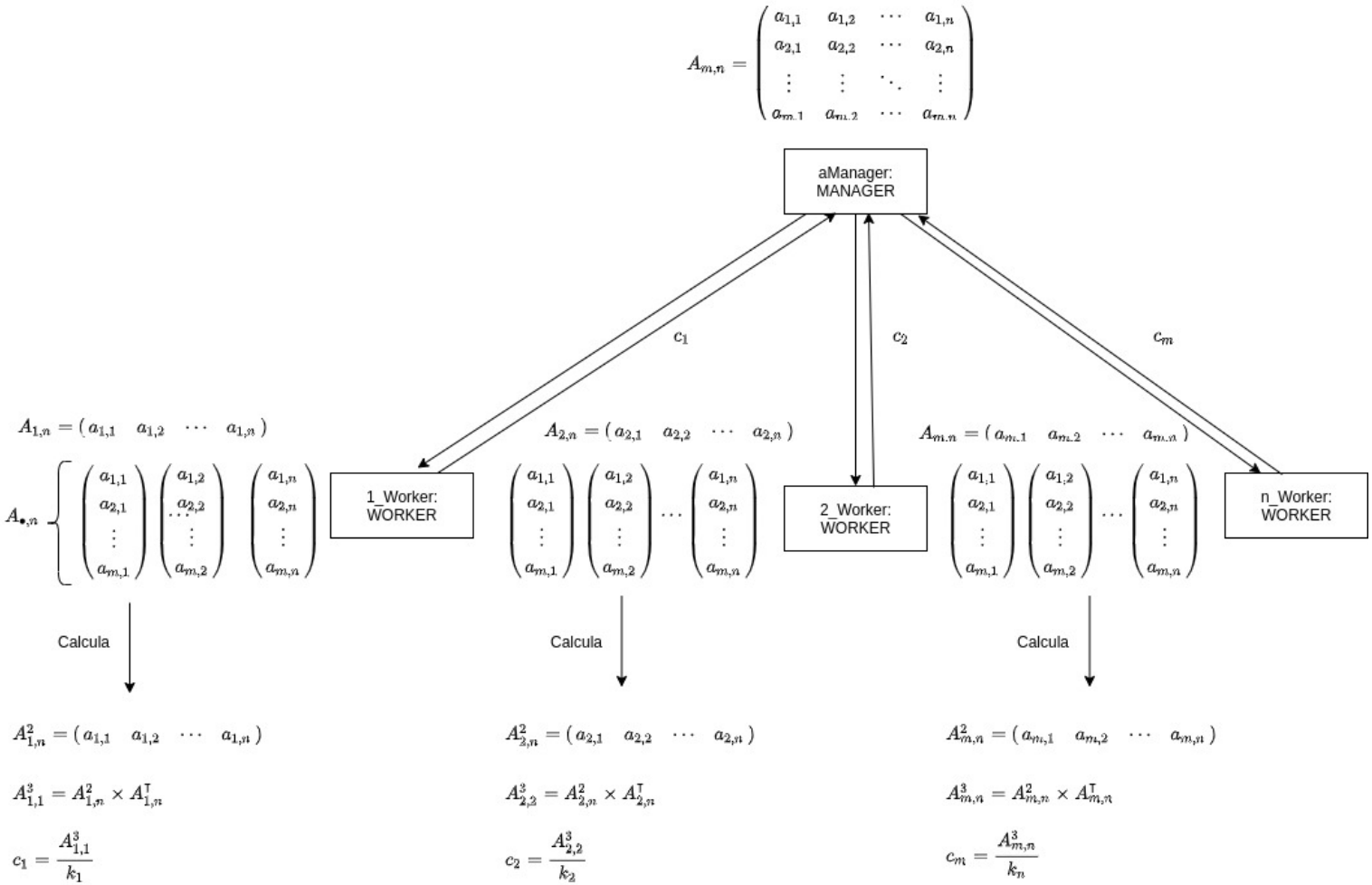


Figura 4.1: Diagrama de objetos de la propuesta de cálculo

Capítulo 5

Resultados

En esta sección se presentan y analizan los resultados del desempeño de la implementación de CAF y Akka de la propuesta de cálculo del coeficiente de clustering de los nodos de una red no dirigida mediante el patrón arquitectónico Manager-Workers, con el objetivo de conocer qué implementación tiene mejor desempeño en tiempo de ejecución. Se realizan una evaluación a nivel local y otra de forma distribuida[1]. Para la primera obtenemos que la implementación en Akka tiene un mejor desempeño en términos de tiempo ejecución, además de presentar speedup y eficiencia superior a la implementación en CAF. Por su parte, para la evaluación distribuida los resultados muestran que la implementación en CAF presenta un mejor desempeño tanto en tiempo de ejecución, speedup y eficiencia cuando agregamos un nodo worker adicional, mientras que en Akka empeora los tres indicadores.

5.1. Prueba a nivel local

La primera prueba se realiza de forma local en un equipo con 6 cores Intel(R) Core(TM) i7-8750H CPU 2.20GHz con 2 hilos de procesamiento por nucleo con 12Gb en RAM, corriendo en Ubuntu 18.04.4 LTS.

Se calculó el coeficiente de clustering para 1000 nodos, de manera que la matriz de adyacencia es de dimensión 1000 X 1000. El cálculo se realizó definiendo

[1]Los scripts para replicar estos resultados se pueden encontrar en el siguiente sitio `https://github.com/milocortes/actor-model-comp-caf-akka`

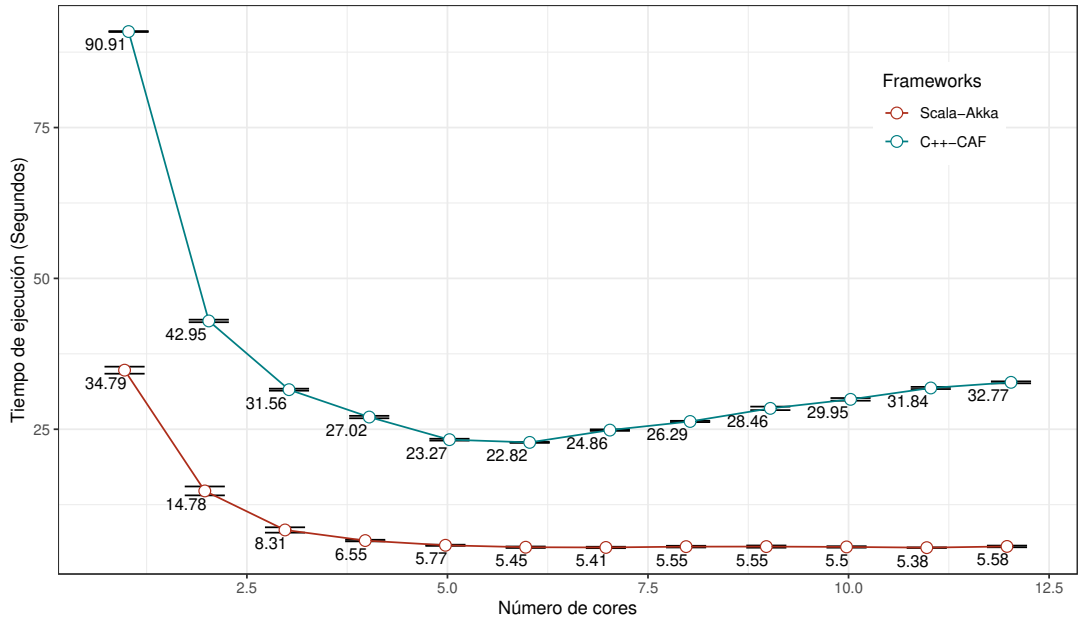
100, 250, 500 y 1000 actores, teniendo que cada actor se encargará de calcular el coeficiente de 10 nodos, 4 nodos, 2 nodos y 1 nodo, respectivamente. Se realiza este cálculo usando de 1 a 12 cores. Por cada prueba se realizaron 10 ejecuciones.

La Figura 5.1 presenta los comparativos a nivel local de los tiempos de ejecución. En términos generales se observa que en todos los casos la implementación en Akka tiene un tiempo de ejecución menor a la implementación en CAF. Analizando el desempeño individual de cada framework, tenemos que Akka no presenta una disminución en los tiempos de ejecución al agregar más actores. En cambio, CAF presenta disminuciones significativas al aumentar la cantidad de actores. Cuando se definen 1000 actores, el desempeño de CAF es similar al de Akka, e incluso supera su desempeño en uno de los casos.

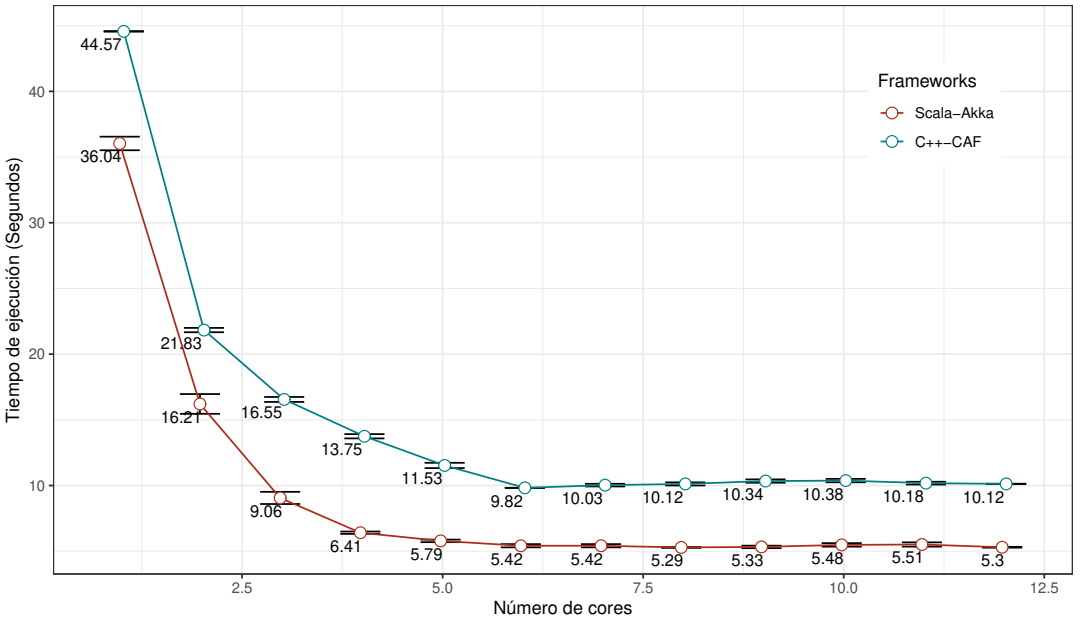
La Figura 5.2 presenta el comparativo del speedup a nivel local. En todos los casos, el speedup de la implementación de Akka supera al de CAF. Llama la atención que para 100 y 250 actores, al utilizar de 2 a 6 cores, el speedup correspondientes a Akka es mayor al speedup ideal.

En cuanto a la eficiencia (Figura 5.3), se observa que en todos los casos para la implementación de Akka este indicador es mayor al de CAF. Por otro lado, en la mayoría la eficiencia en Akka es superior al 50 %.

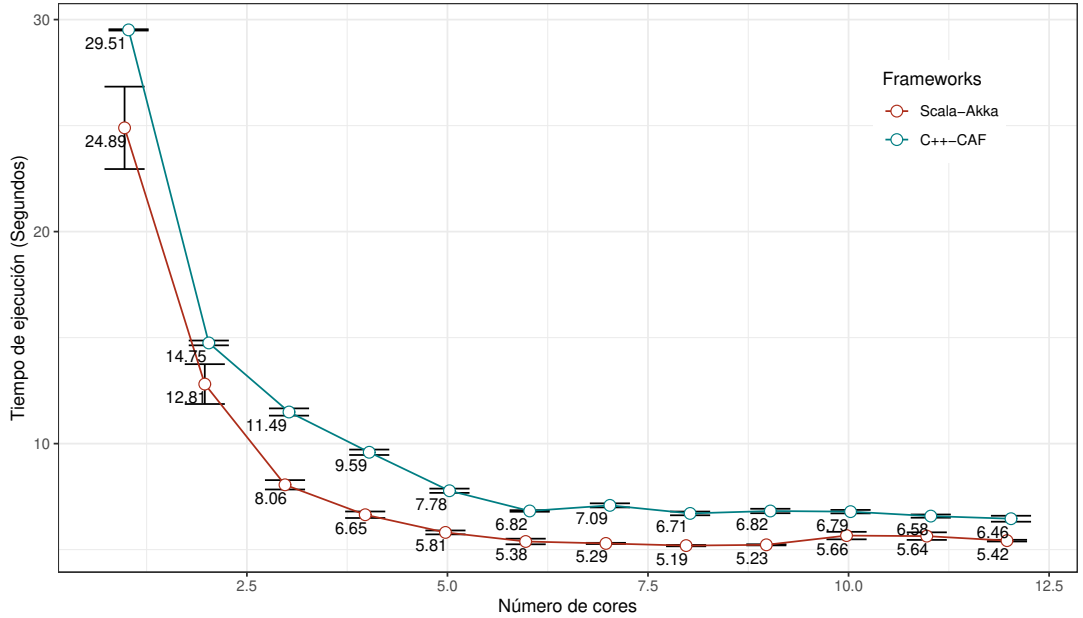
Considerando estos resultados, se cuenta con evidencia para indicar que a nivel local el cálculo de cálculo del coeficiente de clustering, con el hardware y la propuesta de cálculo dada, tiene un mejor desempeño en términos de tiempo ejecución, además de presentar speedup y eficiencia superior a la implementación en CAF.



