Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Programa de Maestría en Computación

EVALUACIÓN DE PROGRAMACIÓN CONCURRENTE BASADA EN EL MODELO DE ACTORES EN TELÉFONOS INTELIGENTES

Tesis para optar por el grado de Magister Scientiae en Computación

Estudiante:
HUGO MAURICIO MORA ARLEY

Profesor Asesor:
JOSÉ CASTRO MORA, PH. D.

Cartago, Costa Rica
Noviembre 2015

A mi madre y mi padre

Agradecimientos

Se agradece al Dr. José Castro por servir de guía durante todo el proceso ya que su ayuda fue determinante al indicar cómo manejar los diferentes pasos de la investigación y dar puntos de vista como investigador a través de todo el proceso.

Se agradece al Dr. Esteban Meneses por su aporte como investigador experto en temas de programación concurrente que enriquecieron los contenidos de la investigación y por su detallada revisión de los documentos y múltiples sugerencias durante el proceso.

Se agradece a la Máster Wendy Alvarado por su revisión de la investigación que aporta aspectos importantes de forma y contenido, y ayudó a un mejor entendimiento de la tesis e incrementó la calidad de la misma.

Por último se agradece al Dr. Roberto Cortés, como director del programa de maestría aportó pautas de orden importantes y una visión a futuro en la investigación que fue decisiva en el resultado de la misma.

Resumen

Los dispositivos móviles conocidos como “smartphones” están evolucionando rápidamente. Algunos han llegado a incorporar varios procesadores principales y memorias RAM que exceden el gigabyte, lo cual provoca que las aplicaciones que manejan desarrollen exigencia alta en el consumo de recursos. Sin embargo, estos dispositivos, pensados para ofrecer disponibilidad y movilidad de la información, fueron diseñados para ser llevados de un lugar a otro, son de tamaño pequeño y usan una batería como fuente de energía para proveer una autonomía considerable. Aunque han evolucionado, el objetivo de estos dispositivos sigue siendo el mismo: una alta disponibilidad de la información (en cualquier momento y lugar). Estos aparatos no son diseñados para altas cargas computacionales que dediquen a los procesadores a trabajos intensivos dejando de lado la alta disponibilidad de la información (un procesador ocupado no atiende solicitudes alternas de manejo de información).

Por lo anterior, al tener mayores capacidades, surge la pregunta: ¿qué posibilidades ofrecen estos dispositivos en procesos que exploten la concurrencia usando sus varios procesadores? Inclusive, debido a su popularidad y ubicuidad, se podría pensar en procesamiento distribuido entre varios dispositivos para resolver labores complejas de cómputo [29]. En la actualidad, ya se manejan algunas aplicaciones que requieren un procesamiento intensivo de información en los smartphones [11] [17] [28] [38].

La programación concurrente ofrece enfoques prometedores que ayudan en la creación de sistemas paralelos y distribuidos. El modelo de actores [7] es un concepto útil que facilita

la programación concurrente, ya que es un diseño que ayuda a evitar errores complejos de programación que se dan en el manejo de los recursos del sistema en entornos paralelos y distribuidos. Esta investigación evalúa el rendimiento de la programación concurrente basada en el modelo de actores en los dispositivos móviles llamados teléfonos inteligentes o smartphones.

Abstract

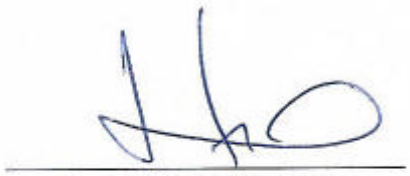
Mobile devices known as smartphones are evolving to incorporate several processors and a RAM memory greater than a gigabyte in response to the high resource demand of today's software applications. These devices were designed to be highly available information devices and are equipped with special features for that purpose; they are designed in a way that facilitates their mobility from place to place and with the use of a battery as a power source to provide considerable autonomy. Even with the recent evolution of these devices, the goal for them remains the same: high information availability (anytime, anywhere). These devices are not designed for high computational loads, the processor's intensive work blocks the high availability of information (a busy processor doesn't address alternate requests for information handling), however, the increasing power of these devices opens the scenario for using the concurrent processing features of their multiple processors for other possibilities. Also, given their current popularity and ubiquity, these devices may become useful in the distributed computing scenario, where the solution of complex computational tasks may be shared between many devices [29]. Nowadays there are yet some applications requiring information intensive processing on the smartphones [17] [28] [38].

Concurrent programming offers promising approaches to help in the creation of parallel and distributed systems. The Actors Model [7] is an interesting paradigm because it is a design that helps to minimize complex programming errors using the system resources in parallel and distributed environments. This research evaluates the performance of the concurrent programming using the Actors Model in mobile devices called smartphones.

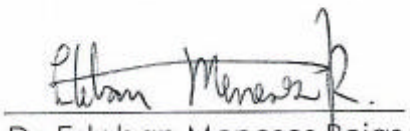
Aprobación de Tesis

Evaluación de Programación Concurrente Basada en el Modelo de Actores en Teléfonos Inteligentes

TRIBUNAL EXAMINADOR



Dr. Jose Castro Mora
Profesor Asesor



Dr. Esteban Meneses Rojas
Profesor Lector

MSi. Wendy Alvarado Arce
Profesor Externo

Dr. Roberto Cortés Morales
Coordinador del Programa
de Maestría en Computación

Noviembre, 2015

ÍNDICE GENERAL

CAPÍTULO 1. INTRODUCCIÓN

El paralelismo a nivel de hardware, en donde se cuenta con más de un procesador principal (CPU), cada día cobra más auge. Hoy en día los teléfonos inteligentes son muy conocidos por incorporar más de un procesador principal y memorias principales que igualan o exceden el gigabyte. Sin embargo, estos dispositivos siguen preservando las intenciones iniciales de los PDA de servir como un dispositivo para proveer información personal que se requiere con alta disponibilidad [45], ya sea para ser consultada o modificada.

La presente Investigación surge de la inquietud por saber la conveniencia del modelo de actores [7] en los dispositivos móviles conocidos como Teléfonos Inteligentes o smartphones (en adelante se usará cualquiera de estos términos para hacer referencia a estos dispositivos), ya que el modelo de actores es un enfoque prometedor que ayuda en la programación concurrente en sistemas paralelos y distribuidos. Más adelante en este capítulo se define con un poco más de extensión los conceptos de computación paralela y computación distribuida [43].

El modelo de actores, el cual se verá con más detalle en el siguiente capítulo, es un modelo de programación que facilita el diseño y la comunicación entre procesos en un sistema distribuido. En este los procesos, llamados actores, solo pueden comunicarse mediante mensajes, y utilizan un buzón asincrónico de mensajería para transmitirse información entre ellos. El lenguaje de programación Erlang ofrece una excelente implementación de este modelo y se utilizó para la implementación de los algoritmos de actores utilizados en esta tesis.

Los objetos distribuidos son otro paradigma que se puede utilizar para resolver problemas similares, sin embargo en esta tesis no abordamos este modelo y nos concentramos en el modelo de actores. Los objetos distribuidos han mostrado ser exitosos en contextos de programación para aplicaciones científicas y lo que se llama computación de alto rendimiento. Estas aplicaciones suelen ejecutarse en hardware dedicado, ya sea máquinas paralelas o clusters de computadoras dentro de una red de confianza. Es característica de los objetos distribuidos el migrar los objetos, esto es, mover el cómputo más sus datos de un nodo a otro. En sistemas distribuidos más abiertos, (i.e. Internet) donde la participación de los nodos en el sistema o cómputo distribuido es menos dedicada, y en algunos casos hasta efímera (el nodo no solo puede caerse, sino que puede desvincularse total y voluntariamente del cómputo).

Algunos autores, como por ejemplo en el protocolo REST[12], sugieren que no es buena idea compartir y migrar estado entre los participantes. Esto debido a la fragilidad de las comunicaciones en una red abierta y la complejidad que agrega darle seguimiento a la consistencia del estado de los objetos que se comparten, además del riesgo de ejecutar código transferido de otro nodo sobre una red pública, la cual no es de fiar (not trusted).

1.1 Computación distribuida

Por computación distribuida se entiende varias computadoras independientes conectadas entre sí que necesitan comunicarse entre ellas para llevar a cabo una tarea. Puede que en este trabajo se trate de brindar un servicio, datos compartidos o incluso para guardar

conjuntos de datos que son muy grandes para manejarlos en una sola máquina. Las computadoras en estos ambientes son independientes y no comparten memoria o procesadores. Estas computadoras se comunican por mensajes, los cuales son piezas de información transferidas de una computadora a otra usando una red.

Los mensajes pueden comunicar muchas cosas, las computadoras pueden indicarle a otras computadoras que ejecuten un proceso con ciertos argumentos en particular, pueden enviar y recibir paquetes de datos o enviar señales que le indican a otras computadoras una manera de comportarse en particular. Las computadoras en los sistemas distribuidos pueden tener diferentes roles, lo cual depende del objetivo del sistema, del hardware de la computadora y de las propiedades del software. Hay dos maneras predominantes de organizar computadoras en los sistemas distribuidos, la primera es la arquitectura cliente-servidor, la segunda es la denominada Peer-to-Peer o P2P en inglés (que en español se denomina red de pares o red entre iguales).

Una red cliente servidor es una forma de ofrecer un servicio por medio de una fuente central, en ella hay un servidor que provee el servicio y hay múltiples clientes que se comunican con el servidor para consumir sus productos. Mientras que las redes cliente-servidor son apropiadas para situaciones orientadas a servicios, en las redes P2P hay otros objetivos, en estos se desea una división más igualitaria del trabajo que tiene el efecto de complicar la comunicación pero evita los cuellos de botella y mejora la escalabilidad. Bajo este otro esquema, todas las computadoras envían y reciben datos, y todas contribuyen con algún poder de procesamiento y memoria.

En el esquema P2P cuando un sistema distribuido crece en tamaño su capacidad de recursos computacionales crece. Una característica importante en este esquema es que todas las computadoras deben tener la posibilidad de comunicarse con cualquier otra computadora en la red, ya que se busca la división del trabajo entre todas las computadoras de la red.

1.2 Computación paralela

Por computación paralela se entiende un dispositivo con varias unidades de ejecución compartiendo recursos. En 1965, Gordon Moore hizo una predicción acerca de cuánto pueden incrementar la velocidad las computadoras conforme pasa el tiempo e indicó que el número de transistores que se pueden montar en un chip se doblaría cada 2 años, a esto se le llama la ley de Moore y aún hoy en día parece cumplirse de manera precisa. Hasta hace unos diez años este incremento en transistores se veía reflejado en un correspondiente incremento en la velocidad del procesador, sin embargo, a mediados de la década del 2000-2010 el incremento en velocidad se empezó a estancar, con el agravante de que la capacidad de los discos y el volumen de datos que se maneja actualmente sigue creciendo exponencialmente. Para solucionar esto, los constructores de chips han optado por ofrecer otro recurso: procesadores con múltiples núcleos e hilos de ejecución.

Si dos o más procesadores están disponibles, entonces muchos programas pueden ser ejecutados más rápidamente. Todos estos procesadores pueden compartir los mismos datos pero el trabajo se hará de manera paralela, ya que mientras un procesador está trabajando en un aspecto de un proceso, otros procesadores pueden estar trabajando en otro aspecto.

Para que varios procesadores puedan trabajar de manera conjunta deben tener la capacidad de compartir información entre ellos, eso se lleva a cabo mediante un ambiente de memoria compartida. Las variables, objetos y estructuras de datos en este ambiente son accesibles por todos los procesos, el rol de un procesador en computación es llevar a cabo la evaluación y ejecución de reglas de un lenguaje de programación. En el modelo de

memoria compartida, diferentes procesos pueden ejecutar diferentes instrucciones, pero podría ser que las instrucciones afecten el ambiente compartido.

1.3 Diferencias entre una computadora de escritorio y un smartphone

Es conveniente aclarar que la naturaleza de las aplicaciones que corren en los smartphones son diferentes a las aplicaciones que corren las computadoras convencionales. Las aplicaciones que corren en estos dispositivos por lo general son para brindar datos de alta disponibilidad necesarios para una persona con el fin de poder organizarse mejor y desenvolverse durante el día. Este conjunto de aplicaciones caen dentro de un grupo denominado PIM por su siglas en inglés (personal information manager) y se vuelven típicas aquí las aplicaciones de calendario, agenda de direcciones, calculadora, procesador de palabras y hoja de cálculo [45].

Es claro que las aplicaciones PIM no contemplan el manejo de procesamiento intensivo ya que dejarían el procesador o procesadores centrales del dispositivo demasiado ocupados para poder brindar la información que el usuario requiera de manera espontánea con otras aplicaciones. Por ejemplo, no se podrían destinar los recursos necesarios para procesar de manera fluida una llamada entrante con la aplicación correspondiente. Tampoco se supone que estos dispositivos usen facilidades como paginación a memoria secundaria ni procesos distribuidos con otros dispositivos, por lo tanto estos dispositivos no cuentan con soporte nativo en los sistemas operativos para esas características. Las aplicaciones para smartphone no suponen conexiones de alta velocidad entre dispositivos, ya que al ser móviles suponen conexiones vía aire. No así las aplicaciones para computadoras

convencionales, que disponen de conexiones con redes vía cable a velocidades apropiadas para compartir los recursos en la red y favorecer los sistemas distribuidos en este aspecto. Otra diferencia es que las aplicaciones para smartphones no son destinadas para captura de de datos masivos, ya que una de las características primordiales del smartphone es ser pequeño, y brinda características de captura y despliegue de información limitadas a un uso ocasional.

Una de las evidencias de la diferencia entre los smartphones y las computadoras de escritorio es indicada en [44]. Aquí se comenta que librerías para el manejo de realidad aumentada se han rediseñado por las diferencias que se encuentran en el hardware que hace que la migración de librerías existentes basadas el las arquitecturas de computadoras de escritorio no ofrezcan un buen resultado (se indica en página #1 del documento).

Con estas restricciones propias de esta plataforma que ya han afectado otros desarrollos, no se sabe qué tanto se ve perjudicado o beneficiado el desempeño del modelo de actores en el smartphone básico monoprocesador o en los que se manejan en una gama más alta con múltiples procesadores. Estos últimos, aunque más poderosos, siguen preservando las restricciones antes mencionadas.

1.4 Tendencias actuales que abren nuevas posibilidades

La tendencia actual y debido al avance normal de la tecnología es que las aplicaciones demanden cada vez más recursos, incluso las que forman parte del PIM. Sensores con más capacidades (por ejemplo las cámaras, mayor velocidad de las redes inalámbricas, mayor

capacidad de despliegue, etc.) hacen que las capacidades de los dispositivos crezcan. Esto los vuelve atractivos para otros campos como lo es el procesamiento concurrente y con ello la programación concurrente [34].

La arquitectura de los smartphones, aunque se extiende día con día, sigue basándose en usar recursos baratos con capacidades limitadas que ahorren energía, ya que su fuente de poder es una batería. Estas limitantes afectan también a sus sistemas operativos, los cuales son mucho más pequeños que los usados en una computadora, sin embargo, el dispositivo debe admitir un universo totalmente heterogéneo de aplicaciones hechas en una diversidad extensa de lenguajes. Esta diversidad de lenguajes y de aplicaciones unida al multiprocesamiento que se maneja en estos dispositivos sugiere la conveniencia de herramientas que favorezcan el manejo seguro de los recursos en la programación concurrente.