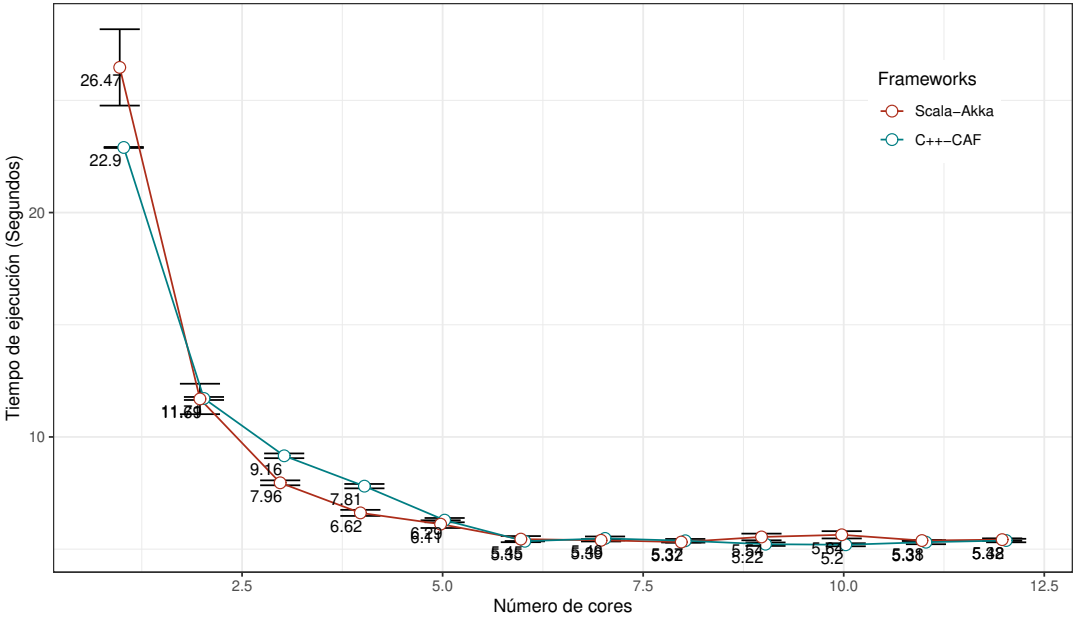
(a) 100 actores



(b) 250 actores

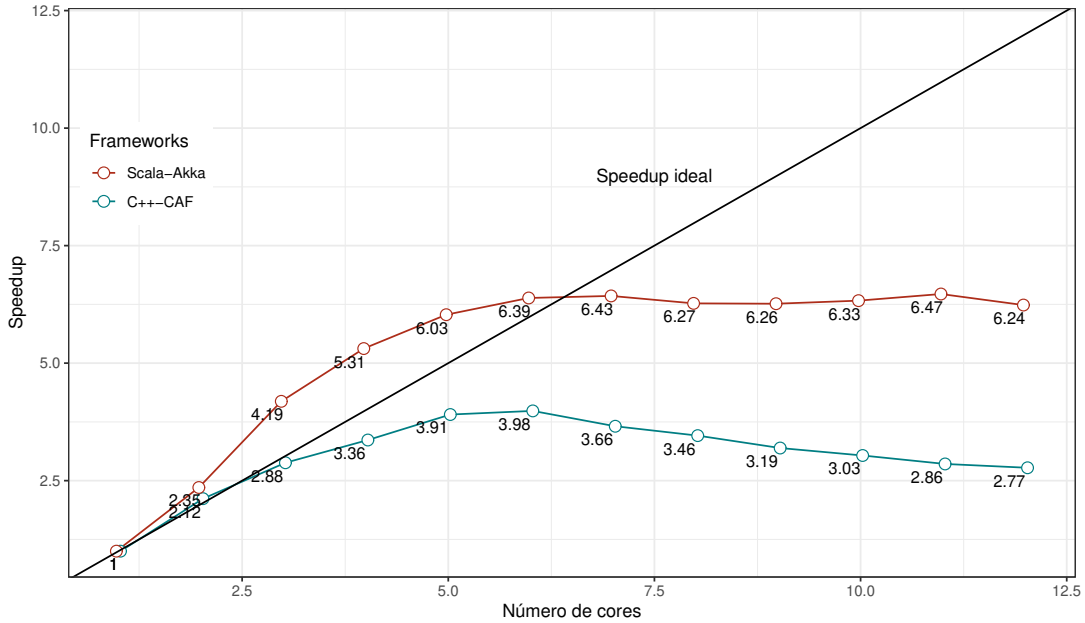


(c) 500 actores

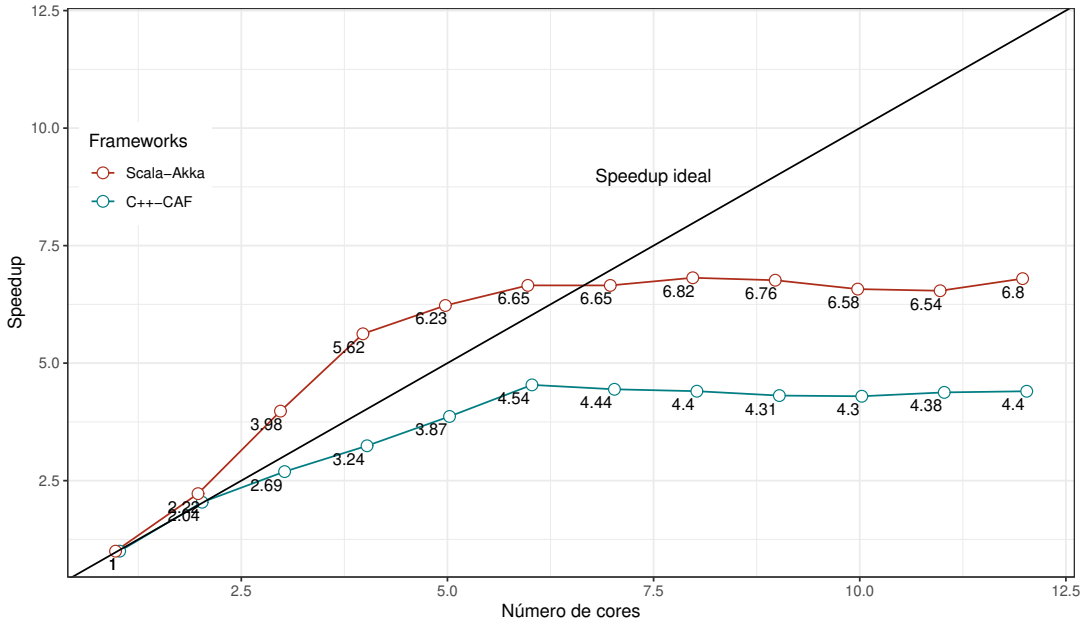


(d) 1000 actores

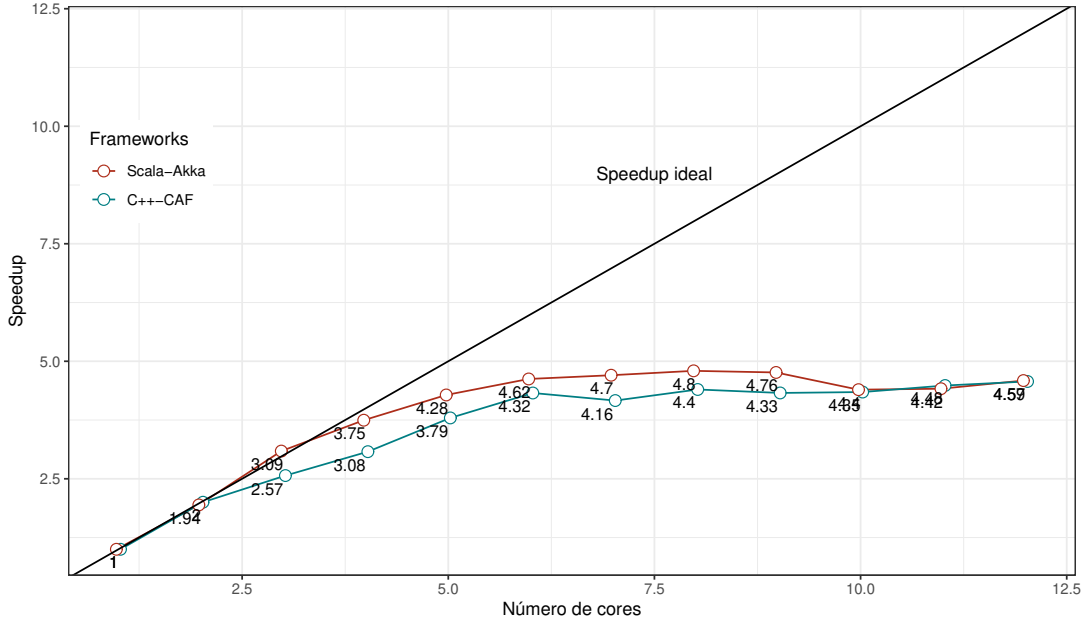
Figura 5.1: Comparación del tiempo de ejecución de los frameworks para el cálculo del coeficiente de clustering. Matriz 1000 X 1000.