Las fortalezas de la plataforma en la que se suponía que correrían los lenguajes de programación concurrente se contraponen con las debilidades reales en los smartphones, producto de ser dispositivos construidos con objetivos diferentes a los que tiene una computadora. Surge aquí la incertidumbre de saber si estos lenguajes podrían ayudar a explotar de manera aceptable la capacidad de procesamiento concurrente y distribuido que aflora en estos dispositivos. De esa incertidumbre es que parte el presente estudio para comprobar, y si se quiere sugerir, la mejor manera de usar este tipo de lenguajes en estos dispositivos. Este estudio viene a indicar qué podemos esperar y cómo debemos trabajar

para obtener los mejores resultados usando un lenguaje de programación concurrente basado en el modelo de actores.

1.4.1 Surgen aplicaciones de cálculo intensivo para el smartphone

El smartphone es una plataforma que cada vez es más utilizada para aplicaciones de cálculo intensivo [28]. Hay varias aplicaciones de cálculo intensivo específicas para dispositivos móviles como el smartphone, a continuación se citan unas cuantas.

1.4.1.1 Aplicaciones médicas

Según un estudio [38], es mejor tener los datos que sirven para graficaciones médicas en el dispositivo móvil ya que esto ayuda a ahorrar mucho tiempo, por ejemplo, en la sala de emergencias (página 2). Inclusive hay aplicaciones como la MobileMIM para IOS, por ejemplo, aprobadas por la FDA para diagnóstico con graficaciones de datos PET-MRI (tomografía por emisión de positrones, imagen de resonancia magnética),

Según el estudio, se justifica el uso de estos dispositivos ya que los profesionales pueden obtener una imagen más robusta de la condición del paciente y pueden dar una respuesta más informada acerca del tratamiento del paciente. Se indica en este estudio que las aplicaciones para diferentes técnicas de generación de gráficos son demandantes en hardware, también se indica que estas aplicaciones son más fáciles de entender por el personal que no sea radiólogo.

Otra ventaja es que dispositivos como la tablet le permiten al cirugano acceder a la información en el momento oportuno, tal como en los salones de cirugía, y evita el

inconveniente de abandonar el recinto para ver imágenes relevantes respetando las reglas de esterilidad. Hay aplicaciones de estas que hacen el proceso de generación de gráficos en 3D en el smartphone usando el CPU en procesos complejos para desplegar gráficos a partir de conjuntos de datos de tamaño considerable, por ejemplo ImageVis3D.

1.4.1.2 Aplicaciones GPS

El GPS por naturaleza es portátil y utiliza gran cantidad de datos para construir vistas de mapas de manera dinámica. Por ejemplo existe Maps 3D PRO, el cual maneja los datos de manera local para las graficaciones en 3D que muestra.

1.4.1.3 Aplicaciones de análisis de imágenes

Aquí se incluyen aplicaciones AR (realidad aumentada). En un estudio [17] que habla de las aplicaciones MAR (realidad aumentada para dispositivos móviles) específicamente se indica que se ejecutan en paralelo tareas de las aplicaciones AR aprovechando el multi-núcleo en los dispositivos (Página 15 del estudio). También indican que las aplicaciones MAR son típicamente aplicaciones de computación intensiva (página 19). En otro estudio [11] se ubican los smartphones como una de las plataformas sobre las cuales existen aplicaciones de realidad aumentada, las cuales usan diversos cálculos para procesar imágenes y agregar información visual útil al usuario en la imagen de la realidad que se contempla. Aquí se habla desde aplicaciones para entretenimiento hasta aplicaciones para diseño de interiores y medicina.

1.5 Objetivo de esta tesis

A continuación se desglosa el objetivo general y los específicos de la tesis.

1.5.1 Objetivos generales

El primer objetivo general es aprobar o rechazar la siguiente hipótesis: El modelo de actores es un modelo robusto para ser utilizado en los smartphones en programación concurrente. Para aprobar o rechazar esta hipótesis se debe hacer una serie de estudios con los que se busca cumplir con el segundo objetivo general, el cual es dar al tomador de decisiones datos importantes para que él mismo sea el que evalúe qué tan conveniente es usar el modelo de actores en la implementación de su aplicación o aplicaciones ante otra opción, como lo es la implementación de concurrencia con hilos. Este segundo objetivo constituye el aporte principal de esta investigación, el cual es validado por dos comunidades internacionales [10][40][22].

Para aprobar o rechazar esta hipótesis, el estudio responderá tres preguntas fundamentales:

1. Se indicará si se puede o no usar lenguajes concurrentes basados en el modelo de actores en los dispositivos móviles conocidos como smartphones.
2. Se indicará si este modelo es o no apropiado en los smartphones, esto desde el punto de vista de desempeño en ejecución de programas hechos bajo el modelo de actores, y se analizará la conveniencia para el desarrollador de aplicaciones bajo este modelo.
3. Se indicará las características encontradas bajo las cuales se puede tener mejores resultados y qué tan buenos son esos resultados usando el modelo de actores.

1.5.2 Objetivos específicos

Los objetivos específicos ayudan a cumplir los objetivos generales. Estos son:

- Diseñar métricas que indiquen la idoneidad de la programación concurrente basada en el modelo de actores en una plataforma nueva para este enfoque.
- Implementar el algoritmos que servirán para comprobar la eficiencia del enfoque de programación concurrente basado en el modelo de actores en los smartphones.
- Medir resultados de Erlang en los smartphones en cuanto a eficiencia en la ejecución de procesos, gasto de memoria y gasto de energía, escogiendo los parámetros que indiquen eficacia para proporcionar los mejores resultados.

CAPÍTULO 2. MARCO TEÓRICO

2.1 Programación concurrente

Debido a que hoy en día los procesadores están creciendo más en la cantidad de hilos de ejecución que soportan que en la velocidad de cada hilo independiente, se ha vuelto más conveniente mejorar el desempeño usando varios procesadores que tratar de optimizar un solo procesador para optimizarlo aún más [13]. Con esto se tiene un beneficio directo para la programación concurrente, al abrirse la posibilidad de ejecutar de manera paralela varias tareas.

La concurrencia es un concepto de sistemas operativos que se refiere a la existencia de varios hilos de ejecución independientes en una máquina. Podemos aplicar la concurrencia a la programación de un problema cuando dividimos la solución de este en soluciones a problemas más pequeños relativamente independientes, que luego se juntan para conformar la solución al problema original. Si además de esto el hardware subyacente puede ejecutar varios hilos a la vez, entonces podemos utilizar paralelismo en la solución.

Con este enfoque surgen problemas que tienen que ver con los recursos compartidos entre los procesos, por ejemplo la memoria. A continuación se describen los principales problemas [4].

2.1.1 Violación de la exclusión mutua

A veces las operaciones hechas en un programa concurrente fallan en producir los resultados esperados si estas operaciones son ejecutadas por dos o más procesos simultáneamente. Esto se debe a que existe una parte del programa donde se realizan dichos procesos que constituye una región crítica, la cual es una parte del programa en la que se debe garantizar que, si un proceso entra a la misma, ningún otro podrá entrar. Es necesario determinar esa región crítica cuidadosamente y entonces garantizar exclusión mutua en esa área.

2.1.2 Bloqueo mutuo o deadlock

Un ejemplo de bloqueo mutuo es el siguiente: un proceso A tiene un recurso R_1 bloqueado y está esperando por el recurso R_2 mientras que el proceso B tiene bloqueado el recurso R_2 y está esperando el recurso R_1. Un proceso se encuentra en estado de deadlock si está esperando por un recurso que no se le dará nunca porque hay otro proceso que lo tiene reservado, que a su vez está esperando algo que no ocurrirá. Se puede producir en la comunicación de procesos y más frecuentemente en la gestión de recursos. Existen cuatro condiciones necesarias para que se pueda producir un deadlock:

- Los procesos necesitan acceso exclusivo a los recursos.
- Los procesos tienen que mantener ciertos recursos con acceso exclusivo mientras esperan por otros.
- No se le puede remover recursos asignados como acceso exclusivo a los procesos que están a la espera.
- Existe una cadena circular de procesos en la cual cada proceso posee uno o más de los recursos que necesita el siguiente proceso en la cadena.

2.1.3 Retraso indefinido o starvation

Un proceso se encuentra en starvation si es retrasado indefinidamente esperando un suceso que no puede ocurrir. Esta situación se puede producir si la gestión de recursos emplea un algoritmo en el que no se tenga en cuenta el tiempo de espera del proceso. Por ejemplo, si un proceso A bloquea un recurso, cuando le toca el turno al proceso B este no puede avanzar porque necesita el mismo recurso, al encontrarse bloqueado pierde el turno sin poder aprovechar el tiempo del procesador. Posteriormente le vuelve a tocar el turno al proceso A, como se aprecia, el proceso A avanza mientras que el proceso B se retrasa indefinidamente.

2.2 Problemática de fallos en las computadoras y las técnicas para minimizarlos

Ya desde hace muchos años la construcción del hardware computacional sigue un esquema modular enfocado en minimizar la incidencia de fallos [14]. En este esquema las computadoras son construidas de manera que el fallo en un módulo solo afecte dicho módulo, cada módulo a su vez es diseñado para que falle de manera rápida, esto es: el módulo o funciona de manera adecuada o deja de funcionar completamente, así evita que el fallo se propague hacia los otros módulos. Esta modularización y funcionamiento independiente, acompañado de una duplicidad o redundancia de dispositivos, extiende el tiempo medio entre fallos (TMEF) más de un orden de magnitud a varios años. Por ejemplo, los discos modernos son conocidos por tener un TMEF de más de cien mil horas, muchos sistemas usan discos espejo en donde se guarda la misma información en ambos usando tarjetas controladoras diferentes para cada uno.

Supongamos que se tiene un TMEF de diez mil horas y con un tiempo medio de recuperación de un fallo (TMRF) de 24 horas, asumiendo que tienen ambientes dirigidos a fallo independiente el TMEF de este par de discos (el TMEF de un fallo doble en estas 24 horas de recuperación) es de más de mil años según el estudio [14]. En la práctica, estos fallos no siempre ocurren de manera independiente pero se manejan TMRF de menos de 24 horas, lo cual conlleva a una alta disponibilidad lograda con este esquema.

En general, el hardware con tolerancia a fallos posee las siguientes características:

- Se descompone de manera jerárquica en módulos.
- Diseño de módulos con un TMEF superior a una año.
- Módulos de fallo rápido, cada módulo hace el trabajo de la manera esperada o para de trabajar.
- Detección rápida de fallos de los módulos.
- Módulos extra que pueden tomar el trabajo de los módulos que fallan. El tiempo que lleva retomar el trabajo para estos módulos incluyendo la detección de sus fallas constituye solo unos segundos. Esto da un aparente TMEF medido en milenios para esas partes con estructuras modulares.

Los sistemas de hardware resultantes que usan estas técnicas tienen TMEF medidos en siglos. Esto constituye un avance importante denominado hardware tolerante a fallos. Pero la fuente principal de los fallos en los sistemas no es el hardware, es el software, por lo que la tolerancia a fallos en el software también debe ser implementada.

Al seguir el ejemplo de las técnicas para tener un hardware tolerante a fallas, se puede enlistar las siguientes características para hacer software tolerante a fallas:

- Modularidad del software a través de procesos y mensajes.
- Contención de las fallas a través de módulos de software que fallen manera rápida.
- Duplicidad de procesos para tolerar problemas transitorios de software y de hardware.
- Mecanismo de transacciones para proveer integridad de datos y mensajes.
- Mecanismo de transacciones combinado con duplicidad de procesos para facilitar el manejo de excepciones y la tolerancia de fallos de software.

Así como en el hardware, la clave para la tolerancia a fallas del software es descomponer los sistemas grandes en módulos funcionales/estructurales de manera jerárquica, donde cada módulo es tanto una unidad de servicio así como una unidad de falla, lo que implica que una falla en un módulo no se propaga más allá de ese módulo. Para que estos módulos tengan independencia de fallo deben contener su propio hilo de ejecución, esto es: los módulos se mapean a procesos independientes. Los procesos proveen contención de las fallas evitando compartir su estado con otros procesos, y para proveer mecanismos de comunicación y coordinación entre ellos sin comprometer su independencia, el único medio de contacto permitido con otros procesos es vía mensajes asincrónicos.

Muchos de los fallos en producción en software tienen un comportamiento especial [14], en donde si el sistema es inicializado nuevamente y la operación que ocasiona el fallo es reintentada, estos fallos desaparecen en el segundo intento. Según algunos estudios [14]

más del 90% de los fallos en software pueden tener rutinas para recobrarse funcionalmente, y estas rutinas tienen un 76% de éxito en continuar con la ejecución del sistema. Con base en esto es que se establece que la duplicidad o redundancia de procesos, que viene a aportar una gran ayuda para evitar que el sistema falle del todo y extender considerablemente su TMEF.

Según [14], el establecer una tripleta de procesos en lugar de la duplicidad de procesos no incrementa el TMEF debido a otros factores en el sistema (como por ejemplo la intervención del humano u operador) que tienen un TMEF varias órdenes de magnitud peor. No se puede hacer mucho por resolver el error humano, lo que se puede hacer es tratar de reducir la intervención humana que se da al hacer labores de configuración, mantenimiento y operación de los sistemas, ya que incrementa los fallos en los mismos. Mucho de este estudio coincide con lo que se implementa en el modelo de actores en el lenguaje Erlang, como se verá posteriormente en este documento.

2.3 El modelo de actores

El modelo de actores [7] [1] es ideado para facilitar la programación concurrente, este enfoque usa el paso de mensajes para enviar información de un proceso al otro sin compartir memoria evitando regiones críticas. El modelo de actores es llevado a la práctica en algunos lenguajes y librerías (Ej. [20]: Erlang [25], Charm++ [32], Scala [36]) de manera exitosa. El grupo meta de usuarios de este modelo son los desarrolladores de aplicaciones que necesitan usar la programación concurrente y que obtienen un beneficio cuando minimizan la complejidad de la implementación de sus aplicaciones [1], esto

debido a que se benefician más si los programadores gastan su tiempo en expresar el "qué" y no se enfrascan en el "cómo" [1].

Fue propuesto por Carl Hewitt en 1973[15] . Surge específicamente porque los modelos de programación de ese entonces, en específico el cálculo lambda, no contemplaban de manera adecuada el procesamiento paralelo [15]. En este modelo, cada objeto es un actor, esta es una entidad que tiene una cola de mensajes o buzón y su propio comportamiento.

Los mensajes pueden ser intercambiados entre los actores y se almacenan en el buzón. Al recibir un mensaje, el comportamiento del actor se ejecuta. El actor puede enviar una serie de mensajes a otros actores o a sí mismo, crear una serie de actores y asumir un nuevo comportamiento para el próximo mensaje; cada actor está identificado con una dirección. En este modelo todas las comunicaciones se llevan a cabo de forma asincrónica, esto implica que el remitente no espera a que un mensaje sea recibido en el momento en que lo envió, solo sigue su ejecución.

Una característica importante es que todas las comunicaciones se producen por medio de mensajes: no hay un estado compartido entre los actores. Si un actor desea obtener información sobre el estado interno de otro actor, se tendrá que utilizar mensajes para solicitar esta información. Esto permite a los actores controlar completamente el acceso a su estado.

El procesamiento se empieza al enviar un mensaje a un actor, en el actor ocurre un evento cuando acepta el mensaje. Nótese que como el procesamiento es manejado por eventos, los actores que no están procesando algún mensaje no necesitan tiempo de procesador. El mensaje en sí mismo es un actor, el cual es un contenedor estándar de información a ser enviado a otro actor.

Al estar restringido un actor a procesar solo un mensaje a la vez, su estado se protege de la concurrencia, dado que máximo solo hay un hilo de ejecución realizando modificaciones sobre él. No existe el concepto de un estado global en el modelo de actores. Esta restricción facilita la forma en la que un desarrollador razona sobre los problemas de concurrencia al evitar en gran medida los problemas clásicos de programación concurrente descritos previamente en este capítulo, ya que, al no compartirse la memoria, la violación de exclusión y el retraso indefinido (starvation) es eliminado, mientras que el deadlock es minimizado [1].