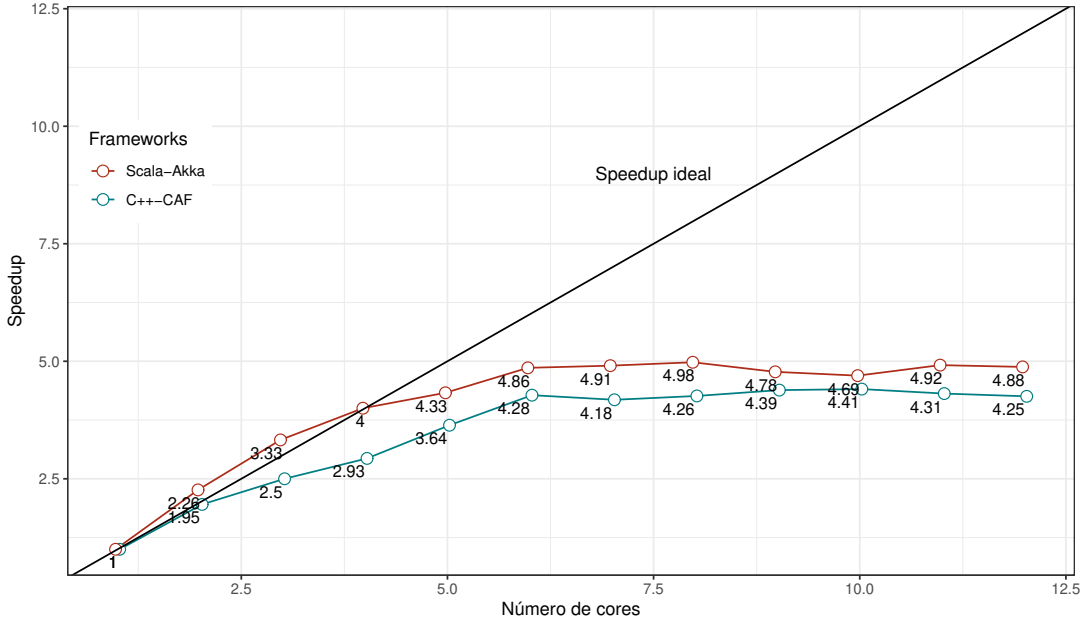
(a) 100 actores



(b) 250 actores

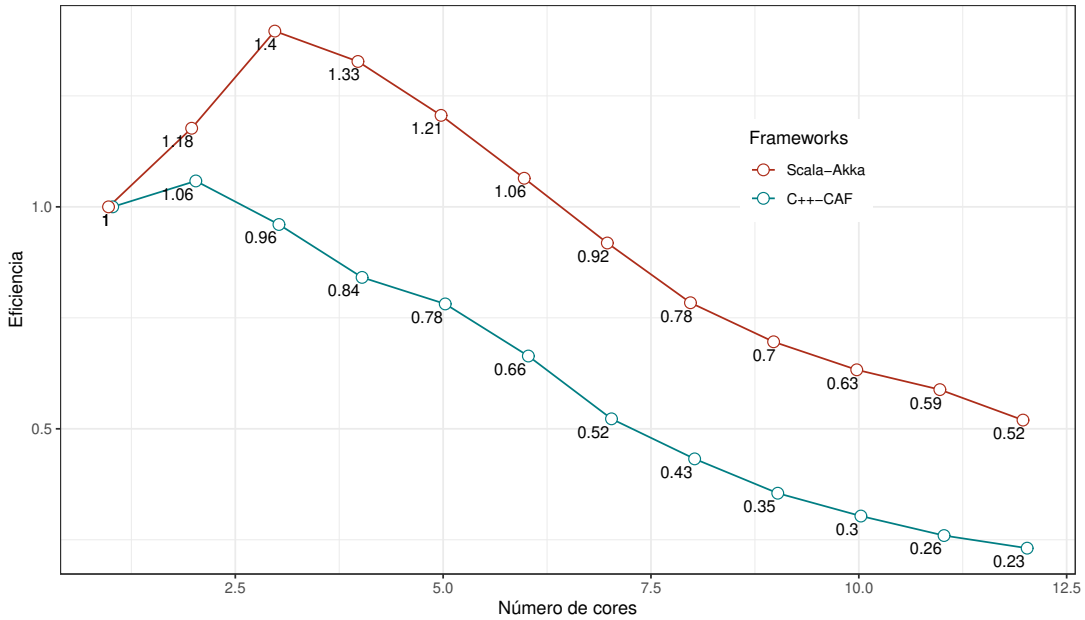


(c) 500 actores

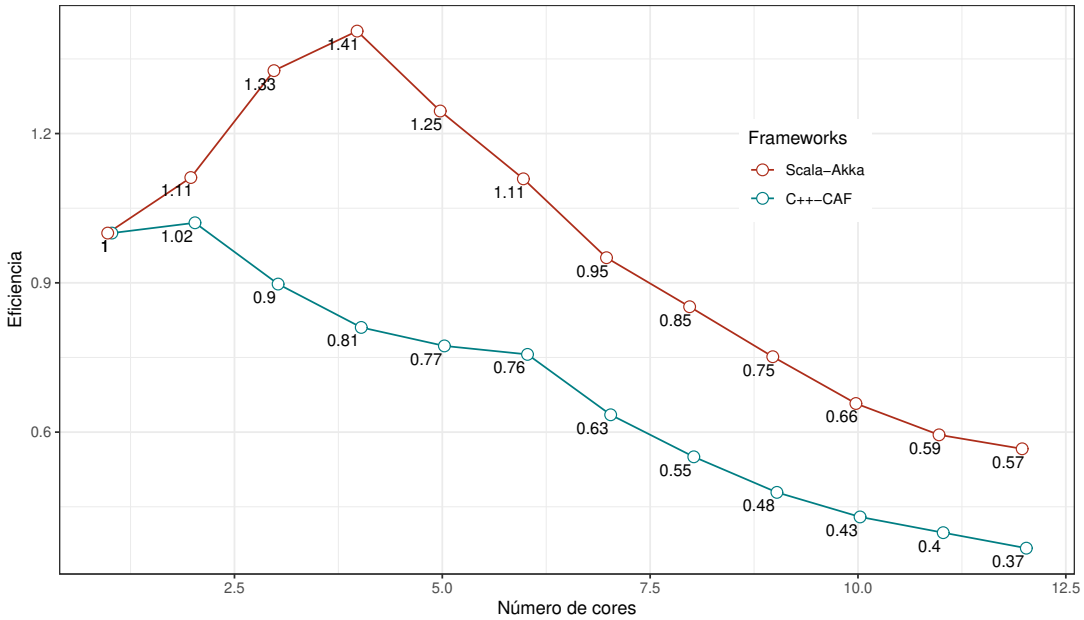


(d) 1000 actores

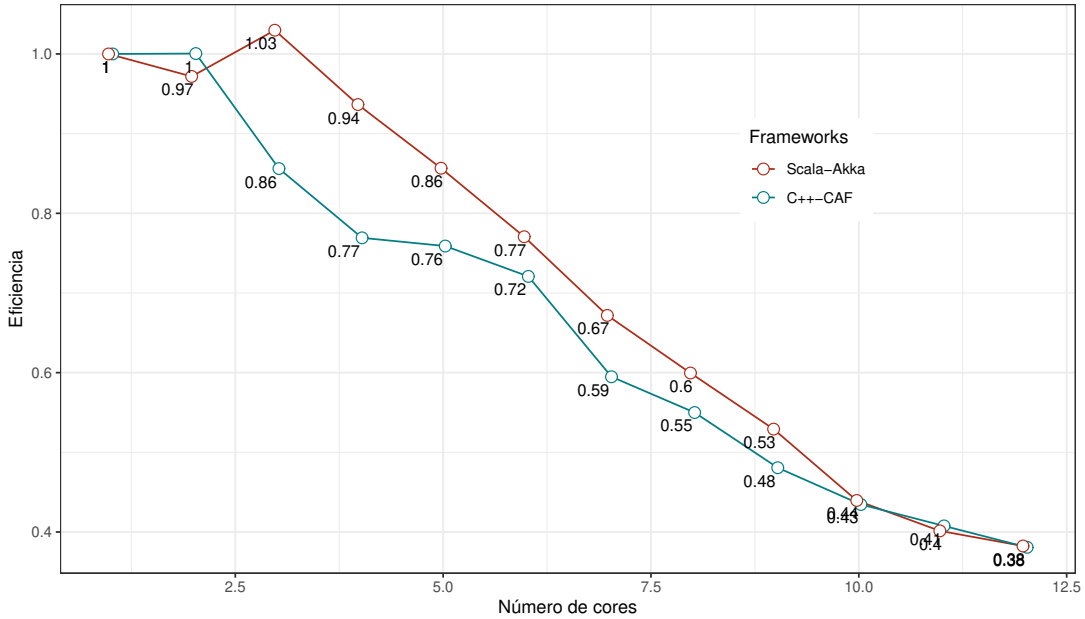
Figura 5.2: Comparación del Speedup de los frameworks para el cálculo del coeficiente de clusteríng. Matriz 1000 X 1000.



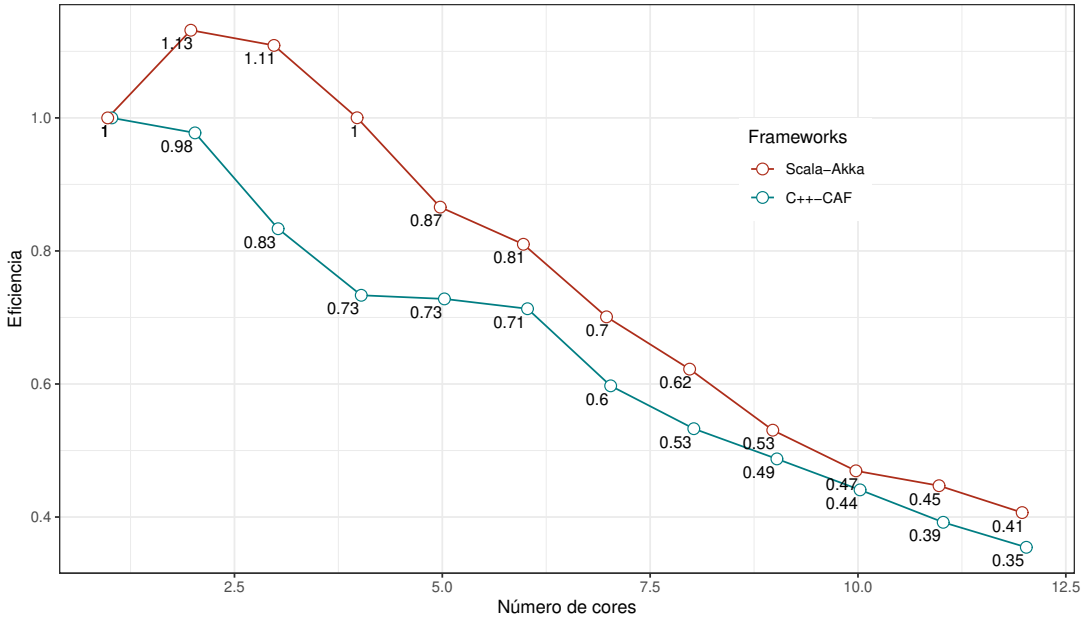
(a) 100 actores



(b) 250 actores



(c) 500 actores



(d) 1000 actores

Figura 5.3: Comparación de la Eficiencia de los frameworks para el cálculo del coeficiente de clusteríng. Matriz 1000 X 1000.

5.2. Prueba de forma distribuida

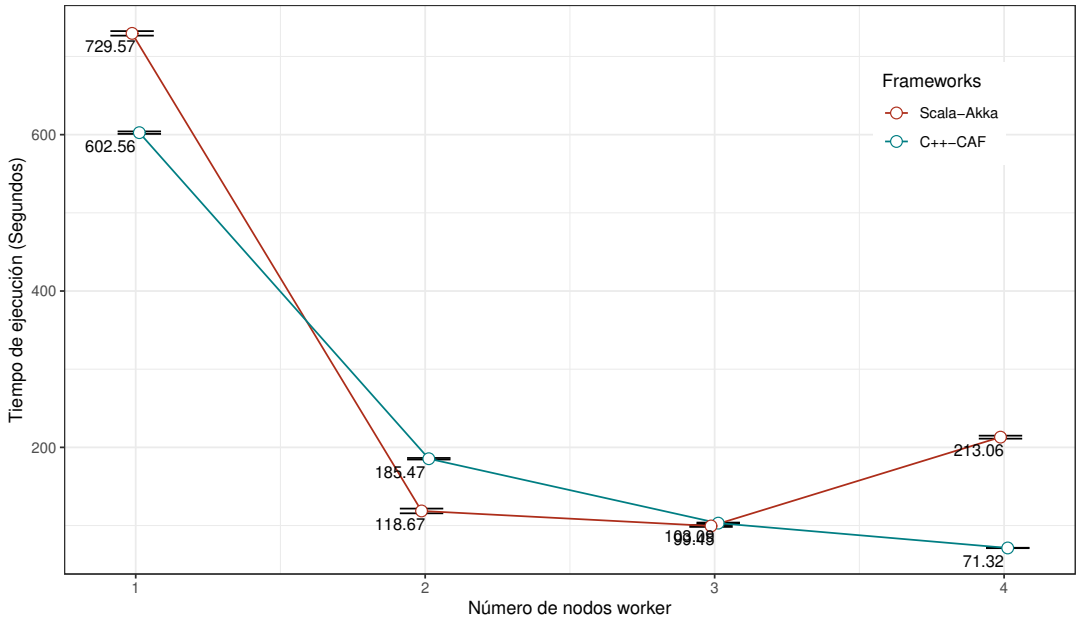
El segundo comparativo se realiza en un ambiente distribuido. Para el nodo Manager se utilizó el mismo equipo en el que se realizó la prueba local. Para los nodos workers utilizamos Raspberry pi 4 corriendo en Raspbian GNU/Linux 10 (buster) con kernel Linux 4.19.75-v7l+, con un socket con 4 cores.

Para esta prueba el cálculo del coeficiente de clustering se realizó para 600 nodos, de manera que la matriz de adyacencia es de dimensión 600 X 600. Para cada nodo worker se definen 50 actores. Por cada prueba se realizaron 10 ejecuciones independientes.

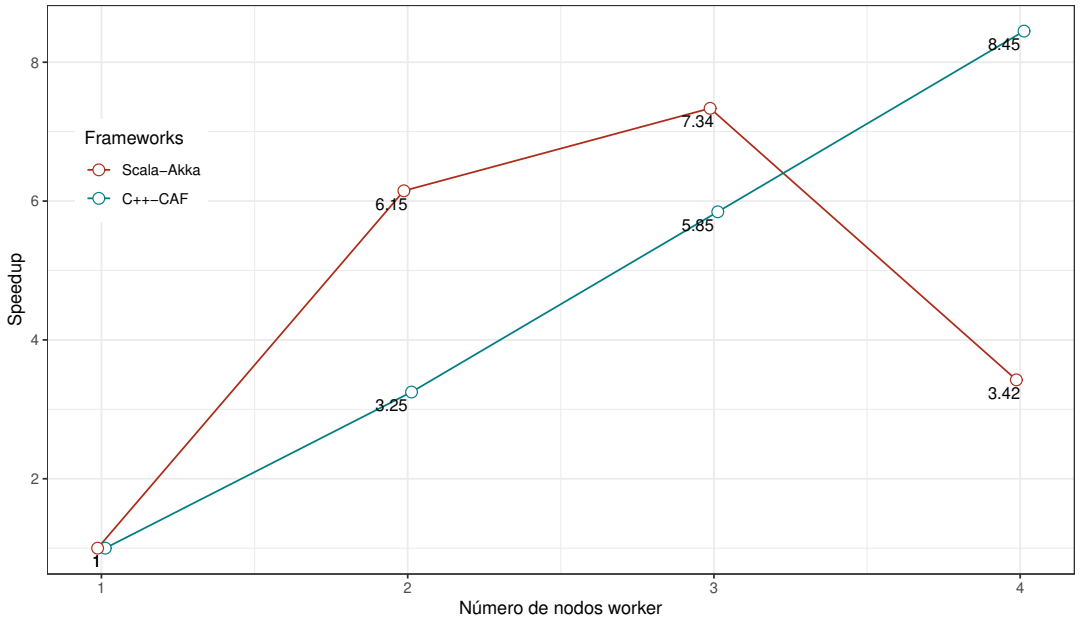
La Figura 5.4 presenta los resultados del tiempo de ejecución, speedup y eficiencia para el entorno distribuido. Cuando consideramos un nodo worker, el tiempo de ejecución de la implementación de CAF es menor por casi dos minutos que el tiempo de ejecución de la ejecución en Akka. Considerando 2 nodos worker, el tiempo de ejecución en Akka es menor a la de CAF por casi un minuto. Para el caso de tres nodos worker, los tiempos de ejecución son prácticamente los mismos para ambos frameworks. Con cuatro nodos worker, se presenta que el tiempo de ejecución de la CAF continua disminuyendo, presentando el tiempo de ejecución más bajo de todas las ejecuciones, mientras que en Akka el tiempo de ejecución se incrementa, quedando en un nivel superior al de la ejecución con dos nodos worker.

Con respecto al comportamiento del speedup y la eficiencia para la implementación distribuida, se observa que para el caso de dos y tres nodos worker, la implementación en Akka supera a la de CAF. Sin embargo, al agregar el cuarto nodo worker, en Akka el speedup y la eficiencia presentan una caida importante que los coloca en niveles inferiores a los resultados de la implementación en CAF. Además, en Akka el máximo speedup se alcanza con tres nodos, mientras que la mayor eficiencia se presenta con dos nodos. Para el caso de CAF agregar el cuarto nodo continúa mejorando el speedup y la eficiencia.

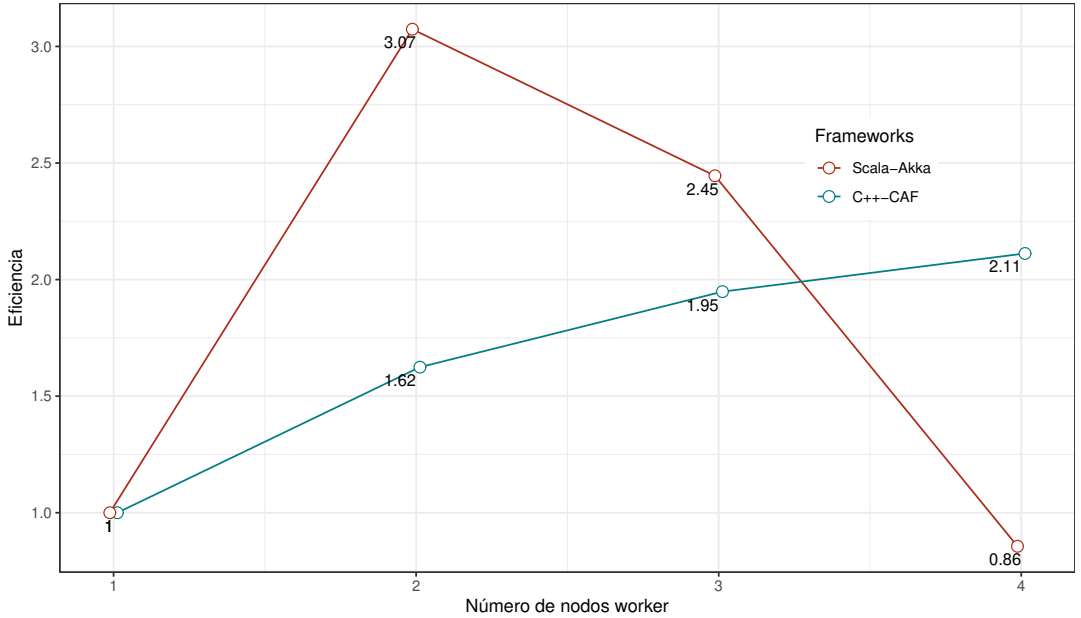
Dado que al agregar un cuarto nodo a la ejecución sólo genera una mejora tanto en el tiempo de ejecución como en el speedup y la eficiencia de la implementación en CAF, podemos indicar que, para la versión distribuida, ésta es la implementación que tiene el mejor desempeño.