El sistema de comunicaciones entre actores maneja un orden de primero en entrar – primero en salir (FIFO), lo que indica que el orden de entrega dicta el orden de recepción de mensajes por parte del actor, además el buzón se maneja mediante un espacio de memoria que va acumulando los mensajes para ser entregados cuando el actor pueda recibirlos [41]. En la figura 2.3.1 se ilustra este modelo.

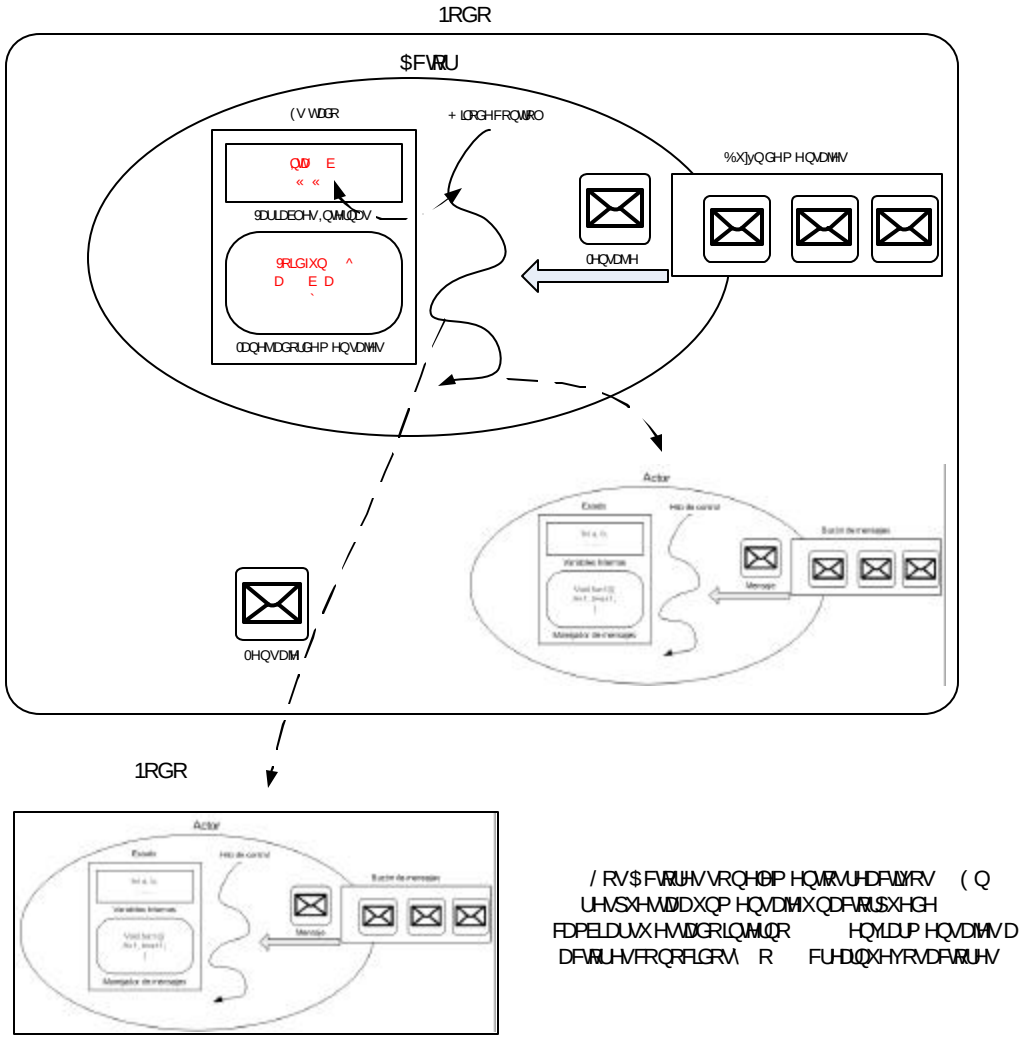


Figura No. 2.3.1 Diagrama de Modelo de Actores.

2.4 Tolerancia a fallas con el modelo de actores

En el apartado 2.2 se muestra el concepto de sistemas tolerantes a fallas y las principales características que estos poseen. Como se muestra en el apartado 2.3, el modelo de actores sigue el diseño en módulos con hilo de ejecución independiente que se comunican por medio de mensajes indicados en el apartado 2.2. Además en el apartado 2.2 se habla del concepto de jerarquía en los procesos. Algunos de los lenguajes o librerías [31] [42] existentes hoy en día basados en el modelo de actores implementan la idea de jerarquía de

procesos para crear sistemas tolerantes a fallos, el lenguaje Erlang por ejemplo contempla esto en su diseño.

Al analizar cómo se implementa en el lenguaje Erlang la jerarquía de procesos, se encuentra el concepto de árboles de supervisión, en donde un proceso puede ser obrero o puede ser a su vez un supervisor. Los procesos obreros son supervisados por su proceso padre, si un proceso obrero falla el proceso supervisor puede reiniciarlo, al reiniciarlo el supervisor tiene la opción de solo reiniciar el proceso que falló o reiniciar todos los subprocesos hermanos dependiendo de lo que se indique por el programador, como se muestra en la figura 2.4.1. Si un supervisor muere automáticamente son reiniciados todos sus procesos supervisados. Hay aspectos de configuración en donde se puede indicar que, si un proceso es reiniciado muchas veces en un tiempo dado, entonces el supervisor a cargo debe morir. Un proceso supervisor también puede ser supervisado, y el proceso supervisor a cargo del supervisor que muere tomará la decisión de qué hacer en ese caso. Esto previendo que tal vez al reiniciar los procesos a un nivel superior más funcionalidades son reiniciadas y con ello puede que desaparezca el problema en el sistema.

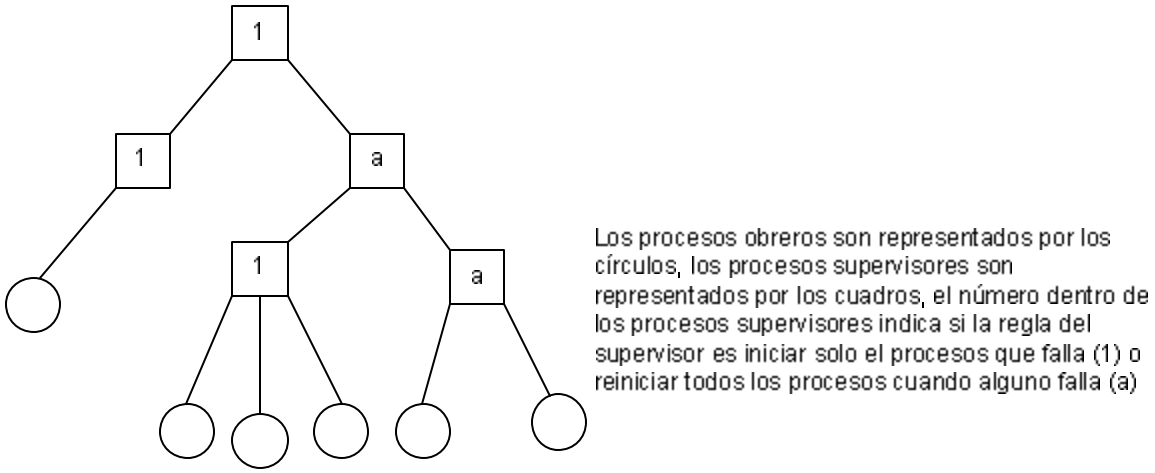


Figura No. 2.4.1 Diagrama de Árbol de Supervisión.

2.5 El lenguaje Erlang

Para dar una idea de las habilidades con las que se cuenta en el lenguaje Erlang para hacer las pruebas de esta investigación, se muestran y explican algunos ejemplos a continuación. Un corto programa secuencial (un solo proceso ejecutando el algoritmo) de cálculo de la función factorial se ve de la siguiente manera:

```
-module(math_test).
-export([fun_fact/1]).

fun_fact(0) -> 1;
fun_factl(Num) -> Num * factorial(Num-1).
```

En este las dos primeras líneas definen el módulo y la función exportable del módulo. Como se observa, el lenguaje es declarativo al estilo Prolog.

El lenguaje también usa pattern matching, tuplas y listas con primitivas para el manejo normal de estos tipos complejos de datos, a continuación un ejemplo:

```
-module(math_test).
-export([calc_area/1]).

calc_area({cuadrado, Lado}) ->
                          Lado * Lado;
calc_area({rectangulo, Alto, Ancho}) ->
                          Alto *Ancho;
calc_area({circulo, Radio}) ->
                          3.1416 * Radio * Radio.
```

Al invocar en el modo interactivo (en el programa shell de Erlang) el siguiente código, se observan los siguientes resultados:

```
>Arrea_a_calcular = {cuadrado, 6}.
{cuadrado,6}
> math_test:calc_area(Arrea_a_calcular).

36
```

Como parte de las ventajas que ofrece el lenguaje, este maneja concurrencia:

```
-module(test_paralelo).
-export([cargar_proceso/0, servidor_duplicar/0, prueba/0]).

cargar_proceso() ->
      spawn(test_paralelo, servidor_duplicar, []).

servidor_duplicar() ->
      receive
            {From, Num} ->
                  Resultado= Num * 2,
                  From ! Resultado
      end.

prueba() ->
             Id=cargar_proceso(),
             Id!{self(), 40},
             receive
                   Num ->
                                Num
             end.
```

Al ejecutar las siguientes instrucciones:

```
test_paralelo:prueba().
```

Se crea un nuevo proceso paralelo al proceso que lo invoca y le manda el mensaje {self(), 40}, lo cual hará que este proceso paralelo mande el mensaje 80 al invocador. Nótese que la función "spawn" crea un nuevo proceso en memoria, el cual es un actor bajo la teoría del modelo de actores, la primitiva "receive" espera hasta que se recibe un mensaje, el operador ! es usado para enviar un mensaje a un proceso designado por un identificador que antecede el operador y un mensaje que es el término que sigue del operador. "self" es una primitiva que retorna el identificador del proceso actual.

El lenguaje cuenta con la capacidad de manejar programación distribuida. A continuación se da un pequeño ejemplo de un servidor simple para un proceso que en vez de duplicar hace una multiplicación en un nodo remoto (otra computadora):

```
-module(test_distribuido).
-export([cargar_servidor/2, servidor_multiplicador/1, prueba/2]).

cargar_servidor(NombreNodoRemoto, Multiplicador) ->
                         spawn(NombreNodoRemoto, test_distribuido,
                         servidor_multiplicador, [Multiplicador]).

servidor_multiplicador(Multiplicador) ->
       receive
             {_From, morir} -> true;
             {From, Num} ->
                       Resultado= Num * Multiplicador,

                        From ! {self(), Resultado},
                       servidor_multiplicador(Multiplicador)
       end.

prueba(Num_a_multiplicar, NombreNodoRemoto) ->
             Id=cargar_servidor(NombreNodoRemoto, 50),
             Id!{self(), Num_a_multiplicar},
             {IdServidor, Resultado} = receive
                         {TmpIdServidor, TmpNum} ->
                                {TmpIdServidor, TmpNum}
                          end,
              io:format("Primer resultado de multiplicar ~w * 50: ~w~n",
                                       [Num_a_multiplicar, Resultado]),
             IdServidor!{self(), Resultado},
             SegundoResultado = receive
                                {_, Tmp_segundoResultado} ->
                                       Tmp_segundoResultado
                                end,
             IdServidor!{self(), morir},
             io:format("Segundo resultado de multiplicar ~w * 50: ~w~n",
                                            [Resultado, SegundoResultado]).
```

Como se puede observar, se sigue una estructura en donde la función "spawn" también puede ser usada para invocar procesos en un nodo remoto.

Al ejecutar las siguientes instrucciones:

```
(b@pc)16> test_distribuido:prueba(5, a@pc).
Primer resultado de multiplicar 5 * 50: 250
Segundo resultado de multiplicar 250 * 50:
12500
ok
```

Se puede observar que el programa logra contactar la computadora con el nombre a@pc para hacer las multiplicaciones requeridas, basta con que la red esté configurada correctamente, y que el nodo remoto (la otra computadora) este corriendo el Erlang y tenga acceso al archivo binario del programa compilado para que todo funcione de la manera esperada. Nótese que el programa puede invocar funciones en la otra computadora aunque esta no esté corriendo más que el Erlang. Otra cosa que se puede notar es que para contestar un mensaje recibido de otro nodo, el identificador de proceso se maneja de igual manera que en el proceso paralelo descrito anteriormente, esto hace muy fácil pasar en Erlang de la programación paralela a la programación distribuida con el modelo de actores.

2.6 Inferencia estadística

Esta investigación consta de varias partes, una de ellas es la evaluación del comportamiento de los smartphones una vez obtenido una serie de datos a partir de algunos experimentos. La inferencia estadística es un método formal que permite tomar decisiones con base en un muestreo de una población y que se usa para dar un respaldo más riguroso al análisis de los resultados obtenidos.

Uno de los métodos usados es el contraste de hipótesis, con este método se plantea una hipótesis nula (H_0) y otra alterna (H_a), donde la hipótesis alterna es la que el investigador está interesado en verificar como cierta o falsa, y la hipótesis nula corresponde a aquella que no provee información y no es más que una confirmación del estado actual del conocimiento. Por lo general las hipótesis tienen que ver con la media de la población o poblaciones, la intención es tratar de plantear una hipótesis nula que se pueda rechazar con los datos de la muestra para así comprobar que se cumple la hipótesis alternativa. Por ejemplo la hipótesis nula puede ser que las medias entre dos poblaciones son iguales, y la hipótesis alternativa es que no son iguales. Esta prueba se hace usando un margen de error comúnmente conocido como Alpha, por lo general este Alpha se escoge con un valor de 0.05, lo que indica que cuando rechazamos H_0 hay un 5% de margen de error tipo uno, el error tipo uno sería rechazar H_0 cuando es cierta. En esta investigación se usó el Alpha de 0.05.

También en las pruebas de contraste existe el concepto del p-valor o también indicado como valor p, el p-valor es la posibilidad de error tipo uno al rechazar H_0 para esa prueba

en específico. Si el p-valor obtenido es mayor al Alpha escogido, entonces se dice que no se puede rechazar H_0 con la certeza necesaria en la prueba, si el p-valor es menor o igual al Alpha entonces se rechaza H_0, y se acepta H_a.

El p-valor es conveniente indicarlo ya que así el lector de los resultados podrá saber si a su criterio es aún posible rechazar H_0, ya que este lector puede considerar un Alpha diferente al planteado. A este lector el p-valor le dará la información requerida para rechazar o no la hipótesis nula con su margen de confianza.

El contraste de hipótesis es útil a la hora de analizar los datos de 2 poblaciones, pero cuando hay más poblaciones que analizar o cuando se necesita el análisis de varios factores es preferible utilizar el método análisis de varianza o ANOVA [5]. En los experimentos diseñados en esta investigación se usa un solo factor que incide en el cambio de los resultados de los experimentos, este factor es el tamaño del problema. Es con los distintos tamaños del problema en 3 diferentes escenarios que se hacen las comparaciones respectivas. Tal vez con un diseño diferente de los experimentos y con un poco más de tiempo sea conveniente usar el ANOVA, pero con el tiempo asignado y el diseño de los experimentos de esta investigación se cree más conveniente el uso de contraste de hipótesis entre dos medias de poblaciones grandes.

Esta prueba de contraste exige dos características fundamentales en el muestreo de la información: independencia y aleatoriedad. La “Muestra Aleatoria Simple” [6] es aquella que realiza “n” experimentos para obtener las “n” observaciones del valor que de una

población de valores infinito, esta es la técnica aplicada en esta investigación, por lo tanto el muestreo es aleatorio. Por otra parte, la técnica aplicada de "Muestra Aleatoria Simple" en esta investigación se caracteriza por haber hecho selecciones de manera independiente ya que ninguna observación se ve afectada por otra, por lo tanto se cumple también con la independencia requerida, y se tiene así la aleatoriedad e independencia necesarias para las pruebas de contraste de hipótesis.

Tamaños de muestra mayores a 30 elementos se consideran ya muestras grandes [24] y suficientes para usar la distribución normal para el muestreo sin importar la distribución de la población origen. Es por esto que se usa la curva de distribución normal como parámetro de las pruebas de contraste de hipótesis en esta investigación, ya que se procuró hacer muestreos de 35 elementos cada prueba realizada.

Otro concepto estadístico usado en esta investigación es el "Coeficiente de Variabilidad". Este se define como sigue: sea "s" igual a la desviación estándar de la muestra y $\bar{x}$ su media, la fórmula del coeficiente de variabilidad estaría dada por:

$$CV = \frac{s}{\bar{x}} * 100$$

Este concepto es muy útil para saber qué tan dispersas son las muestras que se tienen para un tipo de prueba, ya que dan un porcentaje que indica qué tan desviados están los datos de la media en la muestra.

CAPÍTULO 3. METODOLOGÍA

Para este análisis se parte de una versión de un lenguaje basado en el modelo de actores que se distribuyó por su casa matriz con fuentes abiertas para correr en una computadora de escritorio. La premisa es que si la casa matriz saca una versión destinada a ser usada en una computadora de escritorio de manera oficial, entonces esto establece una base aceptable en el presente estudio para migrar este lenguaje (sin cambiar nada en código fuente) a un Smartphone. Esto servirá para establecer comparaciones válidas en ambas plataformas (computadoras de escritorio y smartphones).

Para la presente investigación fue escogido el lenguaje llamado Erlang [8] [18], este lenguaje es migrado a la plataforma smartphone y fue escogido porque es un lenguaje comercialmente reconocido que funciona en su propia máquina virtual. Se dejan de lado librerías montadas sobre máquinas virtuales de Java porque son montadas en máquinas virtuales que no son diseñadas exclusivamente para manejo del modelo de actores, lo cual puede dejar la duda acerca del origen de cualquier deficiencia que se pueda encontrar, ya que la deficiencia puede estar dada por el modelo de actores o por la máquina virtual sobre la que corre, por no ser construida específicamente para el manejo de modelo de actores. En esta investigación se desea saber con certeza las deficiencias propias del modelo de actores sin dejar dudas por la participación de máquinas virtuales de un tercero.

Aunque ya hay una versión denominada Erlang Embedded [21] que podría probarse en dispositivos móviles, el proyecto de Erlang Embebed ya no está vigente debido a que los

archivos que se manejan en el sitio no están actualizados desde Octubre del 2013 y no se conoce de aplicaciones comerciales que indiquen el uso de esta versión de Erlang. Por el contrario la versión estándar [30] de Erlang es usada y probada de manera comercial por aplicaciones como Facebook y Whatsapp [30], además, la versión normal está vigente y se encuentra evolucionando con nuevas versiones continuamente. Inclusive está adaptada para ser compilada para Android, lo que indica que ya soporta esa plataforma, la cual es una de las principales en los smartphones. Los problemas antes mencionados de Erlang Embedded le restan atractivo para un tomador de decisiones que desea utilizar Erlang para migrar o crear sus sistemas, por lo que se deja de lado para utilizar la versión estándar en esta investigación.

3.1 Alcance

Para la presente investigación fue necesario migrar el lenguaje Erlang para comprobar su funcionamiento. Se migró las versión R12B-5 al smartphone con menos capacidad y el 18.1 al smartphone con más capacidad en las pruebas hechas. La primera versión no es muy grande y por eso fue la escogida para ser corrida en el smartphone con menos capacidad, para evitar la degradación del desempeño debido a la existencia de funcionalidades que no son importantes en las pruebas ya que estas abarcan solo los conceptos básicos del lenguaje en sí. Esta versión de Erlang es usada tanto en la computadora de escritorio (computadora 1) como en el smartphone básico para correr las pruebas. Por otra parte la versión 18.1 es la última versión del Erlang al momento de realizar esta investigación y se consideró adecuada para el smartphone con más capacidad, debido a que este dispositivo cuenta con los recursos suficientes de memoria y capacidad de procesamiento como para desenvolverse con la última versión. En las computadoras de escritorio usadas para las pruebas fue instalada también la versión OTP 18.1.

Se hacen pruebas usando el algoritmo Quicksort. Este algoritmo se utiliza con la intención de medir el modelo de actores trabajando en ambientes donde se requiera muchos hilos con cargas de trabajo livianas. Aplicaciones para manejo de muchos dispositivos o manejo de muchos sensores son representativas de este tipo de aplicaciones. Solo para dar un contexto de cómo se implementa el Quicksort de manera paralela y distribuida, se explica el funcionamiento del algoritmo implementado. En la figura 3.2.1 se puede apreciar en números rojos los pivotes escogidos en las distintas iteraciones del algoritmo. En cada

iteración, una vez escogido el pivote, que es el último elemento del arreglo, se divide el vector en 2 subvectores, uno con elementos menores al pivote y otro con elementos mayores al pivote, después recursivamente se aplica el algoritmo en cada uno de los subvectores obtenidos.

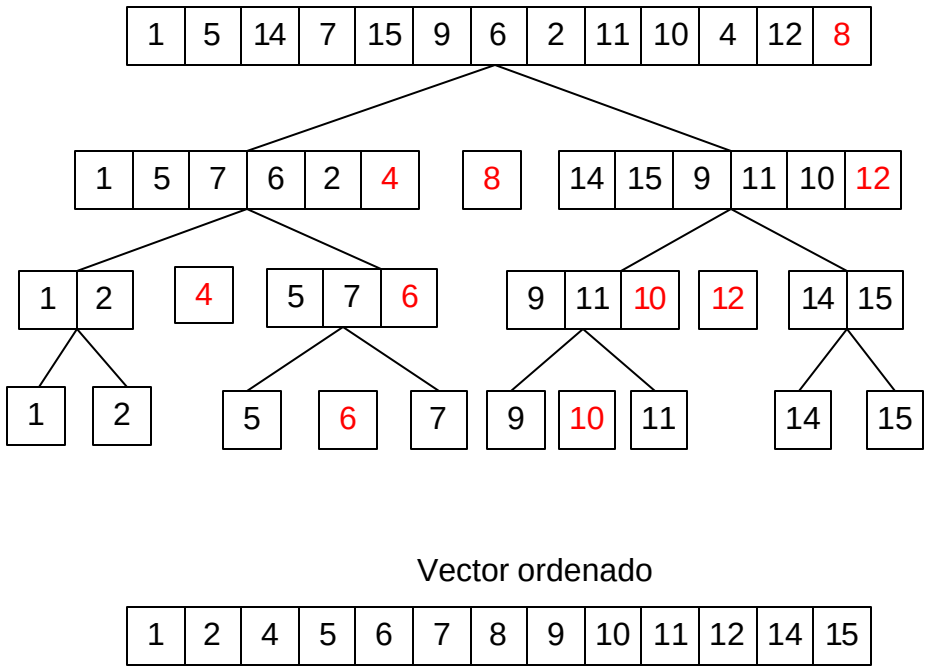


Figura No. 3.1.1. Algoritmo Quicksort Implementado

Al final, cuando se juntan los subvectores mínimos (de un elemento) se tendrá el vector ordenado. La modificación para el enfoque paralelo del algoritmo consiste en que cada vez que se generen 2 subvectores, se crearán 2 hilos o procesos independientes que se encargan de aplicar el algoritmo a los subvectores. Para la variante del algoritmo con procesamiento distribuido, estos hilos son abiertos para pedir el ordenamiento que hace cada uno de ellos en la máquina remota, en donde esta a su vez posteriormente mandará a abrir hilos en la máquina local en futuras iteraciones del algoritmo.

Las pruebas del algoritmo Quicksort se hicieron en todos los ambientes propuestos en esta investigación (secuencial, paralelo y distribuido) para el smartphone básico. Para el smartphone avanzado no fue posible por restricciones de tiempo hacer las pruebas en el ambiente distribuido, ya que se debían hacer ajustes para que el smartphone avanzado funcionara de manera distribuida y con ello extender mucho los procesos de pruebas. Con las pruebas distribuidas que se hacen el en smartphone básico ya se puede establecer una idea de qué esperar del modelo de actores en un ambiente distribuido para estos dispositivos, y se deja como un trabajo futuro el extender los resultados haciendo esas pruebas restantes en el smartphone avanzado, si es que se quiere una certeza aún mayor en el ambiente distribuido.

También son usados los algoritmos de multiplicación de matrices [33] y de búsqueda en anchura o BFS [37] [23] por su siglas en inglés (en adelante se usa en ocasiones BFS para referirse a este algoritmo). La versión que comúnmente se maneja de la multiplicación de matrices en su versión paralela toma la primera matriz y la subdivide en matrices más pequeñas (tantas como procesos o hilos se desee tener), que son matrices con el mismo número de columnas de la primera matriz a multiplicar pero solo con algunas de las filas de esta primera matriz original. Después varios procesos en paralelo multiplican cada uno una de estas matrices pequeñas (una por proceso o hilo) por la segunda matriz completa y al final cuando todos los procesos terminan se unen los resultados para obtener la matriz resultante completa. El algoritmo tradicional y el paralelo fueron implementados en la tesis, no así el distribuido. Esto quedará como un trabajo por hacer, se deja por fuera el algoritmo distribuido por razones de tiempo.

Parta el caso del BFS, la implementación en paralelo realiza la búsqueda de todos los nodos adyacentes no visitados aún, adyacentes con respecto a un conjunto de nodos dado en cada iteración, al final de la iteración el conjunto de nodos inicial pasan a marcarse como visitados y el nuevo conjunto de nodos adyacentes pasan a ser el grupo de nodos base para la siguiente iteración. Para saber los nodos adyacentes de un grupo de nodos en la iteración, se particiona todos los nodos del grupo en subgrupos, el número de subgrupos lo determina el número de procesos concurrentes disponibles para realizar el trabajo. Luego se le asigna a cada uno de esos procesos concurrentes o hilos un subgrupo de nodos para que dé como resultado todos los nodos adyacentes de ese subgrupo de nodos. Al final de la iteración se unen todos los resultados de los procesos concurrentes para conformar el grupo que se resolverá en la siguiente iteración, este grupo es filtrado para sacar del mismo los nodos ya visitados anteriormente. Con cada iteración se va generando información de la distancia a la que se encuentra cada nodo del nodo inicial, el nodo inicial es aquel que se escoge como primer grupo de nodos (esta vez de un solo elemento) para iniciar la primera iteración del algoritmo. Adicionalmente se va registrando información de cuál es el nodo padre de cada nodo en el recorrido BFS, lo cual es muy útil para saber el camino más corto entre un nodo cualquiera y el nodo inicial escogido para la primera iteración. La implementación tradicional secuencial y la paralela antes explicada fueron implementadas, sin embargo por las ya comentadas razones de tiempo no se implementa la versión distribuida de este algoritmo.

También se hacen pruebas de gasto de energía en el smartphone avanzado, pero al realizarse las pruebas se encuentra que algunos programas, principalmente el Quicksort, no eran muy adecuados para esta prueba de la manera en que se había implementado, porque duraba muy poco ejecutándose y si se aumentaba el tamaño de la prueba para que durara más, entonces sobrepasaba la memoria existente en el dispositivo. Aun así se muestra en los resultados las medidas que se obtuvieron de gasto de energía. Por falta de tiempo no se pudo adaptar el Quicksort para ejecutar repeticiones de corridas que dieran un tiempo más largo para obtener promedios de gasto de energía más confiables. Para futuros desarrollos podría considerarse el extender el programa para que sea apto para mediciones de energía. En el cuadro 3.1.1 se muestra la estructura de los módulos que componen los distintos programas y el número de líneas de cada uno.

Nombre Programa	Lenguaje	Número de líneas de código	Descripción de fuente
Quicksort	C	713(tesis_qsort.c)+123(misc_functions.c) + 12(misc_functions.h) = 848	Se desglosa en 2 módulos en C, tesis_qsort.c es el programa principal y misc_functions.c es un programa con funciones misceláneas para lectura de archivos u otros propósitos generales. En este programa se manejan varias funciones que implementan el algoritmo Quicksort secuencial, el paralelo y el distribuido.
Quicksort	Erlang	149 (tesis_qsort.erl) + 59 (misc_functions.erl) = 208	Se desglosa en 2 módulos en Erlang, tesis_qsort.erl es el programa principal y misc_functions.erl es un programa con funciones misceláneas para lectura de archivos u otros propósitos generales. En este programa se manejan varias funciones que implementan el algoritmo Quicksort secuencial, el paralelo y el distribuido.

Multiplicación de matrices	C	195 (matmul.c) + 123(misc_functions.c) + 12(misc_functions.h) = 330	Se desglosa en 2 módulos en C, matmul.c es el programa principal y misc_functions.c es un programa con funciones misceláneas para lectura de archivos u otros propósitos generales. En este programa se manejan varias funciones que implementan el algoritmo de multiplicación de matrices secuencial y el paralelo.
Multiplicación de matrices	Erlang	166 (matmul.erl, no se consideran comentarios internos de 32 líneas, este módulo estaba comentado pero el hecho en C no lo estaba) +59 (misc_functions.erl) = 225	Se desglosa en 2 módulos en Erlang, matmul.erl es el programa principal y misc_functions.erl es un programa con funciones misceláneas para lectura de archivos u otros propósitos generales. En este programa se manejan varias funciones que implementan el algoritmo de multiplicación de matrices secuencial y el paralelo.
Búsqueda en Anchura (BFS)	C	253 (bfs.c) + 130 (adj_mat.c) + 14 (adj_mat.h) + 76 (queue_man.c) + 18 (queue_man h) + 123(misc_functions.c) + 12(misc_functions.h) = 626	Se desglosa en 4 módulos en C, bfs.c es el programa principal, adj_mat.c es el módulo que contiene las funciones de manejo de la matriz de adyacencia que representa el grafo dirigido, queue_man.c es el módulo con las funciones relacionadas con manejo de colas y misc_functions.erl es un programa con funciones misceláneas para lectura de archivos y otros propósitos generales. En este programa se manejan varias funciones que implementan el BFS secuencial y el paralelo.
Búsqueda en Anchura (BFS)	Erlang	260 (bfs.erl, no se consideran comentarios internos de 59 líneas, este módulo estaba comentado pero el hecho en C no lo estaba) + 59(misc_functions.c) + 17(misc header) = 336	Se desglosa en 2 módulos en Erlang, bfs.erl es el programa principal y misc_functions.erl es un programa con funciones misceláneas para lectura de archivos y otros propósitos generales. En este programa se manejan varias funciones que implementan el BFS secuencial y el paralelo.

Cuadro 3.1.1 Programas de Pruebas

Con respecto a la tolerancia a fallas que se implementa en Erlang usando el modelo de actores, no es posible comparar esta característica con el enfoque de hilos en C ya que este no implementa la tolerancia a fallas. El implementar estas pruebas sería con fines ilustrativos, y esto le resta importancia a esta prueba. Otra cosa que se debe tomar en cuenta es algo ya mencionado anteriormente en este documento, en donde se indica que la última versión de Erlang al momento de hacer esta investigación (OTP 18.1) ya contempla la compilación con el método "crosscompiling" para la plataforma Android. Esto implica que la tolerancia a fallas de Erlang ya funciona en los smartphones que usan Android como sistema operativo, por lo que no es necesario verificar si este aspecto funciona o no en el smartphone.