(a) Tiempo de ejecución



(b) Speedup



(c) Eficiencia

Figura 5.4: Comparación de los frameworks para el cálculo del coeficiente de clustering en versión distribuida. 50 actores por nodo. Matriz 600 X 600.

5.3. Conclusiones

En esta sección describimos los resultados de realizar dos pruebas, una local y otra distribuida, a los frameworks CAF y Akka para la implementación de la propuesta de cálculo del coeficiente de clustering de los nodos de una red no dirigida mediante el patrón arquitectónico Manager-Workers.

Para la prueba local, los resultados indican que la implementación en Akka tiene un mejor desempeño en términos de tiempo ejecución, además de presentar speedup y eficiencia superior a la implementación en CAF. Para la prueba distribuida los resultados muestran que la implementación en CAF presenta un mejor desempeño tanto en tiempo de ejecución, speedup y eficiencia cuando agregamos un nodo worker adicional, mientras que en Akka estos tres indicadores empeoran, motivo por el cual podemos indicar que CAF tiene un mejor desempeño en para la prueba distribuida.

Capítulo 6

Conclusiones

Se realizó un análisis comparativo entre los frameworks CAF y Akka para conocer cuál de estos tiene un mejor desempeño en términos de tiempo de ejecución para el cálculo del coeficiente de clustering de los nodos de una red no dirigida. Específicamente, el cuestionamento a responder es conocer cuál framework tiene un mejor desempeño en un cluster, considerando como medida de desempeño el tiempo de ejecución en segundos.

Para responder esta pregunta, se propuso una solución parallela para el cálculo del coeficiente de clustering de los nodos de una red no dirigida, utilizando el patrón arquitectónico *Manager-Workers* y el modelo de actor (Ver Capítulo 4). Se implementó esta solución con los frameworks CAF y Akka. Para evaluar el desempeño de las dos implementaciones, realizamos una prueba a nivel local y otra de forma distribuida. Para la primera obtenemos que la implementación en Akka tiene un mejor desempeño en términos de tiempo ejecución, además de presentar speedup y eficiencia superior a la implementación en CAF. Para la prueba distribuida los resultados muestran que la implementación en CAF presenta un mejor desempeño tanto en tiempo de ejecución, speedup y eficiencia cuando agregamos un nodo worker adicional, mientras que en Akka empeora los tres indicadores.

De acuerdo a nuestros resultados, se cuenta con evidencia para responder que CAF muestra un mejor desempeño en un cluster en términos de tiempo de ejecución para la propuesta de cálculo del coeficiente de clustering de una red no dirigida.

Uno de los pendientes de esta tesina fue la de incluir una mayor cantidad de nodos worker. Por otra parte, queda abierta la investigación a explotar el uso de elementos particulares de cada frameworks con la intención de mejorar el tiempo de ejecución. Para el caso de CAF, resulta interesante realizar una comparación explotando la posibilidad que ofrece el framework de utilizar OpenCL para la creación del actores en GPUs. Por su parte, queda abierta la posibilidad de utilizar serializadores personalizados en Akka para mejorar el desempeño en forma distribuida.

Bibliografía

[1] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Revisiting actor programming in c++," *Computer Languages, Systems & Structures*, vol. 45, pp. 105–131, 2016.

[2] F. D. Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, *et al.*, "A survey of active object languages," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–39, 2017.

[3] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[4] C. Hewitt, P. Bishop, and R. Steiger, "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence," in *Advance Papers of the Conference*, vol. 3, p. 235, Stanford Research Institute, 1973.

[5] G. A. Agha, "Actors: A model of concurrent computation in distributed systems.," tech. rep., Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.

[6] G. A. Agha and W. Kim, "Actors: A unifying model for parallel and distributed computing," *Journal of systems architecture*, vol. 45, no. 15, pp. 1263–1277, 1999.

[7] G. Agha and W. Y. Kim, "Parallel programming and complexity analysis using actors," in *Proceedings. Third Working Conference on Massively Parallel Programming Models (Cat. No. 97TB100228)*, pp. 68–79, IEEE, 1997.

[8] L. Rinaldi, M. Torquati, G. Mencagli, M. Danelutto, and T. Menga, "Accelerating actor-based applications with parallel patterns," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 140–147, IEEE, 2019.

[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996.

[10] J. L. Ortega-Arjona, *Patterns for Parallel Software Design*. Wiley Publishing, 1st ed., 2010.

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

[12] K. Erciyes, *Guide to Graph Algorithms: Sequential, Parallel and Distributed*. Springer Publishing Company, Incorporated, 1st ed., 2018.

[13] V. Latora, V. Nicosia, and G. Russo, *Complex Networks: Principles, Methods and Applications*. Cambridge University Press, 2017.

[14] R. Hiesgen, D. Charousset, and T. C. Schmidt, "Opencl actors–adding data parallelism to actor-based programming with caf," in *Programming with Actors*, pp. 59–93, Springer, 2018.

[15] M. Nash and W. Waldron, *Applied Akka Patterns: A Hands-on Guide to Designing Distributed Applications*. .°'Reilly Media, Inc.", 2016.