3.2 Instalación de ambiente de pruebas

Para la presente investigación fue necesario migrar el lenguaje Erlang para comprobar su funcionamiento en los dispositivos smartphone. Para ello fue necesario instalar Linux en el smartphone básico y hacer algunos cambios e instalaciones en el sistema operativo Android del smartphone avanzado. Para las pruebas se necesitaron 2 computadoras personales y 2 smartphones.

3.2.1 Dispositivos usados en las pruebas

Las especificaciones técnicas se pueden apreciar en el cuadro 3.2.1.1, las imágenes de los dispositivos se pueden apreciar en las figuras 3.2.1.1, 3.2.1.2, 3.2.1.3 y 3.2.1.4. Estas imágenes son fotografías de los dispositivos que realmente se usaron en las pruebas.

Característica	Smartphone Básico	Smartphone Avanzado o QC	Computadora 1	Computadora 2
Modelo	HTC Wizard 8125	HTC Desire 610	Computadora Desktop	Fujitsu Lifebook T4220
CPU	Texas Instruments OMAP 850, 195 MHz, 1 núcleo	Quad-core 1.2 GHz Cortex-A7, 4 núcleos	Intel Q9550 2.83 GHz, 4 núcleos.	Intel Core 2 Duo T8300, 2.2GHz, 2 núcleos
Memoria	64 MB RAM/ 128 MB para almacenar.	1 GB RAM/ 8 GB para almacenar.	8 GB, 2 TB disco duro	4GB RAM, 150 GB Disco Duro
Sistema Operativo de Fábrica	Windows Mobile (5.0 Pocket PC)	Windows Mobile (5.0 Pocket PC)	Windows 7	Microsoft Windows Vista Business

Cuadro 3.2.1.1 Dispositivos Usados en las Pruebas

Figura No. 3.2.1.1 Dispositivo Smartphone Básico

Figura No. 3.2.1.2 Dispositivo Smartphone Avanzado o QC

Figura No. 3.2.1.3 Dispositivo Computadora 1

Figura No. 3.2.1.4 Dispositivo Computadora 2

3.2.2 Pasos para preparar los dispositivos

A las 2 computadoras se les instaló Ubuntu 9.10 en una partición física del disco duro, mientras que al smartphone básico se le instala una versión de Linux denominada Linwizard. Esta versión está especialmente configurada para este dispositivo [39] pero se encontraba solo con lo básico. Una vez instalado Linux en este Smartphone, fue necesario instalar una serie de paquetes para que el compilador c/c++ para smartphone estuviera preparado. La compilación del lenguaje Erlang se hace en el dispositivo, lo que implica instalar todo lo necesario en el mismo para poder compilar el fuente del lenguaje que es extenso. Esto requirió configurar aspectos como el manejo de memora virtual, ya que el dispositivo se quedaba corto para compilar un fuente tan grande si se usaba solo la memoria principal, por eso fue necesario investigar cómo lograr extender de alguna manera la memoria virtualizando la misma en memoria secundaria en una tarjeta mini SD aceptada por el dispositivo con 2GB de capacidad.

En cuanto al dispositivo smartphone avanzado, fue necesario instalar la aplicación Terminal Ide, la cual cuenta con un compilador de código C para compilar los fuentes de las pruebas. Adicionalmente fue necesario compilar el Erlang 18.1 usando la técnica "crosscompiling" para compilarlo para ser usado en la plataforma Android. Es hasta esta versión que se empieza a ofrecer un conjunto de archivos de configuración que dan la capacidad de compilar para Android, además es la última versión de Erlang a la fecha en que esta investigación se realiza. La información para hacer esto se encuentra dentro de los mismos archivos de ayuda ofrecidos con las fuentes del lenguaje.

3.2.3 Programación de algoritmos de pruebas

En este paso, una vez corregidos los problemas que se presentaron en el paso anterior, se hacen 3 programas. Para cada programa se hace 2 versiones, una versión en Erlang y la otra en C, estos son los programas usados para poner a prueba los dispositivos.

El primer programa es uno de los más básicos en programación paralela llamado Quicksort [27], se usa el Quicksort porque se quiere probar la respuesta a aplicaciones que manejen muchos hilos o procesos. En el caso de Quicksort se implementa el algoritmo para los 3 escenarios posibles de prueba, los cuales son: el algoritmo secuencial, algoritmo paralelo y el algoritmo funcionando de manera distribuida.

Para el caso del segundo programa escogido, el cual es multiplicación de matrices, se programa tanto en C como en Erlang el algoritmo secuencial y algoritmo paralelo. Ya para el último programa, el cual es el de Búsqueda en Anchura o BFS, se implementa el algoritmo secuencial y el algoritmo paralelo, los 2 algoritmos en C y en Erlang.

Para el caso de las implementaciones en C de los algoritmos BFS y multiplicación de matrices que se ejecutan de manera paralela, se acepta un parámetro para especificar el número de hilos o procesos usados para ejecutar el algoritmo, el cual es asignado con valor de 4 para usar esa cantidad de hilos en las pruebas, uno para cada procesador de los dispositivos Computadora1 y smartphone avanzado. Las versiones en C de todos los programas usan memoria compartida con hilos PThread. En el caso del Quicksort distribuido en C, se usan sockets [26] con PThreads, en el Quicksort paralelo y distribuido

no se acepta un parámetro con el número de hilos debido a que el algoritmo abre tantos como sea necesario dependiendo del tamaño del problema.

Las fuentes creadas fueron de manera satisfactoria corridas en todas las plataformas compilándolas según correspondía, y no se tuvo que hacer cambios en los mismos para adaptarlos a ninguna plataforma. Adicionalmente se tuvo que crear un programa en C para la generación de grafos aleatoriamente conectados, esto debido a que las pruebas del BFS se diseñan de manera tal que hay varios tamaños de grafos a resolver y se necesitaba una herramienta que permitiera generar los grafos con una cantidad de nodos deseada y una cantidad promedio de arcos por cada nodo. Con este programa se generan varios grafos de manera satisfactoria.

3.2.4 Diseño de las pruebas

El propósito de las pruebas no es comparar Erlang con C de manera directa, lo que se quiere es obtener una medición de la variación en tiempo de ejecución y de consumo de memoria en ambos lenguajes al resolver una problema del mismo tamaño. Esta variación se toma como aceptable en el caso de la PC, por ejemplo, si se obtiene que el algoritmo en Erlang dura un 25% más de tiempo resolviendo un problema que en C, entonces ese 25% se toma como lo normal en el caso de la PC. Posteriormente se ejecuta la misma técnica en el smartphone para obtener la variación en este, luego se comparan estas 2 variaciones obtenidas para saber si en el smartphone se obtiene un mejor o peor desempeño. Lo que impone la base de las variaciones es el algoritmo en C en ambas plataformas; el compilador C es el propio de cada plataforma, por lo que se supone que establece una base aceptable para obtener la variación en cada plataforma. Como se comentó anteriormente, Erlang fue compilado en cada plataforma partiendo del mismo fuente (versión R12B-5 o 18.1) para asegurar que la prueba sea válida al comparar la ejecución en el mismo Erlang en ambas plataformas de una prueba del mismo tamaño.

Este recurso de usar la PC como base es usado en otro estudio. El artículo expuesto [10], indica en las páginas 4 y 6 que en la investigación con resultados similares en las desktop se obtienen resultados diferentes en los dispositivos y esto es relevante. Este estudio parte de los resultados de las desktop, asume que si se obtienen diferencias en los dispositivos móviles es por cuestiones propias de los mismos y esto aporta los resultados de interés en la investigación. Este último escenario solo fue probado con el algoritmo Quicksort usando

el smartphone básico como se comentó en la sección “Alcance” de la metodología en este documento.

Como se explicó, lo que se quiere es comparar variaciones, las variaciones se obtienen para tres escenarios. El primero es el secuencial, en donde el algoritmo se ejecuta en un solo hilo en el caso de C y en un solo proceso en el caso de Erlang; en el segundo escenario se ejecuta el algoritmo paralelo usando varios hilos de ejecución en C y en varios procesos en Erlang. En el último escenario es el distribuido en donde se ejecuta varios procesos de manera local y remota en Erlang con 2 dispositivos conectados a red, también en este escenario se ejecutan varios hilos de forma local y remota en C usando sockets para la trasferencia de datos.

Fue necesario escoger el tamaño máximo de las pruebas de cada Algoritmo basándose en la capacidad del smartphone que participaba en la prueba, ya que es este dispositivo el que pone los límites porque las computadoras tienen límites superiores debido a sus recursos superiores. Una vez determinados los tamaños de las pruebas en el smartphone correspondiente entonces estas se aplican con el mismo tamaño en la PC. Es importante aclarar que la virtualización de memoria que se instaló para compilar el Erlang en el smartphone básico fue desactivada para las pruebas, ya que se quiere evitar retrasos en los procesos producto de la paginación que hiciera el dispositivo. Los dispositivos smartphone cuentan con memoria inferior a las computadoras de prueba, por lo que las pruebas no sobrepasaban ese rango de memoria de las computadoras nunca. Esto indica que no era necesario desactivar la memoria virtual en las PC porque su memoria RAM (8 GB en

Computadora 1 y 4GB en Computadora 2) era más que suficiente para ejecutar estas pruebas.

En el caso del último escenario, el cual es el distribuido, se conectan 2 dispositivos en red para compartir el procesamiento requerido para ejecutar la prueba. En el caso del smartphone básico, este se conectó a la PC (Computadora 1) mediante un puerto USB, y en el caso de la conexión de PC vs PC (Computadora 1 Vs Computadora 2) se usó un cable de red crossover para conectar los dos equipos. Para el escenario distribuido con el algoritmo Quicksort, una vez determinado el tamaño de las pruebas para el caso paralelo se decide usar el mismo tamaño de pruebas para el caso distribuido, ya que aunque la memoria consumida en el caso distribuido podría ser menor que el caso paralelo para una prueba del mismo tamaño (por abrirse menos hilos locales), se espera un mayor tiempo de ejecución porque los datos entre los hilos o procesos ahora viajan por red en lugar de viajar a posiciones de memoria local de cada dispositivo.

A manera de recordatorio, en este escenario se usa un enfoque muy similar al enfoque paralelo, pero cada vez que se abren los 2 nuevos hilos de ejecución para ordenar los 2 subvectores, estos hilos invocan procesos en el dispositivo remoto para hacer el ordenamiento (que a su vez hará lo mismo invocando un ordenamiento remoto de vectores más pequeños en el dispositivo inicial y así consecutivamente). En este escenario (el distribuido) se ejecutan las pruebas usando dos PC para obtener las variaciones de C con respecto a Erlang que sirven como base para comparar las variaciones en el caso del smartphone básico conectado con la PC.

Para cada escenario se ejecutaron pruebas de varios tamaños para ver cómo se comportan los dispositivos según el tamaño del problema a resolver. Cada prueba de un tamaño determinado se ejecutó varias veces para obtener un promedio del tiempo de ejecución y consumo de memoria. Para ejecutar la prueba varias veces se creó un archivo por lotes en Linux que repetía la prueba 35 veces.

Después de ejecutar varias veces la prueba, para un tamaño determinado del vector se obtenía una media y un coeficiente de variabilidad de la memoria consumida y el tiempo tomado para resolver el problema. El coeficiente de variabilidad sirve para saber si el dispositivo estaba estabilizado a la hora de hacer la prueba. Si se obtenía un coeficiente de variabilidad muy grande con respecto a otras pruebas de distinto tamaño hechas en el dispositivo (PC o smartphone), entonces se examinaban los datos de las repeticiones. Si se encontraba que solo algunas repeticiones presentaban desviaciones muy grandes al promedio, entonces se repetía la prueba por completo, esto porque se asumía que el dispositivo no estaba estabilizado en ese momento y muy probablemente se encontraba ejecutando tareas alternas que normalmente no ejecuta (tareas atípicas demandadas por el sistema operativo).

Una vez obtenido el conjunto de datos que conforman el muestreo para un caso en específico, se procede a analizar los datos de dos maneras, una es gráficamente y la otra es con pruebas de contraste de hipótesis. Las pruebas estadísticas de contraste de hipótesis ayudan a respaldar de una manera más convincente los datos mostrados en los gráficos al

comparar las medias obtenidas a partir del muestreo obtenido. Esto aporta desde un enfoque estadístico un argumento de peso en el estudio de los datos. Estas pruebas estadísticas exigen el formalismo requerido para obtener conclusiones a partir de métodos que indican, por ejemplo, que la cantidad de muestras suficiente debe partir de 30 para aplicar la curva de probabilidad normal como el parámetro a usar en las pruebas.

Los datos obtenidos sirven para llenar unas hojas de cálculo de análisis que se discutirán más adelante en este documento.

CAPÍTULO 4. RESULTADOS OBTENIDOS

Los resultados de las pruebas son expuestos en varios gráficos basados en datos insertados en una hoja de cálculo diseñada para indicar si el smartphone participante en la prueba se desenvuelve correctamente en los diferentes escenarios de pruebas para distintos tamaños de las mismas.

4.1 Estructura de cuadros de análisis para obtener los gráficos finales

Para cada escenario de pruebas (secuencial, paralelo y distribuido) se capturaron los resultados en archivos de texto que posteriormente se insertaron en hojas de cálculo para obtener el promedio del tiempo de ejecución y del gasto de memoria. El tiempo es medido en segundos con números a seis decimales, mientras que la memoria es medida en KB. Para medir la memoria se consulta la variable VmPeak reportada en el archivo /proc/[00pid]/status, en donde [00pid] es el proceso a nivel del sistema operativo que realiza la prueba. Es importante aclarar que la variable VmPeak es comúnmente usada para medir la memoria máxima ocupada por el proceso, pero no se limita solo a la memoria RAM del proceso sino también a la memoria utilizada por las librerías compartidas que usa el proceso. Usar la variable VmPeak junto con la medición de tiempo de ejecución en segundos es una técnica usada en la Zona de Desarrolladores de Intel [35] para ver la efectividad de librerías de programación.

4.1.1 Archivo básico de pruebas

Como se indicó anteriormente, primero se tomó el resultado de las pruebas hechas en varios tamaños y se insertó en hojas de cálculo para obtener un promedio y un coeficiente de variabilidad, como se muestra en el cuadro 4.1.1.1. El coeficiente de variabilidad funciona solo como un indicador para saber si el dispositivo estaba estable a la hora de hacer las pruebas.

	A	B	C
1	Sat Oct 31 00:11:12 CST 2015		
2	prueba quicksort c, numero de veces: 35, tipo de programa: 1, altura n	memoria	segundos
3	Sequential quicksort in c took: 0.002897 sec. and memory= 1808 KB	1808	0.002897
4	Sequential quicksort in c took: 0.002991 sec. and memory= 1808 KB	1808	0.002991
5	Sequential quicksort in c took: 0.002830 sec. and memory= 1808 KB	1808	0.00283
6	Sequential quicksort in c took: 0.002815 sec. and memory= 1808 KB	1808	0.002815
•			
•			
•			
34	Sequential quicksort in c took: 0.002950 sec. and memory= 1808 KB	1808	0.00295
35	Sequential quicksort in c took: 0.002826 sec. and memory= 1808 KB	1808	0.002826
36	Sequential quicksort in c took: 0.002829 sec. and memory= 1808 KB	1808	0.002829
37	Sequential quicksort in c took: 0.002789 sec. and memory= 1808 KB	1808	0.002789
38			
39	media	1808	0.002829
40	coeficiente de variabilidad	0	1.734706

10000 | 20000 | 30000 | 40000 | 50000 | 60000 | 70000 | 80000 | 90000 | 100000

Cuadro No. 4.1.1.1 Formato de Archivo de Pruebas Secuenciales en Computadora 1

En el cuadro 4.1.1.1 se muestra la estructura del archivo con varias hojas de cálculo usado para las pruebas, en este caso las pruebas son para el Quicksort secuencial (usando un solo hilo de ejecución) hechas en la PC (Computadora 1). Todas las pruebas que implican solo una computadora se hicieron usado la Computadora 1, solo en las pruebas que implican dos computadoras se adiciona la Computadora 2. Como se puede apreciar en el cuadro

4.1.1.1, hay 35 pruebas para la hoja con nombre “10000”, el cuadro muestra 2 filas al inicio de la hoja de cálculo que son una descripción de la hora y los parámetros de la prueba respectivamente. Esto se repite en todas las hojas, el nombre de la hoja indica que las pruebas fueron hechas para 10 mil elementos, como se puede apreciar hay 10 hojas que son las que tienen pruebas del algoritmo para 10 distintas cantidades de datos en una plataforma dada, ya sea un smartphone o una PC. Siempre hay 35 pruebas en cada hoja, se puede apreciar que las pruebas en el documento indicado en el cuadro 4.1.1.1 van de un tamaño de 10mil elementos en el arreglo hasta 100mil elementos. Para cada uno de los tamaños se obtiene el promedio y un coeficiente de variabilidad (como se observa en el cuadro 4.1.1.1).

Se tienen varios archivos con la estructura indicada anteriormente (cada archivo tiene 350 pruebas, 35 por cada uno de los 10 tamaños de prueba) por cada ambiente de pruebas, como se detalla en el cuadro 4.1.1.2. Cuando en el cuadro se indica que una prueba se hace para smartphone básico o para smartphone avanzado, lo que se da a entender es que el tamaño de las pruebas es basado en los tamaños escogidos para uno u otro de esos dispositivos.

Dispositivos en la Prueba	**Lenguaje**	**Archivo de Pruebas Secuencial**	**Archivo de Pruebas Paralelo**	**Archivo de Pruebas Distribuido**
Smartphones	C	Existe un archivo para Quicksort, otro para multiplicación de matrices y otro para BFS. Este juego de archivos se genera para cada uno de los smartphones ya que los tamaños de las pruebas eran diferentes en cada smartphone. Total=6 archivos	Existe un archivo para Quicksort, otro para multiplicación de matrices y otro para BFS. Este juego de archivos se genera para cada uno de los smartphones ya que los tamaños de las pruebas eran diferentes en cada smartphone. Total=6 archivos	No se genera

Smartphones	Erlang	Existe un archivo para Quicksort, otro para multiplicación de matrices y otro para BFS. Este juego de archivos se genera para cada uno de los smartphones ya que los tamaños de las pruebas eran diferentes en cada smartphone. Total=6 archivos	Existe un archivo para Quicksort, otro para multiplicación de matrices y otro para BFS. Este juego de archivos se genera para cada uno de los smartphones ya que los tamaños de las pruebas eran diferentes en cada smartphone. Total=6 archivos	No se genera
Computadora 1	C	Existe un archivo para Quicksort, otro para multiplicación de matrices y otro para BFS. Este juego de archivos se genera para cada uno de los smartphones ya que los tamaños de las pruebas eran diferentes en cada smartphone. Total=6 archivos	Existe un archivo para Quicksort, otro para multiplicación de matrices y otro para BFS. Este juego de archivos se genera para cada uno de los smartphones ya que los tamaños de las pruebas eran diferentes en cada smartphone. Total=6 archivos	No se genera
Computadora 1	Erlang	Existe un archivo para Quicksort, otro para Multiplicación de matrices y otro para BFS. Este juego de archivos se genera para cada uno de los smartphones ya que los tamaños de las pruebas eran diferentes en cada smartphone. Para las pruebas a compararse con el smartphone básico se compila en Erlang R12B-5, para las pruebas a compararse con el smartphone avanzado se compila con Erlang 18.1. Total=6 archivos	Existe un archivo para Quicksort, otro para Multiplicación de matrices y otro para BFS. Este juego de archivos se genera para cada uno de los smartphones ya que los tamaños de las pruebas eran diferentes en cada smartphone. Para las pruebas a compararse con el smartphone básico se compila en Erlang R12B-5, para las pruebas a compararse con el smartphone avanzado se compila con Erlang 18.1. Total=6 archivos	No se genera
Computadora 1 Vs Smartphone	C en los dos dispositivos	No se genera	No se genera	Existe un archivo como este para Quicksort solamente ya que la prueba distribuida no fue construida para BFS o multiplicación de matrices.

				Esto solo para el smartphone básico ya que para el más avanzado no se hizo esta prueba. Total=1 archivo.
Computadora 1 Vs Smartphone	Erlang en los dos dispositivos	No se genera	No se genera	Existe un archivo como este para Quicksort solamente ya que la prueba distribuida no fue construida para BFS o multiplicación de matrices. Esto solo para el smartphone básico ya que para el más avanzado no se hizo esta prueba. Se hace con Erlang R12B-5 que es el que corre el dispositivo básico. Total=1 archivo.
Computadora 1 Vs Computadora 2	C en los dos dispositivos	No se genera	No se genera	Existe un archivo como este para Quicksort solamente ya que la prueba distribuida no fue construida para BFS o multiplicación de matrices. Esto solo para el smartphone básico ya que para el más avanzado no se hizo esta prueba. Total=1 archivo.
Computadora 1 Vs Computadora 2	Erlang en los dos dispositivos	No se genera	No se genera	Existe un archivo como este para Quicksort solamente ya

				que la prueba distribuida no fue construida para BFS o multiplicación de matrices. Esto solo para el smartphone básico ya que para el más avanzado no se hizo esta prueba. Se hace con Erlang R12B-5 que es el que corre el dispositivo básico. Total=1 archivo.

Cuadro No. 4.1.1.2 Archivos Básicos de Pruebas

4.1.2 Cuadro de pruebas porcentuales

Todos los datos generados en los archivos de prueba del cuadro 4.1.1.2 se exponen en otra hoja de cálculo que conforma un resumen de las pruebas y que consta de varias partes. Antes de presentar un ejemplo de un cuadro completo de resumen (más adelante en este capítulo aparece un ejemplo del cuadro completo) es conveniente explicar cada una de sus partes por separado. En la primera parte del archivo resumen se observan los resultados de las pruebas secuenciales, como se muestra en el cuadro 4.1.2.1.

Tipo de Prueba	Tamaño Prueba (tamaño de arreglo)	Prueba en PC			
		Pruebas C		Pruebas Erlang	
Prueba QuickSort Secuencial		Memoria en KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos
	10000	1808	0.002828686	61413.02857	0.0218714
	20000	1848	0.006027771	62727.31429	0.047158686
	30000	1888	0.0092818	64589.94286	0.071807286
	40000	2092	0.012742	65486.05714	0.1002008
	50000	2164	0.016221571	67483.65714	0.126176657
	60000	2244	0.019722486	69684.57143	0.144620371
	70000	2324	0.0234358	70244.22857	0.172063771
	80000	2404	0.027099457	72681.25714	0.171625571
	90000	2476	0.030686486	74993.37143	0.198917943
	100000	2556	0.034166114	74963.65714	0.21581

Cuadro No. 4.1.2.1 Archivo Resumen, Parte de Pruebas Secuenciales en Computadora 1

Aquí se puede apreciar donde se listan los distintos promedios para la prueba Quicksort secuencial, como se puede ver, este apartado es para las pruebas usando únicamente la PC. Para este apartado, a la derecha del los datos mostrados en el cuadro 4.1.2.1, hay 2 columnas que muestran el porcentaje de Erlang con respecto a C tanto en memoria como en tiempo de ejecución, como se muestra en el cuadro 4.1.2.2

lang		% Cambio de Erlang con Respecto a C	
KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos
2857	0.0218714	3396.74%	773.20%
1429	0.047158686	3394.34%	782.36%
4286	0.071807286	3421.08%	773.64%
5714	0.1002008	3130.31%	786.38%
5714	0.126176657	3118.47%	777.83%
7143	0.144620371	3105.37%	733.28%
2857	0.172063771	3022.56%	734.19%
5714	0.171625571	3023.35%	633.32%
7143	0.198917943	3028.81%	648.23%
5714	0.21581	2932.85%	631.65%

Cuadro No. 4.1.2.2 Archivo Resumen, Porcentaje de Cambio de Erlang con Respecto a C

En el cuadro 4.1.2.2 se muestra el cambio de Erlang con respecto a C, por ejemplo en la primera prueba se tiene que Erlang duró un 773.20% con respecto a C (casi 8 veces el tempo tomado por C) para terminar la prueba. Esto se obtiene de dividir el tiempo durado por Erlang entre el tiempo durado en C y luego multiplicando el resultado por 100 (es un

porcentaje). Del mismo modo se saca el porcentaje de memoria indicado en el cuadro 4.1.2.2, en donde para la primera fila se observa que Erlang requiere casi 34 veces la memoria que requirió C para resolver el mismo problema.

De manera similar a las pruebas hechas para la PC se tiene las mismas pruebas realizadas para el Smartphone, como se muestra en el cuadro 4.1.2.3.

		Prueba en PDA					
ambio de Erlang con Respecto a C		Pruebas C		Pruebas Erlang		% Cambio de Erlang cor Respecto a C	
oria en KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos	Memoria	Tiempo
2458.99%	757.71%	1708	0.10891141	11900.36364	1.368163182	696.74%	1256.22%
2484.39%	994.59%	1748	0.228636361	12813.25	3.070895406	733.02%	1343.14%
2508.23%	1136.00%	1788	0.354714671	13808.11429	4.653679114	772.27%	1311.95%
2344.79%	1126.67%	1992	0.480630687	15937.48571	6.546627771	800.07%	1362.09%
2354.84%	1199.06%	2064	0.60878665	17528.24242	8.497129667	849.24%	1395.75%
2348.55%	1244.32%	2144	0.74512813	19895.65714	10.35857974	927.97%	1390.17%
2321.02%	1280.79%	2224	0.87456173	20321.22222	12.04059522	913.72%	1376.76%
2323.96%	1336.88%	2304	1.009257664	21285.83784	13.96684278	923.86%	1383.87%
2266.86%	1348.22%	2376	1.143709543	21982.77778	15.88746303	925.20%	1389.12%
2283.06%	1377.18%	2456	1.278769187	22920.57143	17.79215129	933.25%	1391.35%

Cuadro No. 4.1.2.3 Archivo Resumen, Parte de Pruebas Secuenciales en Smartphone.

Ya en el extremo derecho del cuadro, en las columnas que le siguen al cuadro 4.1.2.3 se tiene dos columnas comparativas para mostrar si las pruebas hechas en la PC y el smartphone indican un buen desempeño o no del Smartphone, como se muestra en el cuadro 4.1.2.4. Si aquí se muestra un valor mayor al 100% los resultados fueron beneficiosos para el smartphone, de lo contrario no.

			% de Eficiencia de la SmartP con Respecto a la Computadora	
	% Cambio de Erlang con Respecto a C			
ıundos	Memoria	Tiempo	Memoria	Tiempo
7114543	695.37%	1246.51%	488.48%	62.03%
7621086	734.04%	1342.49%	462.42%	58.28%
3679114	772.27%	1313.07%	442.99%	58.92%
3627771	800.07%	1368.54%	391.25%	57.46%
7758857	845.35%	1398.88%	368.89%	55.60%
5857974	927.97%	1379.67%	334.64%	53.15%
4941654	915.10%	1370.81%	330.30%	53.56%
9167537	929.86%	1374.31%	325.14%	46.08%
9929391	925.50%	1394.28%	327.26%	46.49%
3722937	978.17%	1395.87%	299.83%	45.25%

Cuadro No. 4.1.2.4 Archivo Resumen, Columnas Comparativas.

Por ejemplo en la primera fila del cuadro 4.1.1.4 se tiene que, en cuanto a memoria se refiere, Erlang resultó con un desempeño de un 488.48% mejor en el smartphone que en la PC pero solo un 62.03% del desempeño fue logrado en cuanto al tiempo de ejecución. Por consiguiente, fue mejor el smartphone en el manejo de memoria pero peor en la eficiencia de ejecución. Esto para la prueba secuencial con un tamaño de 10 mil elementos, según se puede apreciar en el resumen completo expuesto en los cuadros 4.1.2.8 y 4.1.2.9.

Más abajo en la parte izquierda del cuadro se puede observar una estructura similar para el resto de las pruebas, como se muestra en el cuadro 4.1.2.5.

Prueba QuickSort Paralelo	40	216862.7429	0.001376	59071.77143	0.000355229
	80	427967.2	0.002599286	59152.22857	0.000590886
	120	612928.5714	0.003807857	59108.34286	0.000852229
	160	821897.2571	0.005077886	59122.97143	0.001216286
	200	961404.8	0.006249114	59079.08571	0.001407086
	240	1141043.429	0.007352829	59101.02857	0.001640114
	280	1389089.486	0.009242571	59115.65714	0.002203143
	320	1519142.4	0.010106657	59130.28571	0.002197
	360	1712216.686	0.0109482	59057.14286	0.0029048
	400	1931049.829	0.013830057	59020.57143	0.002851543
		Prueba Distribuida PC Vs PC			
		Pruebas C		Pruebas Erlang	
		Memoria en KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos
Prueba QuickSort Distribuido	40	144137.7143	0.007664657	42659.88571	0.032225771
	80	266573.0286	0.013370229	42703.77143	0.063549686
	120	358309.2571	0.0192334	42440.45714	0.103645743
	160	452212.4571	0.024455943	42440.45714	0.127534457
	200	550004.3636	0.030905394	42498.97143	0.1594142
	240	602754.8571	0.0348598	42498.97143	0.190918457
	280	677514.4	0.041335171	42557.48571	0.222683543
	320	804259.6571	0.046537657	42455.08571	0.2556302
	360	868188.3429	0.051997229	42455.08571	0.2881562
	400	918110.1818	0.058185121	42513.6	0.320143629

Cuadro No. 4.1.2.5 Archivo resumen, Prueba Paralela en Computadora 1 y Distribuida con las dos PC.

Para las pruebas distribuidas hay un título que indica si las pruebas son entre las dos PC o entre la PC (Computadora 1) y el Smartphone, como se muestra en el cuadro 4.1.2.6.

Prueba Distribuida PC Vs PC						**Prueba Distribuida SmartP Vs PC**		
Pruebas C		Pruebas Erlang		% Cambio de Erlang con Respecto a C		Pruebas C		Pruebas Erlang
Memoria en KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos	Memoria en KB
144137.7143	0.007664657	42659.88571	0.032225771	29.60%	420.45%	139903.3143	0.3811128	9748
266573.0286	0.013370229	42703.77143	0.063549686	16.02%	475.31%	236480.6857	1.099489914	9748
358309.2571	0.0192334	42440.45714	0.103645743	11.84%	538.88%	316675.3143	2.031463543	9748
452212.4571	0.024455943	42440.45714	0.127534457	9.39%	521.49%	405529.6	3.191675486	9748
550004.3636	0.030905394	42498.97143	0.1594142	7.73%	515.81%	460152.3429	4.735079286	9748
602754.8571	0.0348598	42498.97143	0.190918457	7.05%	547.68%	547923.5429	6.677762257	9748
677514.4	0.041335171	42557.48571	0.222683543	6.28%	538.73%	639849.6	9.135553914	9748
804259.6571	0.046537657	42455.08571	0.2556302	5.28%	549.30%	730195.5429	11.83294637	9748
868188.3429	0.051997229	42455.08571	0.2881562	4.89%	554.18%	772618.2857	13.73950117	9748
918110.1818	0.058185121	42513.6	0.320143629	4.63%	550.22%	844883.5429	16.69549677	9748

Cuadro No. 4.1.2.6 Archivo Resumen, Tipos de Pruebas Distribuidas.