Apéndice A

Implementaciones

A.1. Implementación local con CAF

caf-local-coef-clustering.cpp

```
/*
Versión local para el cálculo del coeficiente de clustering de una red dirigida con el patrón arquitectónico
Manager−Workers y el modelo de actor.

Ejecutar como sigue:

./caf−actor−car−V00 w d

donde w es la cantidad de workers y d es la cantidad de nodos
*/
#include <cassert>
#include <cstdint>
#include <set>
#include <string>
#include <utility>
#include <iostream>
#include <vector>
#include <fstream>
#include <boost/range/irange.hpp>
#include <boost/range/algorithm_ext/push_back.hpp>
#include <boost/range/numeric.hpp>

#include "caf/init_global_meta_objects.hpp"
#include "caf/all.hpp"

// Se definen los mensajes personalizados como struct
struct filas_asignadas ;
```

```
28 struct receive_worker_row_col;
29 struct calcula_coef_data;
30
31 // Agregamos los mensajes al bloque principal de CAF
32 // También agregamos los atoms, los cuales definen el comportamiento del actor al recibir un mensaje.
33 CAF_BEGIN_TYPE_ID_BLOCK(mensajes_personalizados, first_custom_type_id)
34
35   CAF_ADD_ATOM(mensajes_personalizados, get_id_atom)
36   CAF_ADD_ATOM(mensajes_personalizados, envia_indice_fila_atom)
37   CAF_ADD_ATOM(mensajes_personalizados, comenzar_cal_coef_atom)
38   CAF_ADD_ATOM(mensajes_personalizados,calcula_coef_atom)
39   CAF_ADD_ATOM(mensajes_personalizados,recibe_coef_atom)
40   CAF_ADD_TYPE_ID(mensajes_personalizados, (filas_asignadas))
41   CAF_ADD_TYPE_ID(mensajes_personalizados, (receive_worker_row_col))
42   CAF_ADD_TYPE_ID(mensajes_personalizados, (calcula_coef_data))
43
44 CAF_END_TYPE_ID_BLOCK(mensajes_personalizados)
45
46 using namespace std;
47 using namespace caf;
48
49
50 // Definimos los atributos de los mensajes
51 // Mensaje filas_asignadas
52 struct filas_asignadas {
53   std::vector<unsigned short int> a;
54 };
55
56 // Mensaje receive_worker_row_col
57 struct receive_worker_row_col{
58   std::vector<unsigned short int> fila;
59   std::vector<unsigned short int> columna;
60   unsigned short int indexFila;
61   unsigned short int indexColumna;
62 };
63
64 // Mensaje calcula_coef_data
65 struct calcula_coef_data{
66   unsigned short int index;
67   std::vector<unsigned short int> fila;
68 };
69
70 // Usamos la interface Inspector para serializar los mensajes.
71 // Serializamos el mensaje filas_asignadas
72 template <class Inspector>
73 typename Inspector::result_type inspect(Inspector& f, filas_asignadas & x) {
74   return f(meta::type_name("filas_asignadas"), x.a);
75 }
76 // Serializamos el mensaje receive_worker_row_col
77 template <class Inspector>
```

```
78  typename Inspector::result_type inspect(Inspector& f, receive_worker_row_col& x) {
79    return f(meta::type_name("receive_worker_row_col"), x. fila ,x.columna,x.indexFila, x.indexColumna);
80  }
81  // Serializamos el mensaje calcula_coef_data
82  template <class Inspector>
83  typename Inspector::result_type inspect(Inspector& f, calcula_coef_data& x) {
84    return f(meta::type_name("calcula_coef_data"), x.index,x. fila );
85  }
86
87  namespace{
88    /*
89    CAF proporciona distintas implementaciones para los actores, las cuales difieren en
90    tres características:
91    1) Tipado dinámico o estático.
92    2) Basado en clases o basado en funciones.
93    3) Un manejador de eventos asíncrono o recepciones bloqueantes.
94
95    Estas tres características pueden ser combinadas con libertad, con solo una excepción:
96    los actores estáticamente tipados son siempre basados en eventos.
97
98    Una ventaja de usar actores estáticamente tipados es que permite al compilador verificar la
99    comunicación entre los actores. Por esta razón, para la implementación se decidió utilizar
100   este tipo de actores.
101
102   Al utilizar actores estáticamente tipados el framework exige implementar una interface para los mensajes
103   para permitir al compilador checar los tipos de los mensajes de la comunicación de los actores. Para
          parámetro
104   del template define un insumo y un producto (e.g. replies_to<X1,...,Xn>::with<Y1,...,Yn>). Cuando hay
          insumos que
105   no generan una salida, se utiliza reacts_to<X1,...,Xn>.
106   */
107
108   // Definimos la interface de mensajes del actor manager
109   using manager_actor = typed_actor<replies_to<get_id_atom>::with<filas_asignadas>,
110                               replies_to <envia_indice_fila_atom ,unsigned short int ,unsigned short
                                    int >::with<receive_worker_row_col>,
111                               replies_to <calcula_coef_atom, unsigned short int>::with<calcula_coef_data>,
112                               reacts_to<recibe_coef_atom,unsigned short int, float >>; //manager_actor
113
114   // Definimos la interface de mensajes del actor worker
115   using worker_actor = typed_actor<reacts_to<comenzar_cal_coef_atom>>; //worker_actor
116
117   /*
118   Se mencionó que definimos usar actores estaticamente tipados. Adicionalmente, estos actores están basados
          en funciones.
119   CAF proporciona stateful actors para facilitar el mantener el estado de los actores basados en funciones.
          Se define el struct
120   que contendrá el estado del actor, como sus atributos y métodos. Posteriormente, usamos behavior_type
          que es un conjunto
```

```
121  estaticamente tipado de manejadores de mensajes para agregra el estado de los actores al comportamiento
         de los actores tipados.
122  */
123
124  // Definimos el struct del manager_state
125
126  struct manager_state{
127    // id funciona para determinar qué id le toca a cada solicitud del worker
128    unsigned short int id=0;
129    // Entero que indica la cantidad de actores
130    unsigned short int no_actores;
131    // Entero que indica la dimesión de la matriz
132    unsigned short int dim_mat;
133    // Contador de nodos recibidos para determinar cuándo el manager termina el programa
134    unsigned short int nodos_recibidos;
135
136    // Arreglo que contendrá la matriz de adyacencia
137    vector<vector<unsigned short int>> mat_adj;
138
139    void aumentaID() {
140      id+=1;
141    }
142
143    void aumentaNodosRecibidos(/* arguments */) {
144      nodos_recibidos+=1;
145    }
146    // Método que calcula qué filas le tocan a cada worker de acuerdo al id asignado
147    vector<unsigned short int> calculaFilas(unsigned short int n,unsigned short int p,unsigned short int id){
148      unsigned short int Nrow = n / p;
149
150      unsigned short int  filaInicio  = id * Nrow;
151      unsigned short int  filaFinal  ;
152
153      if (id < (p - 1)) {
154        filaFinal = ((id + 1) * Nrow) -1;
155      } else {
156        filaFinal = (n-1);
157      }
158
159      // Generamos un vector que contendrá los enteros entre filaInicio y filaFinal
160      std::vector<unsigned short int> vector;
161      boost::push_back(vector, boost::irange((unsigned short) filaInicio ,(unsigned short) (filaFinal+1)));
162
163      return vector;
164    }
165
166    // Método para leer el archivo de texto de la matriz de adyacencia
167    vector<vector<unsigned short int>> read_matrix(unsigned short int n){
168      fstream
             in("/home/milo/Documentos/CAR/2doSemestre/Seminario/scripts/tesina/scala/crearModelo/files/AdjMatrix_big.txt");
```

```
169         int rows=n;
170         int cols=n;
171         vector<vector<unsigned short int>> matrix(rows, vector<unsigned short int>(cols));
172         for (auto& row : matrix){
173           for (auto& cell : row){
174             in >> cell;
175           }
176         }
177         return matrix;
178       }
179
180     };//manager_state
181
182     // Definimos el struct del worker_state
183       struct worker_state{
184         // id del worker
185         unsigned short int id;
186         // dimensión de la matriz
187         unsigned short int dim_mat;
188         // Este contador nos dirá cuando ya podemos realizar el proceso de cálculo del coefiente
189         unsigned short int contador_break;
190         // Este contador nos acumulará cuántos Coeficientes de Clustering hemos enviado
191         unsigned short int coef_enviados;
192         // Este vectorde enteros contendrá las filas que calculará cada worker
193         vector<unsigned short int> filas;
194         // Este vector de enteros contendrá el arreglo en 1D de la matriz A2 parcial del worker
195         vector<unsigned short int> A2_1D_Par;
196         // Este vector de enteros contendrá el arreglo en 2D de la matriz A2 parcial del worker
197         vector<vector<unsigned short int>> A2_2D_Par;
198
199         void aumentaContador() {
200           contador_break+=1;
201         }
202
203         void aumentaCoefEnviados() {
204           coef_enviados+=1;
205         }
206         // Método para almacenar el producto punto de la multiplicación de vectores en un vector parcial
207         void insertaValor(unsigned short int valor,unsigned short int fila,unsigned short int
                columna,unsigned short int n) {
208           A2_1D_Par[(fila*n)+columna]=valor;
209         }
210
211         // Método para transformar un vector 1D a uno 2D
212         vector<vector<unsigned short int>> convertir2D(vector<unsigned short int> d1_vector,unsigned short
                int rows,unsigned short int cols){
213
214           vector<vector<unsigned short int>> d2_vector(rows, vector<unsigned short int>(cols));
215
216           for (int i = 0; i < rows; i++) {
```

```
217             for (int j = 0; j < cols; j++) {
218               d2_vector[i][j]=d1_vector[(i*cols) + j];
219             }
220           }
221           return d2_vector;
222         }
223
224       };// worker state
225
226     manager_actor::behavior_type type_checked_manager (manager_actor::stateful_pointer<manager_state>
          self,unsigned short int no_actores,unsigned short int dim_mat ) {
227       self->state.no_actores=no_actores;
228       self->state.dim_mat=dim_mat;
229       self->state.mat_adj=self->state.read_matrix(dim_mat);
230
231      return {
232        // Implementamos el comportamiento del manager ante el mensaje get_id_atom
233        [=]( get_id_atom ) {
234
235          /*
236          Usamos el método calculaFilas para generar el vector con las filas que calculará el worker.
237          El manager responde a este mensaje del worker con el mensaje filas_asignadas.
238         */
239          vector<unsigned short int> v=self->state.calculaFilas(self->state.dim_mat,self->state.no_actores,self
                ->state.id);
240          self->state.aumentaID();
241
242          return filas_asignadas {v};
243        },
244        /* Implementamos el comportamiento del manager ante el mensaje envia_indice_fila_atom.
245           Este mensaje incorpora también el índice de la fila a solicitar así como el índice de la
246           columna a solicitar
247         */
248        [=]( envia_indice_fila_atom ,unsigned short int indexFila,unsigned short int indexColumna ) {
249
250          // Obtenemos el vector fila del índice que necesita el worker
251          std::vector<unsigned short int> fila= self->state.mat_adj[indexFila];
252          // Obtenemos el vector columna del índice que necesita el worker
253          std::vector<unsigned short int> columna= self->state.mat_adj[indexColumna];
254
255          return receive_worker_row_col{ fila , columna, indexFila,indexColumna};
256        },
257        /* Implementamos el comportamiento del manager ante el mensaje calcula_coef_atom.
258           Este mensaje incorpora también el índice de la fila a solicitar .
259         */
260        [=]( calcula_coef_atom,unsigned short int indexFila) {
261
262          // Obtenemos el vector fila del índice que necesita
263          std::vector<unsigned short int> fila= self->state.mat_adj[indexFila];
264
```

```
265         return calcula_coef_data{indexFila, fila };
266       },
267       /* Implementamos el comportamiento del manager ante el mensaje recibe_coef_atom, para el cual
268          el manager recibe el valor calculado del coeficiente de clustering de un nodo por parte del
269          worker
270        */
271       [=](recibe_coef_atom, unsigned short int indexNodo, float coef_clustering ) {
272
273         self->state.aumentaNodosRecibidos();
274         aout(self)<< "Nodo: "<< indexNodo << ". Coeficientes de Clustering: " <<coef_clustering <<". Nodos recibidos: "<< self->state.nodos_recibidos<< "\n";
275
276         // En caso que se hayan recibido los coeficientes de clustering de todos los nodos, el manager termina su ejecución
277         if (self->state.nodos_recibidos==self->state.dim_mat) {
278           aout(self)<< "He recibido todos los Coeficientes. Terminamos el programa" <<"\n";
279
280           self->quit();
281         }
282       }
283     };
284 }// type_checked_manager
285
286   worker_actor::behavior_type type_checked_worker (worker_actor::stateful_pointer <worker_state> self,unsigned short int id,unsigned short int dim_mat, manager_actor ma){
287     self->state.id=id;
288     self->state.dim_mat=dim_mat;
289
290     /*
291     El worker comienza solicitud al manager con el mensaje get_id_atom, el cual regresa un vector de enteros con
292     las filas que le corresponden a cada worker
293     */
294     self->request(ma,30s,get_id_atom_v).await(
295       [=]( filas_asignadas v){
296         self->state.filas=v.a;
297
298         // Hacemos un resize del vector A2_1D_Par con el vector recibido en el mensaje filas_asignadas
299         self->state.A2_1D_Par.resize((self->state.filas.size())*self->state.dim_mat);
300
301         /*
302         Por cada elemento en el vector de filas , se solicitar á al manager las columnas para calcular el producto punto
303         de este por cada vector columna de la matriz de adyacencia
304         */
305         for (unsigned short int x : self->state.filas){
306           for (unsigned short int j = 0; j < self->state.dim_mat; j++) {
307
308             /*
```

```
309             El worker solicita al manager el mensaje envia_indice_fila_atom, el cual regresa el vector fila
                    y columna
310             de los indices enviados por el worker. Con dichos vectores, el worker realiza el producto punto
                    y almancena
311             el resultado en el arreglo A2_1D_Par
312             */
313             self->request(ma,30s, envia_indice_fila_atom_v, x,j).await(
314               [=](receive_worker_row_col mensaje){
315
316                 // Obtenemos el producto punto de dos vectores y lo guardamos en A2_1D_Par
317                 unsigned short int
                        valor=inner_product(mensaje.fila.begin(),mensaje.fila.end(),mensaje.columna.begin(),0);
318                 self->state.insertaValor(valor, (x-(self->state.filas[0])), j, self->state.dim_mat);
319
320                 // Incrementamos el valor de contador_break en una unidad.
321                 self->state.aumentaContador();
322
323                 // Cuando el worker ha realizado todos los productos punto, puede comenzar a realizar el
                        cálculo del coefiente
324                 if (self->state.contador_break==(int)(self->state.filas.size()*self->state.dim_mat)) {
325
326                   // Convertimos A2_1D_Par en un arreglo 2D
327                   self->state.A2_2D_Par=self->state.convertir2D(self->state.A2_1D_Par,
                          (int)self->state.filas.size(),(int)self->state.dim_mat);
328
329                   /*
330                   Por cada una de las filas de este arreglo, el worker solicitará la fila correspondiente al
                          manager
331                   para calcular A3ii y la cantidad de conexiones del nodo. Con esto, puede calcular el
                          coeficiente
332                   de clustering del nodo. El resultado lo envía al manager mediante el mensaje
                          recibe_coef_atom
333                   */
334                   for(unsigned short int i: self->state.filas){
335                     /*
336                     Solicitamos al manager el mensaje calcula_coef_atom, el cual regresa la fila de la
                            matriz de adyacencia
337                     del índice que le enviamos.
338                     */
339                     self->request(ma,30s,calcula_coef_atom_v,i).await(
340                       [=](calcula_coef_data mensaje){
341
342                         // Obtenemos el producto punto para calcular A3ii
343                         unsigned short int
                                valor=inner_product(mensaje.fila.begin(),mensaje.fila.end(), self->state.A2_2D_Par[mensaje.index-(self
344                         // Obtenemos la cantidad de conexiones del nodo
345                         unsigned short int sum = boost::accumulate(mensaje.fila, 0);
346
347                         // Calculamos el coeficiente de clustering
348                         float coef_clustering = (float)valor/ (float)(sum*(sum-1));
```

```
349
350                         // Aumentamos el contador de coeficientes calculados
351                         self->state.aumentaCoefEnviados();
352
353                         // Enviamos el resultado al manager mediante el mensaje recibe_coef_atom
354                         self->send(ma,recibe_coef_atom_v,mensaje.index,coef_clustering);
355
356                         // Cuando el worker ha enviado todos sus coeficientes, termina su ejecución
357                         if (self->state.coef_enviados==(int)self->state.filas.size()) {
358                           std::cout << "El worker "<< self->state.id << " terminó el cálculo de sus
                                  coeficientes." << '\n';
359                           self->quit();
360                         }
361                       }
362
363                     );
364                   }
365                 }
366               }
367             );
368           }
369         }
370       }
371     );
372     return {
373       // Implementamos el comportamiento del manager ante el mensaje comenzar_cal_coef_atom
374       [=]( comenzar_cal_coef_atom ) {
375         std::cout << "Soy el worker "<< self->state.id<< " y comenzaré a calcular mis Coeficientes" <<
                '\n';
376         for (unsigned short int x : self->state.A2_1D_Par){aout(self) << x << " ";}
377       }
378     };
379   }// type_checked_worker
380
381 }
382
383
384
385 int main(int argc, char** argv) {
386
387   // Inicializamos la información global de los tipos de mensajes personalizados
388   init_global_meta_objects<id_block::mensajes_personalizados>();
389   core::init_global_meta_objects();
390
391   // Con actor_system_config configuramos la aplicación
392   actor_system_config cfg;
393
394   // Con actor_system pasamos esta configuración al sistema de actor
395   actor_system system{cfg};
396
```

```
397   scoped_actor self{system};
398
399   // Inicializamos no_actores y dim_mat con los valores recibidos del usuario
400   int no_actores= atoi(argv[1]);
401   int dim_mat= atoi(argv[2]);
402
403   // Calculamos e imprimimos la cantidad de filas que calculará cada worker
404   int Nrow=dim_mat/no_actores;
405   std::cout << Nrow << '\n';
406
407   // Generamos al manager
408   auto manager=self->spawn(type_checked_manager,no_actores,dim_mat);
409
410   /*
411    Generamos tantos workers como el valor recibido en no_actores. Inmediatamete que son
412   creados los workers, comienzan a solicitar sus filas al manager
413   */
414     for (size_t i = 0; i < (size_t)no_actores; i++) {
415       self->spawn(type_checked_worker,i,dim_mat,manager);
416     }
417
418 }
```

A.2. Implementación distribuida con CAF

caf-distribuido-coef-clustering.cpp

```
1  /*
2  Ejecutar modo manager ----> ./distributed-caf-car-V00 -s -w workerTotal -d nodos
3  Ejecutar modo worker ----> ./distributed-caf-car-V00 -w worker -d nodos
4
5  * -s indica que se está en modo server
6  * -d es la bandera de cantidad de nodos
7  * -w es la bandera de cantidad de workers
8  * nodos es la cantidad de nodos
9  * workerTotal es la cantidad total de workers a generar
10 * worker es la cantidad parcial de workers generados
11 */
12
13 #include <cassert>
14 #include <cstdint>
15 #include <set>
16 #include <string>
17 #include <utility>
18 #include <iostream>
19 #include <vector>
```

```
20 #include <fstream>
21 #include <boost/range/irange.hpp>
22 #include <boost/range/algorithm_ext/push_back.hpp>
23 #include <boost/range/numeric.hpp>
24
25 #include "caf/init_global_meta_objects.hpp"
26 #include "caf/all.hpp"
27 #include "caf/io/all.hpp"
28
29 // Se definen los mensajes personalizados como struct
30 struct filas_asignadas;
31 struct receive_worker_row_col;
32 struct calcula_coef_data;
33
34 // Agregamos los mensajes al bloque principal de CAF
35 // También agregamos los atoms, los cuales definen el comportamiento del actor al recibir un mensaje.
36 CAF_BEGIN_TYPE_ID_BLOCK(remote_types, first_custom_type_id)
37
38   CAF_ADD_ATOM(remote_types, get_id_atom)
39   CAF_ADD_ATOM(remote_types, set_manager_atom)
40   CAF_ADD_ATOM(remote_types, inicia_atom)
41   CAF_ADD_ATOM(remote_types, termina_atom)
42   CAF_ADD_ATOM(remote_types, envia_indice_fila_atom)
43   CAF_ADD_ATOM(remote_types, calcula_coef_atom)
44   CAF_ADD_ATOM(remote_types, recibe_coef_atom)
45
46   CAF_ADD_TYPE_ID(remote_types, (filas_asignadas))
47   CAF_ADD_TYPE_ID(remote_types, (receive_worker_row_col))
48   CAF_ADD_TYPE_ID(remote_types, (calcula_coef_data))
49
50 CAF_END_TYPE_ID_BLOCK(remote_types)
51
52 using namespace std;
53 using namespace caf;
54
55 // Definimos los atributos de los mensajes
56 // Mensaje filas_asignadas
57 struct filas_asignadas {
58   std::vector<unsigned short int> a;
59 };
60
61 // Mensaje receive_worker_row_col
62 struct receive_worker_row_col{
63   std::vector<unsigned short int> fila;
64   std::vector<unsigned short int> columna;
65   unsigned short int indexFila;
66   unsigned short int indexColumna;
67 };
68
69 // Mensaje calcula_coef_data
```

```
70  struct calcula_coef_data{
71    unsigned short int index;
72    std::vector<unsigned short int> fila;
73  };
74
75  // Usamos la interface Inspector para serializar los mensajes.
76  // Serializamos el mensaje filas_asignadas
77  template <class Inspector>
78  typename Inspector::result_type inspect(Inspector& f, filas_asignadas & x) {
79    return f(meta::type_name("filas_asignadas"), x.a);
80  }
81
82  // Serializamos el mensaje receive_worker_row_col
83  template <class Inspector>
84  typename Inspector::result_type inspect(Inspector& f, receive_worker_row_col& x) {
85    return f(meta::type_name("receive_worker_row_col"), x.fila ,x.columna,x.indexFila, x.indexColumna);
86  }
87
88  // Serializamos el mensaje calcula_coef_data
89  template <class Inspector>
90  typename Inspector::result_type inspect(Inspector& f, calcula_coef_data& x) {
91    return f(meta::type_name("calcula_coef_data"), x.index,x.fila );
92  }
93
94  namespace{
95    /*
96    CAF proporciona distintas implementaciones para los actores, las cuales difieren en
97    tres características:
98    1) Tipado dinámico o estático.
99    2) Basado en clases o basado en funciones.
100   3) Un manejador de eventos asíncrono o recepciones bloqueantes.
101
102   Estas tres características pueden ser combinadas con libertad, con solo una excepción:
103   los actores estáticamente tipados son siempre basados en eventos.
104
105   Una ventaja de usar actores estáticamente tipados es que permite al compilador verificar la
106   comunicación entre los actores. Por esta razón, para la implementación se decidió utilizar
107   este tipo de actores.
108
109   Al utilizar actores estáticamente tipados el framework exige implementar una interface para los mensajes
110   para permitir al compilador checar los tipos de los mensajes de la comunicación de los actores. Para
          parámetro
111   del template define un insumo y un producto (e.g. replies_to<X1,...,Xn>::with<Y1,...,Yn>). Cuando hay
          insumos que
112   no generan una salida, se utiliza reacts_to<X1,...,Xn>.
113   */
114
115   // Definimos la interface de mensajes del actor manager
116   using manager_actor = typed_actor<replies_to<get_id_atom>::with<filas_asignadas>,
```

```
117                         replies_to <envia_indice_fila_atom, unsigned short int, unsigned short
                                int>::with<receive_worker_row_col>,
118                         replies_to <calcula_coef_atom, unsigned short int>::with<calcula_coef_data>,
119                         reacts_to<recibe_coef_atom,unsigned short int, float>>;//manager_actor
120 // Definimos la interface de mensajes del actor worker
121 using worker_actor =
        typed_actor<reacts_to<set_manager_atom,actor>,reacts_to<termina_atom>,reacts_to<inicia_atom>>;
122 /*
123 Se mencionó que definimos usar actores estaticamente tipados. Adicionalmente, estos actores están basados
        en funciones.
124 CAF proporciona stateful actors para facilitar el mantener el estado de los actores basados en funciones.
        Se define el struct
125 que contendrá el estado del actor, como sus atributos y métodos. Posteriormente, usamos behavior_type
        que es un conjunto
126 estaticamente tipado de manejadores de mensajes para agregra el estado de los actores al comportamiento
        de los actores tipados.
127 */
128
129 // Definimos el struct del manager_state
130 struct manager_state{
131   // id funciona para determinar qué id le toca a cada solicitud del worker
132   unsigned short int id=0;
133   // Entero que indica la cantidad de actores
134   unsigned short int no_actores;
135   // Entero que indica la dimesión de la matriz
136   unsigned short int dim_mat;
137   unsigned short int suma_total;
138   unsigned short int contador_suma;
139   // Contador de nodos recibidos para determinar cuándo el manager termina el programa
140   unsigned short int nodos_recibidos;
141   // Arreglo que contendrá la matriz de adyacencia
142   vector<vector<unsigned short int>> mat_adj;
143
144   void aumentaID() {
145     id+=1;
146   }
147
148   void aumentaNodosRecibidos(/* arguments */) {
149     nodos_recibidos+=1;
150   }
151
152   // Método que calcula qué filas le tocan a cada worker de acuerdo al id asignado
153   vector<unsigned short int> calculaFilas(unsigned short int n,unsigned short int p,unsigned short int id){
154     unsigned short int Nrow = n / p;
155
156     unsigned short int  filaInicio  = id * Nrow;
157     unsigned short int  filaFinal  ;
158
159
160     if (id < (p - 1)) {
```

```
161         filaFinal = ((id + 1) * Nrow) -1;
162       } else {
163         filaFinal = (n-1);
164       }
165
166       // Generamos un vector que contendrá los enteros entre filaInicio y filaFinal
167       std::vector<unsigned short int> vector;
168       boost::push_back(vector, boost::irange((unsigned short) filaInicio ,(unsigned short) (filaFinal+1)));
169
170       return vector;
171     }
172     // Método para leer el archivo de texto de la matriz de adyacencia
173     vector<vector<unsigned short int>> read_matrix(unsigned short int n){
174       fstream
              in("/home/milo/Documentos/CAR/2doSemestre/Seminario/scripts/tesina/scala/crearModelo/files/AdjMatrix_big.txt");
175       int rows=n;
176       int cols=n;
177       vector<vector<unsigned short int>> matrix(rows, vector<unsigned short int>(cols));
178       for (auto& row : matrix){
179         for (auto& cell : row){
180           in >> cell;
181         }
182       }
183       return matrix;
184     }
185
186   };//manager_state
187
188   // Definimos el struct del worker_state
189   struct worker_state{
190     // id del worker
191     unsigned short int id;
192     /*
193      El worker recibirá como argumento el apuntados del manager
194      para realizar la comunicación
195     */
196     actor manager;
197     // dimensión de la matriz
198     unsigned short int dim_mat;
199     // Este contador nos dirá cuando ya podemos realizar el proceso de cálculo del coefiente
200     unsigned short int contador_break;
201     // Este contador nos acumulará cuántos Coeficientes de Clustering hemos enviado
202     unsigned short int coef_enviados;
203     // Este vectorde enteros contendrá las filas que calculará cada worker
204     vector<unsigned short int> filas;
205     // Este vector de enteros contendrá el arreglo en 1D de la matriz A2 parcial del worker
206     vector<unsigned short int> A2_1D_Par;
207     // Este vector de enteros contendrá el arreglo en 2D de la matriz A2 parcial del worker
208     vector<vector<unsigned short int>> A2_2D_Par;
209
```

```
210     void aumentaContador() {
211       contador_break+=1;
212     }
213
214     void aumentaCoefEnviados() {
215       coef_enviados+=1;
216     }
217     // Método para almacenar el producto punto de la multiplicación de vectores en un vector parcial
218     void insertaValor(unsigned short int valor,unsigned short int fila ,unsigned short int columna,unsigned
            short int n) {
219       A2_1D_Par[(fila*n)+columna]=valor;
220     }
221
222     // Método para transformar un vector 1D a uno 2D
223     vector<vector<unsigned short int>> convertir2D(vector<unsigned short int> d1_vector,unsigned short int
            rows,unsigned short int cols){
224
225       vector<vector<unsigned short int>> d2_vector(rows, vector<unsigned short int>(cols));
226
227       for (int i = 0; i < rows; i++) {
228         for (int j = 0; j < cols; j++) {
229           d2_vector[i][j]=d1_vector[(i*cols) + j];
230         }
231       }
232       return d2_vector;
233     }
234
235   };// worker_state
236
237   manager_actor::behavior_type type_checked_manager (manager_actor::stateful_pointer<manager_state>
          self,unsigned short int no_actores,unsigned short int dim_mat) {
238
239     self->state.no_actores=no_actores;
240     self->state.dim_mat=dim_mat;
241     self->state.mat_adj=self->state.read_matrix(dim_mat);
242
243     return {
244       // Implementamos el comportamiento del manager ante el mensaje get_id_atom
245       [=]( get_id_atom ) {
246         /*
247         Usamos el método calculaFilas para generar el vector con las filas que calculará el worker.
248         El manager responde a este mensaje del worker con el mensaje filas_asignadas.
249        */
250         vector<unsigned short int>
                v=self->state.calculaFilas(self->state.dim_mat,self->state.no_actores,self ->state.id);
251         self->state.aumentaID();
252
253         return filas_asignadas {v};
254       },
255       /* Implementamos el comportamiento del manager ante el mensaje envia_indice_fila_atom.
```

```
256           Este mensaje incorpora también el índice de la fila a solicitar así como el índice de la
257           columna a solicitar
258         */
259         [=]( envia_indice_fila_atom ,unsigned short int indexFila,unsigned short int indexColumna ) {
260
261           // Obtenemos el vector fila del índice que necesita el worker
262           std::vector<unsigned short int> fila= self->state.mat_adj[indexFila];
263           // Obtenemos el vector columna del índice que necesita el worker
264           std::vector<unsigned short int> columna= self->state.mat_adj[indexColumna];
265
266           return receive_worker_row_col{ fila , columna, indexFila,indexColumna};
267         },
268         /* Implementamos el comportamiento del manager ante el mensaje calcula_coef_atom.
269           Este mensaje incorpora también el índice de la fila a solicitar .
270         */
271         [=]( calcula_coef_atom,unsigned short int indexFila) {
272
273           // Obtenemos el vector fila del índice que necesita
274           std::vector<unsigned short int> fila= self->state.mat_adj[indexFila];
275
276           return calcula_coef_data{indexFila, fila };
277         },
278         /* Implementamos el comportamiento del manager ante el mensaje recibe_coef_atom, para el cual
279           el manager recibe el valor calculado del coeficiente de clustering de un nodo por parte del
280           worker
281         */
282         [=](recibe_coef_atom, unsigned short int indexNodo, float coef_clustering ) {
283           self->state.aumentaNodosRecibidos();
284           aout(self)<< "Nodo: "<< indexNodo << ". Coeficientes de Clustering: " <<coef_clustering <<". Nodos recibidos: "<< self->state.nodos_recibidos<< "\n";
285
286           // En caso que se hayan recibido los coeficientes de clustering de todos los nodos, el manager termina su ejecución
287           if ( self->state.nodos_recibidos==self->state.dim_mat) {
288             aout(self)<< "He recibido todos los Coeficientes. Terminamos el programa" <<"\n";
289             self->quit();
290           }
291         },
292       };
293     }// type_checked_manager
294
295     worker_actor::behavior_type type_checked_worker (worker_actor::stateful_pointer<worker_state>
            self,unsigned short int dim_mat){
296       self->state.dim_mat=dim_mat;
297
298      return {
299         [=](set_manager_atom, actor manager){
300           self->state.manager=manager;
301           self->send(self,inicia_atom_v);
302         },
```

```
303         /*
304         El worker comienza solicitud al manager con el mensaje inicia_atom, el cual regresa un vector de
                enteros con
305         las filas que le corresponden a cada worker
306         */
307         [=](inicia_atom){
308           self->request(self->state.manager,500s,get_id_atom_v).await(
309             [=]( filas_asignadas v){
310
311               self->state.filas=v.a;
312               std::cout << "Soy un worker y me asignaron las filas: "<<
                      self->state.filas.front()<<"-"<<self->state.filas.back()<<'\n';
313
314               // Hacemos un resize de nuestro vector A2_1D_Par
315               self->state.A2_1D_Par.resize((self->state.filas.size())*self->state.dim_mat);
316               /*
317               Por cada elemento en el vector de filas , se solicitar á al manager las columnas para calcular el
                      producto punto
318               de este por cada vector columna de la matriz de adyacencia
319               */
320               for (unsigned short int x : self->state.filas){
321                 for (unsigned short int j = 0; j < self->state.dim_mat; j++) {
322                   /*
323                   El worker solicita al manager el mensaje envia_indice_fila_atom, el cual regresa el vector
                          fila y columna
324                   de los indices enviados por el worker. Con dichos vectores, el worker realiza el producto
                          punto y almancena
325                   el resultado en el arreglo A2_1D_Par
326                   */
327                   self->request(self->state.manager,500s, envia_indice_fila_atom_v, x,j).await(
328                     [=](receive_worker_row_col mensaje){
329
330                       // Obtenemos el producto punto de dos vectores y lo guardamos en A2_1D_Par
331                       unsigned short int
                              valor=inner_product(mensaje.fila.begin(),mensaje.fila.end(),mensaje.columna.begin(),0);
332
333                       self->state.insertaValor(valor, (x-(self->state.filas[0])), j, self->state.dim_mat);
334                       aout(self)<<self->state.contador_break<<"\n";
335                       // Incrementamos el valor de contador_break en una unidad.
336                       self->state.aumentaContador();
337
338                       // Cuando el worker ha realizado todos los productos punto, puede comenzar a realizar el
                              cálculo del coefiente
339                       if (self->state.contador_break==(int)(self->state.filas.size()*self->state.dim_mat)) {
340                         // Convertimos A2_1D_Par en un arreglo 2D
341                         self->state.A2_2D_Par=self->state.convertir2D(self->state.A2_1D_Par,
                                (int)self->state.filas.size(),(int)self->state.dim_mat);
342                         /*
343                         Por cada una de las filas de este arreglo, el worker solicitar á la fila
                                correspondiente al manager
```

```
344                     para calcular A3ii y la cantidad de conexiones del nodo. Con esto, puede calcular el
                            coeficiente
345                     de clustering del nodo. El resultado lo envía al manager mediante el mensaje
                            recibe_coef_atom
346                     */
347                     for(unsigned short int i: self->state.filas){
348                       /*
349                       Solicitamos al manager el mensaje calcula_coef_atom, el cual regresa la fila de la
                              matriz de adyacencia
350                       del índice que le enviamos.
351                       */
352                       self->request(self->state.manager,500s,calcula_coef_atom_v,i).await(
353                         [=](calcula_coef_data mensaje){
354                           // Obtenemos el producto punto para calcular A3ii
355                           unsigned short int
                                  valor=inner_product(mensaje.fila.begin(),mensaje.fila.end(), self->state.A2_2D_Par[mensaje.index-
356                           // Obtenemos la cantidad de conexiones del nodo
357                           unsigned short int sum = boost::accumulate(mensaje.fila, 0);
358                           // Calculamos el coeficiente de clustering
359                           float coef_clustering = (float)valor/ (float)(sum*(sum-1));
360                           // Aumentamos el contador de coeficientes calculados
361                           self->state.aumentaCoefEnviados();
362                           // Enviamos el resultado al manager mediante el mensaje recibe_coef_atom
363                           anon_send(self->state.manager,recibe_coef_atom_v,mensaje.index,coef_clustering);
364
365                           // Cuando el worker ha enviado todos sus coeficientes, termina su ejecución
366                           if (self->state.coef_enviados==(int)self->state.filas.size()) {
367                             self->send(self, termina_atom_v);
368                           }
369                         }
370
371                       );
372                     }
373                   }
374
375                 }
376               );
377             }
378           }
379         }
380       );
381     },
382     [=]( termina_atom ) {
383       aout(self)<<"El worker terminó el cálculo de sus coeficientes"<< "\n";
384       self->quit();
385     },
386   };
387 }// type_checked_worker
388
389 /*
```

```
390     La clase config permite recibir la configuración de la aplicación con argumentos
391     de la linea de comandos
392    */
393    class config : public actor_system_config {
394    public:
395      // Definimos la configuración por default
396      uint16_t port = 0;
397      string host = "10.10.10.10";
398      bool server_mode = false;
399      unsigned short int workers=0;
400      unsigned short int dim_mat=0;
401
402      // Definimos las banderas de la configuración
403      config() {
404        opt_group{custom_options_, "global"}
405        .add(host, "host,H", "define el nodo")
406        .add(server_mode, "server,s", "correr en modo server")
407        .add(port, "port,p", "define el puerto")
408        .add(workers, "workers,w", "define la cantidad de workers")
409        .add(dim_mat, "dim,d", "define la dimensión de la matriz");
410      }
411    };
412  }
413
414  void caf_main(actor_system& system,const config& cfg) {
415
416    /*
417     Creamos una instancia de la clase abstracta actor la cual posteriormente
418     recibirá la instancia del actor manager
419    */
420    actor m;
421
422    // Para el caso en que estemos en modo server
423    if (cfg.server_mode) {
424      // Abrimos el puerto definido en la configuración
425      auto res = system.middleman().open(cfg.port);
426      if (!res) {
427        cerr << "*** No se puede abrir el puerto: " << to_string(res.error()) << endl;
428        return;
429      }
430
431      std::cout << "Estas en modo server" << '\n';
432
433      // Generamos al actor manager tal como lo hicimos en la version local
434      auto manager=system.spawn(type_checked_manager,cfg.workers, cfg.dim_mat);
435
436      // Publicamos al actor en el puerto definido
437      auto expected_port = system.middleman().publish(actor_cast<actor>(manager), 4242);
438
439      if (!expected_port) {
```

```
440       std::cerr << "*** falló la publicación: " << to_string(expected_port.error())
441                 << endl;
442       return;
443     }
444     cout << "*** El server se publicó de forma exitosa en el puerto " << *expected_port << endl;
445   }
446   else{
447     std::cout << "Estás en modo cliente" << '\n';
448
449     // Abrimos la conexión con el server en la dirección y puerto indicado
450     auto r = system.middleman().remote_actor("127.0.0.1", 4242);
451     if (!r)
452       cerr << "Incapaz de conectar el nodo: " << to_string(r.error()) << '\n';
453     else{
454       std::cout << "Nodo conectado" << '\n';
455
456       // Generamos un vector de workers
457       std::vector<worker_actor> vector_actores;
458       /*
459       Agregamos los actores workers al vector. Se presenta un cambio con respecto a la versión
460       local. La creación del worker no necesita un manager. Posteriormente agregamos la referencia
461       a este para poder iniciar la comunicación.
462       */
463       for (unsigned short int i = 0; i < cfg.workers; i++) {
464         vector_actores.insert(vector_actores.end(),system.spawn(type_checked_worker, cfg.dim_mat));
465       }
466       // Iniciamos la ejecución de los workers
467       for(worker_actor worker: vector_actores){
468         anon_send(worker, set_manager_atom_v, *r);
469       }
470     }
471   }
472 }
473
474 /*
475 Agregamos la información global de los tipos de mensajes personalizados así como el módulo
476 I/O middleman
477 */
478 CAF_MAIN(id_block::remote_types,io::middleman)
```