Estas pruebas distribuidas también cuentan con las columnas de análisis en la parte extrema derecha, como se muestra en el cuadro 4.1.2.7. Estas columnas vienen a indicar si el smartphone trabajando de manera distribuida tiene o no un nivel aceptable con respecto a la misma prueba distribuida entre las dos PC. Si muestran un valor mayor al 100% los resultados fueron beneficiosos para el smartphone, de lo contrario no.

a SmartP Vs PC						
	Pruebas Erlang		% Cambio de Erlang con Respecto a C			
o en Segundos	Memoria en KB	Tiempo en Segundos	Memoria	Tiempo	Memoria	Tiempo
0.3811128	9748	0.723835429	6.97%	189.93%	424.77%	221.37%
1.099489914	9748	1.325451571	4.12%	120.55%	388.62%	394.28%
2.031463543	9748	1.869708371	3.08%	92.04%	384.79%	585.50%
3.191675486	9748	2.438793086	2.40%	76.41%	390.43%	682.48%
4.735079286	9748	2.945862257	2.12%	62.21%	364.75%	829.10%
6.677762257	9748	3.467720686	1.78%	51.93%	396.32%	1054.65%
9.135553914	9748	3.995242343	1.52%	43.73%	412.31%	1231.86%
11.83294637	9748	4.415998943	1.33%	37.32%	395.42%	1471.88%
13.73950117	9748	4.981458514	1.26%	36.26%	387.58%	1528.49%
16.69549677	9748	5.4448502	1.15%	32.61%	401.34%	1687.12%

Cuadro No. 4.1.2.7 Archivo Resumen, Análisis a la Extrema Derecha de Pruebas Distribuidas.

Los cuadros 4.1.2.8 y 4.1.2.9 muestran la hoja de cálculo que constituye el ejemplo de resumen completo de las pruebas para el Quicksort basado en el smartphone básico. Hay varios archivos de estos que al final, con archivos de pruebas estadísticas que se explican más adelante, darán como producto los gráficos que muestran los resultados finales de las pruebas de esta investigación.

Tipo de Prueba	Tamaño Prueba (tamaño de arreglo)	Prueba en PC					
		Pruebas C		Pruebas Erlang		% Cambio de Erlang con Respecto a C	
Prueba QuickSort Secuencial		Memoria en KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos
	10000	1808	0.002828686	61413.02857	0.0218714	3396.74%	773.20%
	20000	1848	0.006027771	62727.31429	0.047158686	3394.34%	782.36%
	30000	1888	0.0092818	64589.94286	0.071807286	3421.08%	773.64%
	40000	2092	0.012742	65486.05714	0.1002008	3130.31%	786.38%
	50000	2164	0.016221571	67483.65714	0.126176657	3118.47%	777.83%
	60000	2244	0.019722486	69684.57143	0.144620371	3105.37%	733.28%
	70000	2324	0.0234358	70244.22857	0.172063771	3022.56%	734.19%
	80000	2404	0.027099457	72681.25714	0.171625571	3023.35%	633.32%
	90000	2476	0.030686486	74993.37143	0.198917943	3028.81%	648.23%
	100000	2556	0.034166114	74963.65714	0.21581	2932.85%	631.65%
Prueba QuickSort Paralelo	40	216862.7429	0.001376	59071.77143	0.000355229	27.24%	25.82%
	80	427967.2	0.002599286	59152.22857	0.000590886	13.82%	22.73%
	120	612928.5714	0.003807857	59108.34286	0.000852229	9.64%	22.38%
	160	821897.2571	0.005077886	59122.97143	0.001216286	7.19%	23.95%
	200	961404.8	0.006249114	59079.08571	0.001407086	6.15%	22.52%
	240	1141043.429	0.007352829	59101.02857	0.001640114	5.18%	22.31%
	280	1389089.486	0.009242571	59115.65714	0.002203143	4.26%	23.84%
	320	1519142.4	0.010106657	59130.28571	0.002197	3.89%	21.74%
	360	1712216.686	0.0109482	59057.14286	0.0029048	3.45%	26.53%
	400	1931049.829	0.013830057	59020.57143	0.002851543	3.06%	20.62%
		Prueba Distribuida PC Vs PC					
		Pruebas C		Pruebas Erlang		% Cambio de Erlang con Respecto a C	
Prueba QuickSort Distribuido	40	Memoria en KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos
	80	144137.7143	0.007664657	42659.88571	0.032225771	29.60%	420.45%
	120	266573.0286	0.013370229	42703.77143	0.063549686	16.02%	475.31%
	160	358309.2571	0.0192334	42440.45714	0.103645743	11.84%	538.88%
	200	452212.4571	0.024455943	42440.45714	0.127534457	9.39%	521.49%
	240	550004.3636	0.030905394	42498.97143	0.1594142	7.73%	515.81%
	280	602754.8571	0.0348598	42498.97143	0.190918457	7.05%	547.68%
	320	677514.4	0.041335171	42557.48571	0.222683543	6.28%	538.73%
	360	804259.6571	0.046537657	42455.08571	0.2556302	5.28%	549.30%
	400	868188.3429	0.051997229	42455.08571	0.2881562	4.89%	554.18%
		918110.1818	0.058185121	42513.6	0.320143629	4.63%	550.22%

Cuadro No. 4.1.2.8 Archivo Resumen, Primera Parte.

Tipo de Prueba	Tamaño Prueba (tamaño de arreglo)	Prueba en SmartP						% de Eficiencia de la SmartP con Respecto a la Computadora	
		Pruebas C		Pruebas Erlang		% Cambio de Erlang con Respecto a C			
Prueba QuickSort Secuencial		Memoria en KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos	Memoria	Tiempo	Memoria	Tiempo
	10000	1708	0.1096756	11876.91429	1.367114543	695.37%	1246.51%	488.48%	62.03%
	20000	1748	0.2292466	12830.97143	3.077621086	734.04%	1342.49%	462.42%	58.28%
	30000	1788	0.354411257	13808.11429	4.653679114	772.27%	1313.07%	442.99%	58.92%
	40000	1992	0.478366086	15937.48571	6.546627771	800.07%	1368.54%	391.25%	57.46%
	50000	2064	0.606753818	17448.11429	8.487758857	845.35%	1398.88%	368.89%	55.60%
	60000	2144	0.750800686	19895.65714	10.35857974	927.97%	1379.67%	334.64%	53.15%
	70000	2224	0.8790008	20351.77143	12.04941654	915.10%	1370.81%	330.30%	53.56%
	80000	2304	1.018085114	21424	13.99167537	929.86%	1374.31%	325.14%	46.08%
	90000	2376	1.140323886	21989.94286	15.89929391	925.50%	1394.28%	327.26%	46.49%
	100000	2456	1.281436848	24023.88571	17.88722937	978.17%	1395.87%	299.83%	45.25%
Prueba QuickSort Paralelo	40	184546.6286	0.448746943	9748	0.015642171	5.28%	3.49%	515.69%	740.62%
	80	353360.3429	0.948425057	9748	0.036000257	2.76%	3.80%	501.03%	598.89%
	120	493853.1429	1.434899057	9748	0.054872457	1.97%	3.82%	488.56%	585.25%
	160	634082.6286	1.859479714	9748	0.073234886	1.54%	3.94%	467.92%	608.17%
	200	780949.9394	2.471626576	9748	0.0946818	1.25%	3.83%	492.31%	587.78%
	240	908894.9714	3.070856429	9748	0.115703971	1.07%	3.77%	482.94%	592.01%
	280	1042980.457	3.606515286	9748	0.136737143	0.93%	3.79%	455.34%	628.71%
	320	1163285.829	4.251014571	9748	0.158691371	0.84%	3.73%	464.50%	582.32%
	360	1330900	4.954903086	9748	0.178601457	0.73%	3.60%	470.92%	736.08%
	400	1486392.848	5.514214818	9748	0.201056886	0.66%	3.65%	466.05%	565.48%
		Prueba Distribuida SmartP Vs PC							
		Pruebas C		Pruebas Erlang		% Cambio de Erlang con Respecto a C			
Prueba QuickSort Distribuido	40	Memoria en KB	Tiempo en Segundos	Memoria en KB	Tiempo en Segundos	Memoria	Tiempo	Memoria	Tiempo
	80	139903.3143	0.3811128	9748	0.723835429	6.97%	189.93%	424.77%	221.37%
	120	236480.6857	1.099489914	9748	1.325451571	4.12%	120.55%	388.62%	394.28%
	160	316675.3143	2.031463543	9748	1.869708371	3.08%	92.04%	384.79%	585.50%
	200	405529.6	3.191675486	9748	2.438793086	2.40%	76.41%	390.43%	682.48%
	240	460152.3429	4.735079286	9748	2.945862257	2.12%	62.21%	364.75%	829.10%
	280	547923.5429	6.677762257	9748	3.467720686	1.78%	51.93%	396.32%	1054.65%
	320	639849.6	9.135553914	9748	3.995242343	1.52%	43.73%	412.31%	1231.86%
	360	730195.5429	11.83294637	9748	4.415998943	1.33%	37.32%	395.42%	1471.88%
	400	772618.2857	13.73950117	9748	4.981458514	1.26%	36.26%	387.58%	1528.49%
		844883.5429	16.69549677	9748	5.4448502	1.15%	32.61%	401.34%	1687.12%

Cuadro No. 4.1.2.9 Archivo Resumen, Segunda Parte.

4.1.3 Pruebas estadísticas de contraste de hipótesis y archivos finales

Además de las pruebas porcentuales descritas anteriormente, en donde se tomó la media de los resultados obtenidos para cada tamaño de prueba en los distintos escenarios, también se analizan los datos usando la técnica estadística de contraste de hipótesis para saber si la media que indica el rendimiento de Erlang con respecto a C es mejor en el smartphone que en la PC. Para eso se crearon archivos que cargan los datos de las 2 plataformas a comparar, usando como fuente los datos de cada uno de los archivos explicados en el apartado 4.1.1.

Cada uno de estos archivos contiene las pruebas de contraste de hipótesis para cada tamaño de prueba. En cada hoja se hace una prueba estadística de dos poblaciones grandes con igual número de elementos para saber si las medias de esas poblaciones son diferentes o no. El Alpha usado en la prueba es de 0.05, el tamaño del muestreo es más de 35

elementos por prueba, por lo que se usó la distribución de probabilidad normal como parámetro en la prueba. En cada una de las hojas aparece un apartado como el expuesto en el cuadro 4.1.3.1. No es necesario saber si los datos origen están normalmente distribuidos, ya que como se indicó en el capítulo 3 en el apartado de inferencia estadística, en estos experimentos se puede usar la distribución de probabilidad normal sin importar la distribución de la población origen debido a los cuidados que se tuvieron para recopilar las muestras y al número de datos en cada experimento.

En este cuadro se indica si se detecta que las medias son iguales. Las medias que se comparan en ambas plataformas son: 1) Aprovechamiento observado en el tiempo de duración que se obtiene al resolver un programa en Erlang con respecto al tiempo utilizado en C, 2) Aprovechamiento observado en el uso de memoria que se obtiene al resolver un programa en Erlang con respecto a la memoria utilizada en C.

C	Prueba estadística	
	Rubro de la prueba	Valor
3%	Alpha escogido para las pruebas	0.05
9%	**Memoria (Número de veces de Erlang con respecto a C)**	
7%	¿Aprovechamiento es diferente?	Si
0%	¿Mejor en caso Pc (Primer muestreo comparado)?	No
3%	¿Mejor en caso SmartP (Segundo muestreo comparado)?	Si
0%	Conteo de elementos de la muestra	35
4%	Media Pc	2932.85%
2%	Varianza Pc	1.810336126
7%	Media SmartP	978.17%
8%	Varianza SmartP	0.930112456
6%	z obtenido	69.85519082
7%	Valor p obtenido (dos colas, o p*2)	0.000000
8%	z crítico 2 colas	1.959963985
3%		
5%	**Procesador (Número de veces de Erlang con respecto a C)**	
7%	¿Aprovechamiento es diferente?	Si
8%	¿Mejor en caso Pc (Primer muestreo comparado)?	Si
2%	¿Mejor en caso SmartP (Segundo muestreo comparado)?	No
6%	Conteo de elementos de la muestra	35
5%	Media Pc	632.49%
8%	Varianza Pc	0.318802439
1%	Media SmartP	1396.44%
2%	Varianza SmartP	0.707352237
0%	z obtenido	-44.61599397
7%	Valor p obtenido (dos colas, o p*2)	0.000000
4%	z crítico 2 colas	1.959963985

Cuadro No. 4.1.3.1 Archivo con Pruebas de Contraste de Hipótesis.

En los archivos descritos anteriormente en el apartado 4.1.2, se tiene una hoja de cálculo por aparte que resume los resultados de la prueba estadística indicada, esta hoja resume la información estadística que es una de las bases junto con los datos explicados en el apartado 4.1.2 para generar los gráficos finales. Un ejemplo de este resumen de la prueba estadística es el generado para Quicksort basado en el smartphone básico, el cual se muestra en los cuadros 4.1.3.2 y 4.1.3.3. Allí se aprecia el resumen de pruebas estadísticas tanto de la información de uso de la memoria como de la información de uso del procesador. Este resumen estadístico junto con el cuadro explicado en el apartado 4.1.1 son la información fuente para generar los gráficos finales que también acompañan el archivo final de pruebas.

Tipo de Prueba	Tamaño Prueba (tamaño de arreglo)	Prueba estadística de contraste de hipótesis					
		Tiempo se Segundos				Memoria	
Prueba QuickSort Secuencial		SmartP diferente a PC(H0:eficiencia en SmartP = eficiencia en PC, H1= eficiencia SmartP diferente a eficiencia PC)	p-valor	SmartP mejor a PC (deacuerdo a prueba de hipótesis anterior y signo de resultado de estadístico de prueba)	PC mejor que SmartP (deacuerdo a prueba de hipótesis anterior y signo de resultado de estadístico de prueba)	SmartP diferente a PC(H0:eficiencia en Smartp = eficiencia en PC, H1= eficiencia SmartP diferente a eficiencia PC)	p-valor
	10000	Si	0.000000	No	Si	Si	0.000000
	20000	Si	0.000000	No	Si	Si	0.000000
	30000	Si	0.000000	No	Si	Si	0.000000
	40000	Si	0.000000	No	Si	Si	0.000000
	50000	Si	0.000000	No	Si	Si	0.000000
	60000	Si	0.000000	No	Si	Si	0.000000
	70000	Si	0.000000	No	Si	Si	0.000000
	80000	Si	0.000000	No	Si	Si	0.000000
	90000	Si	0.000000	No	Si	Si	0.000000
	100000	Si	0.000000	No	Si	Si	0.000000
Prueba QuickSort Paralelo							
	40	Si	0.000000	Si	No	Si	0.000000
	80	Si	0.000000	Si	No	Si	0.000000
	120	Si	0.000000	Si	No	Si	0.000000
	160	Si	0.000000	Si	No	Si	0.000000
	200	Si	0.000000	Si	No	Si	0.000000
	240	Si	0.000000	Si	No	Si	0.000000
	280	Si	0.000000	Si	No	Si	0.000000
	320	Si	0.000000	Si	No	Si	0.000000
	360	Si	0.000000	Si	No	Si	0.000000
	400	Si	0.000000	Si	No	Si	0.000000
Prueba QuickSort Distribuido		PC Vs SmartP diferente a PC Vs PC(H0:eficiencia en PC Vs SmartP = eficiencia en PC Vs PC, H1=eficiencia PC Vs SmartP diferente a eficiencia PC)	p-valor	PC Vs SmartP mejor a PC Vs PC (deacuerdo a prueba de hipótesis anterior y signo de resultado de estadístico de prueba)	PC Vs PC mejor que PC Vs SmartP (deacuerdo a prueba de hipótesis anterior y signo de resultado de estadístico de prueba)	PC Vs SmartP diferente a PC Vs PC(H0:eficiencia en PC Vs SmartP = eficiencia en PC Vs PC, H1=eficiencia PC Vs SmartP diferente a eficiencia PC Vs PC)	p-valor
	40	Si	0.000000	Si	No	Si	0.000000
	80	Si	0.000000	Si	No	Si	0.000000
	120	Si	0.000000	Si	No	Si	0.000000
	160	Si	0.000000	Si	No	Si	0.000000
	200	Si	0.000000	Si	No	Si	0.000000
	240	Si	0.000000	Si	No	Si	0.000000
	280	Si	0.000000	Si	No	Si	0.000000
	320	Si	0.000000	Si	No	Si	0.000000
	360	Si	0.000000	Si	No	Si	0.000000
	400	Si	0.000000	Si	No	Si	0.000000

Cuadro No. 4.1.3.2 Resumen Pruebas de Contraste de Hipótesis para la Media de Velocidad Quicksort Smartphone Básico.