A.3. Implementación local con Akka

Worker.scala

```
1 import org.saddle._
2 import scala.collection.mutable.ArrayBuffer
```

```
3  import akka.actor.ActorRef
4  import Manager._
5
6  object Worker {
7
8    // Definimos los mensajes del worker
9    case class StartFilas( filas : List[Int])
10   case class RecibirFila(
11         fila : Vec[Int],
12         indexFila: Int,
13         columna: Vec[Int],
14         indexColumna: Int
15   )
16   case class CalculaCoef(columna: Vec[Int], indexColumna: Int)
17 }
18
19 import akka.actor.{Actor, ActorRef, PoisonPill}
20 import Manager._
21 import Worker._
22 import org.saddle._
23
24 class Worker(manager: ActorRef, val n: Integer) extends Actor {
25
26   // Este arreglo de enteros contendrá las filas que calculará cada worker
27   var miArregloFilas: Array[Int] = _
28   // Este arreglo de enteros contendrá el arreglo en 1D de la matriz A2 parcial del worker
29   val A2_1D_Par = ArrayBuffer[Int]()
30   // Este contador se utiliza para conocer en qué momento el worker comienza a realizar el cálculo de
31   // A3ii y de coeficiente de clustering
32   var contadorWorker: Int = 0
33
34   override def receive = {
35
36     // Implementamos el comportamiento del worker ante el mensaje "Inicia"
37     case "Inicia" => {
38       // EL worker envía al manager el mensaje "Listo para la ejecución"
39       manager ! "Listo para la ejecución"
40     }
41
42     // Implementamos el comportamiento del worker ante el mensaje StartFilas
43     case StartFilas( filas : List[Int]) => {
44       // En este mensaje, el worker recibe la fila de inicio y final para el
45       // cálculo del coeficiente de clustering
46
47       miArregloFilas = filas(0) to filas(1) toArray
48
49       // Por cada una de estas filas , el worker envía al manager el mensaje
50       // EnviarIndiceFila para recibir cada una de las columnas de la matriz de adyancencia
51
52       for ( fila <- miArregloFilas) {
```

```
53          manager ! EnviarIndiceFila(fila, filas(1), filas(0))
54        }
55
56      }
57
58      // Implementamos el comportamiento del worker ante el mensaje RecibirFila
59      case RecibirFila(
60          fila: Vec[Int],
61          indexFila: Int,
62          columna: Vec[Int],
63          indexColumna: Int
64          ) => {
65
66        // Obtenemos el producto punto de dos vectores y lo guardamos en A2_1D_Par
67        val valor = fila dot columna
68
69        A2_1D_Par += valor
70
71        contadorWorker = contadorWorker + 1
72
73        // Cuando el worker ha realizado todos los productos punto, puede comenzar a realizar el cálculo de
              A3ii y del coefiente
74        if (contadorWorker == (miArregloFilas.size * n)) {
75          for (fila <- miArregloFilas) {
76            manager ! EnviaColumna(fila)
77          }
78        }
79
80      }
81
82      // Implementamos el comportamiento del worker ante el mensaje CalculaCoef
83
84      // En este mensaje el worker recibe una fila del manager
85      // para calcular A3ii y la cantidad de conexiones del nodo. Con esto, puede calcular el coeficiente
86      // de clustering del nodo. El resultado lo envía al manager mediante el mensaje RecibeCoefClustering
87
88      case CalculaCoef(columna: Vec[Int], index: Int) => {
89        var filaIndex = miArregloFilas.indexOf(index)
90        var A2_Par = Mat(miArregloFilas.size, n, A2_1D_Par.toArray)
91
92        // Obtenemos el producto punto para calcular A3ii
93        val valor = A2_Par.row(filaIndex) dot columna
94
95        // Obtenemos la cantidad de conexiones del nodo
96        val suma_conexiones = columna.sum
97
98        // Calculamos el coeficiente de clustering
99        val coef_clustering
100           : Double = valor.toDouble / (suma_conexiones * (suma_conexiones - 1)).toDouble
101
```

```
102         // Enviamos el resultado al manager mediante el mensaje RecibeCoefClustering
103         manager ! RecibeCoefClustering(index, coef_clustering)
104       }
105
106     }
107
108 }
```

Manager.scala

```
1  import org.saddle._
2  import java.io.{FileReader, BufferedReader}
3
4  object Manager {
5
6    // Definimos los mensajes del manager
7    case class EnviarIndiceFila(index: Int, limiteSup: Int, limiteInf: Int)
8    case class EnviaColumna(index: Int)
9    case class RecibeCoefClustering(index: Int, coef_clustering: Double)
10
11   // Método que calcula qué filas le tocan a cada worker de acuerdo al id asignado
12   def calculaFilas(n: Int, p: Int, id: Int): List[Int] = {
13     var Nrow: Int = n / p
14
15     var filaInicio = id * Nrow
16     var filaFinal = 0
17
18     if (id < (p - 1)) {
19       filaFinal = filaFinal + ((id + 1) * Nrow - 1)
20     } else {
21       filaFinal = filaFinal + (n - 1)
22     }
23
24     val misFilas = List( filaInicio , filaFinal )
25
26     return misFilas
27   }
28
29   // Metodo que lee un archivo de texto dónde se encuentra la matriz
30   // de adyacencia y la convierte a una matriz
31   def obtenMatriz(n: Int): Mat[Int] = {
32     // El manager leerá la matriz
33     val file = new FileReader(
34       "/home/milo/Documentos/CAR/2doSemestre/Seminario/scripts/tesina/scala/crearModelo/files/AdjMatrix_big_scala.txt"
35     )
36
37     val reader = new BufferedReader(file)
38
39     val mat1DString = new StringBuilder();
```

```
40
41      try {
42        var line: String = null
43        while ({ line = reader.readLine(); line } != null) {
44          mat1DString += ','
45          mat1DString ++= line
46        }
47
48      } finally {
49        reader.close()
50      }
51
52      val lamatrixString = mat1DString.toString().split(",").filter(_ != "")
53
54      val lamatrixInt = lamatrixString.map(x => x.toInt)
55
56      val m = Mat(n, n, lamatrixInt)
57
58      return m
59    }
60  }
61
62  import akka.actor._
63
64  import Manager._
65  import Worker._
66
67  class Manager(n_workers: Integer, n_nodos: Integer) extends Actor {
68
69    // Esta variable dará el id corresponiente a cada solicitud de los workers
70    var id: Integer = 0
71    // Esta variable servirá como contador para saber en qué momento el manager termina la ejecución de
          programa
72    var contador: Integer = 0
73
74    // El manager tendrá un arreglo de enteros en el que irá almacenando los resultados
75    // enviados por los workers (en este caso, número de conexiones por nodo)
76    var conex_por_nodo: Array[Int] = new Array[Int](n_nodos)
77
78    // Obtenemos la matriz del archivo de texto
79    val A = Manager.obtenMatriz(n_nodos)
80
81    override def receive = {
82
83      // Implementamos el comportamiento del manager ante el mensaje "Listo para la ejecución"
84      case "Listo para la ejecución" => {
85
86        val milista = Manager.calculaFilas(n_nodos, n_workers, id)
87        // El manager responde al worker con una lista de dos elementos, uno corresponiente a la fila
88        // de inicio y a la fila final que debe calcular el worker
```

```
89          sender ! StartFilas(milista)
90
91          id = id + 1
92        }
93
94        // Implementamos el comportamiento del manager ante el mensaje EnviarIndiceFila
95        case EnviarIndiceFila(index: Int, limiteSup: Int, limiteInf: Int) => {
96          val fila = A.row(index)
97
98          // Por cada columna de la matriz de adyacencia, el manager las envía al worker
99          // mediante el mensaje RecibirFila
100
101         for (i <- 0 to (n_nodos - 1)) {
102           val columna = A.col(i)
103
104           sender ! RecibirFila(fila, index, columna, i)
105
106         }
107
108       }
109
110       // Implementamos el comportamiento del manager ante el mensaje EnviaColumna
111       case EnviaColumna(index: Int) => {
112         // El manager envía la columna específica al índice enviado por el worker.
113         val columna = A.col(index)
114         // Para hacer el envío al worker, el manager lo hace con el mensaje CalculaCoef
115         sender ! CalculaCoef(columna, index)
116       }
117
118       // Implementamos el comportamiento del manager ante el mensaje RecibeCoefClustering
119       // Con este mensaje, el manager recibe el coeficiente de clustering que calculó el worker
120       case RecibeCoefClustering(index: Int, coef_clustering: Double) => {
121
122         println(
123           s"Para el nodo ${index} su coeficiente de clustering es ${coef_clustering}"
124         )
125
126         contador = contador + 1
127
128         // Cuando el manager ha recibido todos los coeficientes de clustering, termina en programa
129         if (contador == n_nodos) {
130           println("Todos los workers calcularon sus coeficientes.")
131           context.system.terminate
132         }
133
134       }
135
136     }
137
138 }
```

Main.scala

```
/*
Versión local para el cálculo del coeficiente de clustering de una red dirigida con el patrón arquitectónico
Manager−Workers y el modelo de actor.

Ejecutar como sigue:

sbt "run workers dim"

Donde workers es la cantidad de workers y dim es la cantidad de nodos

 */
import akka.actor.{ActorRef, ActorSystem, Props, PoisonPill}
import Worker._
import Manager._
import org.saddle._

import java.io.{FileReader, BufferedReader}

object Main extends App {

  // Inicializamos n_workers y n_nodos con los valores recibidos del usuario
  val n_workers = args(0).toInt
  val n_nodos = args(1).toInt

  // Definimos el sistema de actor
  val system: ActorSystem = ActorSystem("ManagerSystem")

  // Creamos al actor manager
  val managerProps: Props = Props(classOf[Manager], n_workers, n_nodos)
  val manager: ActorRef = system.actorOf(managerProps, "manager")

  // Generamos tantos workers como el valor recibido en n_workers. Inmediatamete que son
  // creados los workers, comienzan a solicitar sus filas al manager

  val workerGenerator = for (i <− 0 until n_workers) yield {
    system
      .actorOf(Props(classOf[Worker], manager, n_nodos), name = "Worker_" + i)
  }

  // Por cada worker generado, se envía el mensaje "Inicia"
  for (worker <− workerGenerator) {
    worker ! "Inicia"
  }

}
```

A.4. Implementación distribuida con Akka

Worker.scala

```
1  import breeze.linalg._
2  import breeze.numerics._
3  import scala.collection.mutable.ArrayBuffer
4  import akka.actor.ActorRef
5  import Manager._
6
7  object Worker {
8    // Definimos los mensajes del worker. Al contrario de la versión local,
9    // en la versión distribuida los mensajes deben ser serialzados.
10   // Usamos el serializador jackson-json, el cual se incorpora a los mensajes
11   // al extender del trait Serializador
12
13   case class StartFilas(filas: List[Int]) extends Serializador
14   case class RecibirFila(fila: Array[Int]) extends Serializador
15   case class RecibirFilaColumna(
16         fila: Array[Int],
17         indexFila: Int,
18         columna: Array[Int],
19         indexColumna: Int
20   ) extends Serializador
21   case class CalculaCoef(columna: Array[Int], indexColumna: Int)
22         extends Serializador
23 }
24
25 import breeze.linalg._
26 import breeze.numerics._
27 import akka.actor.{Actor, ActorRef}
28
29 import scala.concurrent.Await
30 import akka.pattern.ask
31 import akka.util.Timeout
32 import scala.concurrent.duration._
33
34 import Manager._
35 import Worker._
36
37 class Worker(manager: ActorRef, val n: Integer) extends Actor {
38   // Este arreglo de enteros contendrá las filas que calculará cada worker
39   var miArregloFilas: Array[Int] = _
40
41   // Este arreglo de enteros contendrá el arreglo en 1D de la matriz A2 parcial del worker
42   val A2_1D_Par = ArrayBuffer[Int]()
43
44   // Este contador se utiliza para conocer en qué momento el worker comienza a realizar el cálculo de
45   // A3ii y de coeficiente de clustering
```

```
46  var contadorWorker: Int = 0
47
48  var coeficientesEnviados: Int = 0
49
50  override def receive = {
51    // Implementamos el comportamiento del manager ante el mensaje "Inicia"
52    case "Inicia" => {
53      // Para la versión distribuida, utilizamos Futuros. En Akka, hay dos formas
54      // de recibir una respuesta de un Actor. La primera es al enviar un mensaje
55      // del tipo actor ! msg, que funciona únicamente si el emisor original fue
56      // un actor. La segunda es mediante Futuros. Usando actor ? msg para enviar un
57      // mensaje nos regresará un futuro. De esta forma, enviaremos el mensaje y
58      // esperaremos el resultado real más tarde. Esto causará que el hilo actual se
59      // bloqué y espere al actor hasta que haya completado el Futuro con la respuesta.
60      // Las operaciones bloqueantes son Await.result y Await.ready, lo que facilita
61      // identificar en qué parte se realizó el bloqueo.
62
63      // Definimos el tiempo de espera del futuro
64      implicit val timeout = Timeout(10 seconds)
65      // Enviamos el mensaje al manager y obtendremos un futuro.
66      val future = manager ? "GetId"
67      // El resultado del futuro estará en la variable result
68      val result =
69        Await.result(future, timeout.duration).asInstanceOf[RecibirFila]
70
71      // Cuando está listo el resultado, obtenemos la fila que ha enviado el manager
72      miArregloFilas = result.fila
73
74      // Por cada elemento en el vector de filas, se solicitará al manager las
75      // columnas para calcular el producto punto de este por cada vector columna
76      // de la matriz de adyacencia
77      for (i <- 0 until result.fila.length) {
78        for (j <- 0 to (n - 1)) {
79          var fila = result.fila(i)
80          println(s"Para la fila ${fila} solicito la columna ${j}")
81
82          //El worker solicita al manager el mensaje EnviarIndiceFila, el cual regresa el vector fila y
                columna
83          //de los indices enviados por el worker. Con dichos vectores, el worker realiza el producto punto
                y almancena
84          //el resultado en el arreglo A2_1D_Par
85
86          val future_indice_fila_columna = manager ? EnviarIndiceFila(fila, j)
87          val result_indice_fila_columna =
88            Await
89              .result(
90                future_indice_fila_columna,
91                timeout.duration
92              )
93              .asInstanceOf[RecibirFilaColumna]
```

```
94          // Convertimos en vector el arreglo de enteros recibido
95          var fila_vector = DenseVector(result_indice_fila_columna. fila )
96          // Convertimos en vector el arreglo de enteros recibido
97          var columna_vector = DenseVector(result_indice_fila_columna.columna)
98
99          val suma_conexiones = sum(fila_vector)
100
101         // Obtenemos el producto punto de dos vectores y lo guardamos en A2_1D_Par
102         val valor = fila_vector dot columna_vector
103         A2_1D_Par += valor
104
105         contadorWorker = contadorWorker + 1
106
107         // Cuando el worker ha realizado todos los productos punto, puede comenzar a realizar el cálculo
                del coefiente
108         if (contadorWorker == (miArregloFilas.size * n)) {
109
110           //Por cada una de las filas de este arreglo, el worker solicitará la fila correspondiente al
                manager
111           //para calcular A3ii y la cantidad de conexiones del nodo. Con esto, puede calcular el
                coeficiente
112           //de clustering del nodo. El resultado lo envía al manager mediante el mensaje
                RecibeCoefClustering
113
114           for ( fila <- miArregloFilas) {
115
116             // Enviamos al manager el mensaje EnviaColumna para recibir un futuro
117             val future_envia_columnas = manager ? EnviaColumna(fila)
118             val result_envia_columnas =
119               Await
120                 . result (future_envia_columnas, timeout.duration)
121                 .asInstanceOf[CalculaCoef]
122
123             // Con el resultado del futuro, obtenemos el índice de la columna que está enviando
124             // el manager.
125             var filaIndex =
126               miArregloFilas.indexOf(result_envia_columnas.indexColumna)
127
128             // Convertimos el arreglo A2_1D_Par en una matriz con dimensión n_filas X n
129             var A2_Par =
130               new DenseMatrix(miArregloFilas.size, n, A2_1D_Par.toArray)
131             // Convertimos en vector el arreglo de enteros que está enviando el manager
132             var columna_vector = DenseVector(result_envia_columnas.columna)
133
134             // Obtenemos el producto punto para calcular A3ii
135             val valor = A2_Par(filaIndex, ::) .t dot columna_vector
136             // Obtenemos la cantidad de conexiones del nodo
137             val suma_conexiones = sum(columna_vector)
138             // Calculamos el coeficiente de clustering
139             val coef_clustering
```

```
140                     : Double = valor.toDouble / (suma_conexiones * (suma_conexiones - 1)).toDouble
141                 // Enviamos el resultado al manager mediante el mensaje RecibeCoefClustering
142                 manager ! RecibeCoefClustering(
143                   result_envia_columnas.indexColumna,
144                   coef_clustering
145                 )
146
147                 coeficientesEnviados += 1
148                 // Cuando el worker ha enviado todos sus coeficientes, termina su ejecución
149                 if (coeficientesEnviados == miArregloFilas.size) {
150                   println("El worker ha terminado de calcular sus coeficientes .")
151                   context.system.terminate
152
153                 }
154
155               }
156             }
157
158           }
159         }
160       }
161
162     }
163
164 }
```

WorkerMain.scala

```
1  /*
2
3   Versión distribuida para el cálculo del coeficiente de clustering de una red dirigida con el patrón
        arquitectónico
4   Manager-Workers y el modelo de actor.
5
6   Para ejecutar el main del worker, quien será el cliente, ejecutar como sigue:
7
8   sbt "-Dakka.remote.artery.canonical.port=2552" "runMain WorkerMain workers dim"
9
10  Donde workers es la cantidad de workers y dim es la cantidad de nodos
11
12  */
13
14 import akka.actor.{ActorRef, ActorSystem, Props}
15 import akka.util.Timeout
16
17 import scala.concurrent.ExecutionContext.Implicits.global
18 import scala.concurrent.duration.SECONDS
19 import scala.util.{Failure, Success}
20
```

```
21 object WorkerMain extends App {
22
23   // Inicializamos n_workers y n_nodos con los valores recibidos del usuario
24   val n_workers = args(0).toInt
25   val n_nodos = args(1).toInt
26
27   // Definimos el sistema de actor para el Manager
28   val system: ActorSystem = ActorSystem("WorkerSystem")
29
30   // Abrimos la conexión con el server en la dirección y puerto indicado
31   val path =
32     "akka://ManagerSystem@127.0.0.1:2553/user/manager"
33
34   // Definimos el tiempo de espera del futuro
35   implicit val timeout: Timeout = Timeout(5, SECONDS)
36
37   // En caso que se encuentre al actor manager en la dirección dada,
38   // generamos n_workers workers.
39
40   system.actorSelection(path).resolveOne().onComplete {
41     case Success(manager) =>
42       val workerGenerator = for (i <- 0 until n_workers) yield {
43         system.actorOf(
44           Props(classOf[Worker], manager, n_nodos),
45           name = "Worker_" + i
46         )
47       }
48       // Por cada worker generado, se envía el mensaje "Inicia"
49       for (worker <- workerGenerator) {
50         worker ! "Inicia"
51       }
52
53     case Failure(e) => println(e)
54   }
55 }
```

Manager.scala

```
1  import java.io.{FileReader, BufferedReader}
2  import breeze.linalg._
3  import breeze.numerics._
4
5  object Manager {
6    // Definimos los mensajes del manager. Al contrario de la versión local,
7    // en la versión distribuida los mensajes deben ser serialzados.
8    // Usamos el serializador jackson-json, el cual se incorpora a los mensajes
9    // al extender del trait Serializador
10   case class EnviarIndiceFila(fila: Int, columna: Int) extends Serializador
11   case class EnviaColumna(index: Int) extends Serializador
```

```
12  case class RecibeCoefClustering(index: Int, coef_clustering : Double)
13      extends Serializador
14
15  // Método que calcula qué filas le tocan a cada worker de acuerdo al id asignado
16  def calculaFilas(n: Int, p: Int, id: Int): List[Int] = {
17    var Nrow: Int = n / p
18
19    var filaInicio = id * Nrow
20    var filaFinal = 0
21
22    if (id < (p - 1)) {
23      filaFinal = filaFinal + ((id + 1) * Nrow - 1)
24    } else {
25      filaFinal = filaFinal + (n - 1)
26    }
27
28    val misFilas = List( filaInicio , filaFinal )
29
30    return misFilas
31  }
32
33  // Metodo que lee un archivo de texto dónde se encuentra la matriz
34 // de adyacencia y la convierte a una matriz
35
36  def obtenMatriz(n: Int): DenseMatrix[Int] = {
37    // El manager leerá la matriz
38
39    val file = new FileReader(
40      "/home/milo/Documentos/CAR/2doSemestre/Seminario/github/akka-distribuido-coef-clustering/data/AdjMatrix_big_scala.txt
41    )
42
43    val reader = new BufferedReader(file)
44
45    ////////////////////////////////////////////////////////////////
46    val mat1DString = new StringBuilder();
47
48    try {
49      var line: String = null
50      while ({ line = reader.readLine(); line } != null) {
51        mat1DString += ','
52        mat1DString ++= line
53      }
54
55    } finally {
56      reader.close()
57    }
58
59    val lamatrixString = mat1DString.toString().split(",").filter(_ != "")
60
61    val lamatrixInt = lamatrixString.map(x => x.toInt)
```

```
62
63      val m = new DenseMatrix(n, n, lamatrixInt)
64
65      return m
66    }
67
68  }
69
70  import java.util.{Currency, Locale}
71
72  import akka.actor.Actor
73  import akka.dispatch.RequiresMessageQueue
74  import akka.dispatch.BoundedMessageQueueSemantics
75
76  import Manager._
77  import Worker._
78
79  class Manager(n_workers: Integer, n_nodos: Integer) extends Actor {
80    // Esta variable dará el id corresponiente a cada solicitud de los workers
81    var id: Integer = 0
82  // Esta variable servirá como contador para saber en qué momento el manager termina la ejecución de
        programa
83    var contador: Integer = 0
84
85  // El managaer tendrá un arreglo de enteros en el que irá almacenando los resultados
86  // enviados por los workers (en este caso, número de conexiones por nodo)
87    var conex_por_nodo: Array[Int] = new Array[Int](n_nodos)
88
89    // Obtenemos la matriz del archivo de texto
90    val A = Manager.obtenMatriz(n_nodos)
91
92    override def receive = {
93
94      // Implementamos el comportamiento del manager ante el mensaje "GetId"
95      case "GetId" => {
96        val milista = Manager.calculaFilas(n_nodos, n_workers, id)
97        var miArregloFilas = milista(0) to milista(1) toArray
98
99        // El manager envía al worker el arreglo de filas que debe calcular mediante
100       // el mensaje RecibirFila
101       sender ! RecibirFila(miArregloFilas)
102       id = id + 1
103
104     }
105
106     // Implementamos el comportamiento del manager ante el mensaje EnviarIndiceFila
107     case EnviarIndiceFila( fila: Int, columna: Int) => {
108       val fila_mat = A(::, fila)
109
110       val columna_mat = A(::, columna)
```

```
111
112         // Por cada solicitud del worker, el manager envía la fila y columna de acuerdo a
113         // los indices enviados por el worker.
114         sender ! RecibirFilaColumna(
115           fila_mat.toArray,
116           fila,
117           columna_mat.toArray,
118           columna
119         )
120       }
121
122       // Implementamos el comportamiento del manager ante el mensaje EnviaColumna
123       case EnviaColumna(index: Int) => {
124         // El manager envía la columna específica al índice enviado por el worker.
125         val columna = A(::, index)
126         // Para hacer el envío al worker, el manager lo hace con el mensaje CalculaCoef
127         sender ! CalculaCoef(columna.toArray, index)
128       }
129
130       // Implementamos el comportamiento del manager ante el mensaje RecibeCoefClustering
131       // Con este mensaje, el manager recibe el coeficiente de clustering que calculó el worker
132       case RecibeCoefClustering(index: Int, coef_clustering: Double) => {
133
134         println(
135           s"Para el nodo ${index} su coeficiente de clustering es ${coef_clustering}"
136         )
137
138         contador = contador + 1
139
140         // Cuando el manager ha recibido todos los coeficientes de clustering, termina en programa
141         if (contador == n_nodos) {
142           println(
143             "Todos los workers calcularon sus coeficientes."
144           )
145           context.system.terminate
146         }
147
148       }
149
150     }
151 }
```

ManagerMain.scala

```
1 /*
2 Versión distribuida para el cálculo del coeficiente de clustering de una red dirigida con el patrón
      arquitectónico
3 Manager−Workers y el modelo de actor.
4
```

```
Para ejecutar el main del manager, quien será el servidor, ejecutar como sigue:

sbt "-Dakka.remote.artery.canonical.port=2553" "runMain ManagerMain workers dim"

Donde workers es la cantidad de workers y dim es la cantidad de nodos

 */

import akka.actor.{ActorRef, ActorSystem, Props}

object ManagerMain extends App {

  // Inicializamos n_workers y n_nodos con los valores recibidos del usuario
  val n_workers = args(0).toInt
  val n_nodos = args(1).toInt

  // Definimos el sistema de actor para el Manager
  val system: ActorSystem = ActorSystem("ManagerSystem")

  // Creamos al actor manager
  val managerProps: Props = Props(classOf[Manager], n_workers, n_nodos)
  val manager: ActorRef =
    system.actorOf(managerProps, "manager")
}
```