Tipo de Prueba	Tamaño Prueba (tamaño de arreglo)					
		Memoria				
Prueba QuickSort Secuencial		SmartP diferente a PC(H0:eficiencia en Smartp = eficiencia en PC, H1=eficiencia SmartP diferente a eficiencia PC)	p-valor	SmartP mejor a PC (deacuerdo a prueba de hipótesis anterior y signo de resultado de estadístico de prueba)	PC mejor queSmartP (deacuerdo a prueba de hipótesis anterior y signo de resultado de estadístico de prueba)	Alpha de las pruebas
	10000	Si	0.000000	Si	No	0.05
	20000	Si	0.000000	Si	No	0.05
	30000	Si	0.000000	Si	No	0.05
	40000	Si	0.000000	Si	No	0.05
	50000	Si	0.000000	Si	No	0.05
	60000	Si	0.000000	Si	No	0.05
	70000	Si	0.000000	Si	No	0.05
	80000	Si	0.000000	Si	No	0.05
	90000	Si	0.000000	Si	No	0.05
	100000	Si	0.000000	Si	No	0.05
Prueba QuickSort Paralelo	40	Si	0.000000	Si	No	0.05
	80	Si	0.000000	Si	No	0.05
	120	Si	0.000000	Si	No	0.05
	160	Si	0.000000	Si	No	0.05
	200	Si	0.000000	Si	No	0.05
	240	Si	0.000000	Si	No	0.05
	280	Si	0.000000	Si	No	0.05
	320	Si	0.000000	Si	No	0.05
	360	Si	0.000000	Si	No	0.05
	400	Si	0.000000	Si	No	0.05
Prueba QuickSort Distribuido		PC Vs SmartP diferente a PC Vs PC(H0:eficiencia en PC Vs SmartP = eficiencia en PC Vs PC, H1=eficiencia PC Vs SmartP diferente a eficiencia PC Vs PC)	p-valor	PC Vs SmartP mejor a PC Vs PC (deacuerdo a prueba de hipótesis anterior y signo de resultado de estadístico de prueba)	PC Vs PC mejor que PC Vs SmartP (deacuerdo a prueba de hipótesis anterior y signo de resultado de estadístico de prueba)	Alpha de las pruebas
	40	Si	0.000000	Si	No	0.05
	80	Si	0.000000	Si	No	0.05
	120	Si	0.000000	Si	No	0.05
	160	Si	0.000000	Si	No	0.05
	200	Si	0.000000	Si	No	0.05
	240	Si	0.000000	Si	No	0.05
	280	Si	0.000000	Si	No	0.05
	320	Si	0.000000	Si	No	0.05
	360	Si	0.000000	Si	No	0.05
	400	Si	0.000000	Si	No	0.05

Cuadro No. 4.1.3.3 Resumen Pruebas de Contraste de Hipótesis para la Media de la Memoria Consumida.

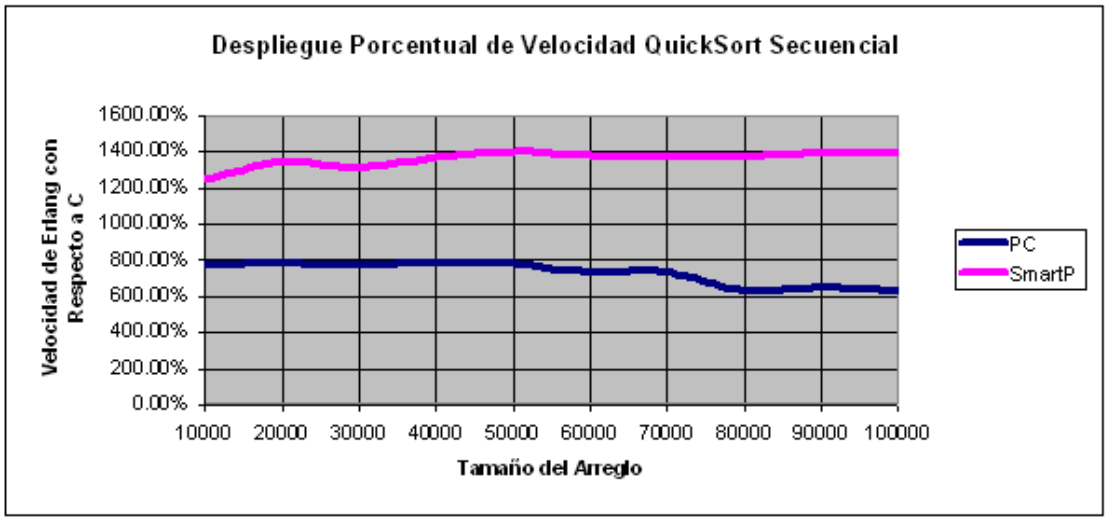
En el ejemplo mostrado en los cuadros 4.1.3.2 y 4.1.3.3 se aprecia valores de cero para la columna p-valor, esto es por que en las pruebas Quicksort en el smartphone básico vs la computadora se dan casos donde las medias son muy distintas en la prueba estadística, como se aprecia en las figuras analizadas en el apartado 4.2. Esto ocasiona que el error posible sea prácticamente nulo al tratar de probar que las medias son distintas y por eso el p-valor de cero.

Un archivo final de pruebas en resumen tendría 3 hojas principales, una que contiene un cuadro como el mostrado en el cuadro 4.1.2.8 y 4.1.2.9, otra hoja como la que se muestra en los cuadros 4.1.3.2 y 4.1.3.3 y una última hoja con los gráficos finales que resumen la información de las 2 primeras hojas. Estos gráficos serán explicados en el siguiente apartado.

4.2 Gráficos resultantes de primer caso y su interpretación

Con la información explicada anteriormente se construyen los gráficos de rendimiento en el smartphone con respecto a la PC. En este apartado se explican y muestran los gráficos obtenidos para las pruebas hechas con el algoritmo Quicksort en el smartphone básico, a manera de ejemplo, para que se puedan entender los demás gráficos mostrados en el apartado 4.3. Este algoritmo Quicksort trata de probar las plataformas con un programa que genere muchos hilos y poco procesamiento por cada hilo con un nivel de datos bajo por cada hilo.

En la figura 4.2.1 se puede ver la primera gráfica de las pruebas.

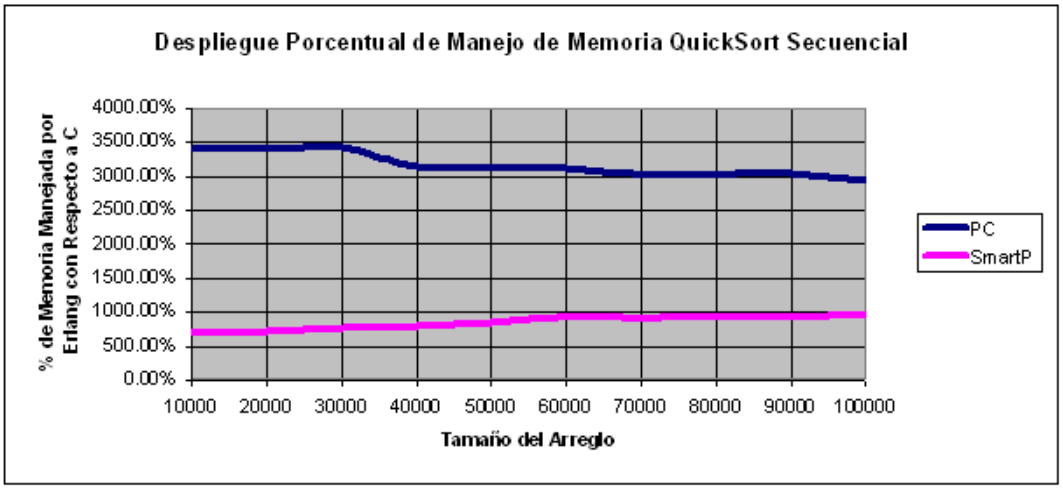


Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	No	No	No	No	No	No	No	No	No	No
PC Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
	10k	20k	30k	40k	50k	60k	70k	80k	80k	100k

Figura No. 4.2.1 Tendencia de Velocidad en Pruebas Secuenciales de Erlang con Respecto a C.

La figura 4.2.1 muestra una función de velocidad usando el porcentaje de cambio con respecto a C tanto para la PC como para el smartphone simple (indicado como SmartP en el gráfico) en la prueba secuencial. Como se puede ver de manera sencilla, cuanto más grande es la prueba mejor se desempeña Erlang con un solo hilo de ejecución en el PC. Las pruebas estadísticas coinciden con el despliegue de los gráficos en este como en el resto de las figuras que se aprecian a continuación. Para aclarar cómo se interpreta este gráfico, por ejemplo, para un tamaño de prueba de 10 mil elementos se tiene que Erlang dura ocho veces el tiempo que se dura en C solucionando el problema en la PC, sin embargo en el smartphone, Erlang dura un poco más de 12 veces el tiempo que se dura en C solucionando el problema.

Posteriormente un análisis similar es hecho en la gráfica 4.2.2 para la memoria usada.

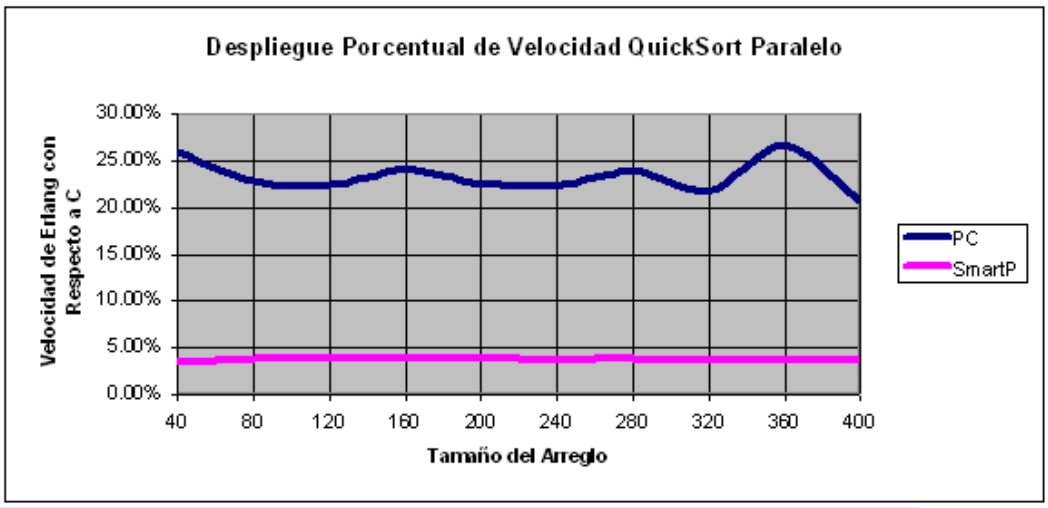


Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
PC Mejor	No	No	No	No	No	No	No	No	No	No
	10k	20k	30k	40k	50k	60k	70k	80k	80k	100k

Figura No. 4.2.2 Tendencia de Gasto de Memoria en Pruebas Secuenciales de Erlang con Respecto a C.

En la figura 4.2.2 se aprecia que Erlang ofrece una ventaja considerable en cuanto a manejo de memoria se refiere. Mientras en la PC, por ejemplo, para un tamaño de prueba de 10 mil elementos se tiene que Erlang gasta 35 veces la memoria que se gasta en C solucionando el problema en la PC, sin embargo en el smartphone, Erlang gasta como unas 6 o 7 veces la memoria que se gasta en C solucionando el problema. Se puede ver que para problemas grandes esta diferencia se empieza a recortar, pero aún así permanece muy pronunciada.

En la figura 4.2.3 se aprecia un análisis parecido al que se tiene en la figura 4.2.1.



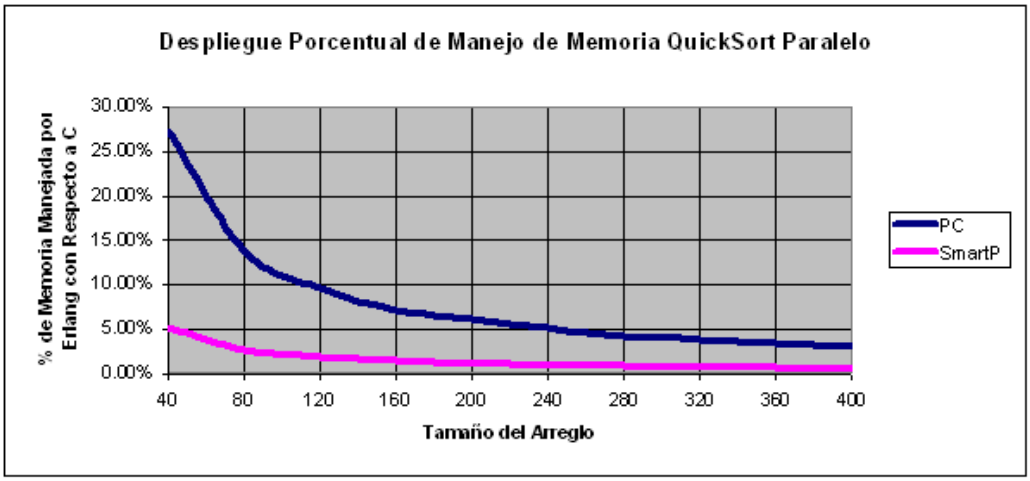
Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
PC Mejor	No	No	No	No	No	No	No	No	No	No
	40	80	120	160	200	240	280	320	360	400

Figura No. 4.2.3 Tendencia de Velocidad en Pruebas Paralelas de Erlang con Respecto a C.

En la figura 4.2.3 se puede apreciar que los tamaños de los arreglos son más pequeños, esto es debido a que para el caso paralelo y distribuido se abren 2 hilos cada vez que se desea ordenar los 2 subarreglos obtenidos en una iteración de Quicksort, como se indica en el capítulo 4, específicamente en la sección 4.1. Esto causa mucho consumo de memoria en la implementación en C del algoritmo, según pruebas hechas en el caso del smartphone básico bastaba con 400 elementos para abarcar la totalidad de la memoria disponible en el dispositivo. En la figura 4.2.3 se indica que Erlang es mucho más eficiente resolviendo este problema en el smartphone que en la PC, se puede observar que Erlang ofrece una ventaja considerable de velocidad en el smartphone sin importar el tamaño de la prueba e inclusive cuanto más grande el problema, la ventaja es levemente mayor. Por ejemplo, para un tamaño de prueba de 40 elementos se tiene que Erlang dura un poco más de una cuarta parte de lo que se dura en C solucionando el problema en la PC, mientras tanto en el

smartphone, Erlang dura menos de una veinteava parte de lo que dura C solucionando el problema.

En la figura 4.2.4 se aprecia un análisis parecido al que se tiene en la figura 4.2.2.

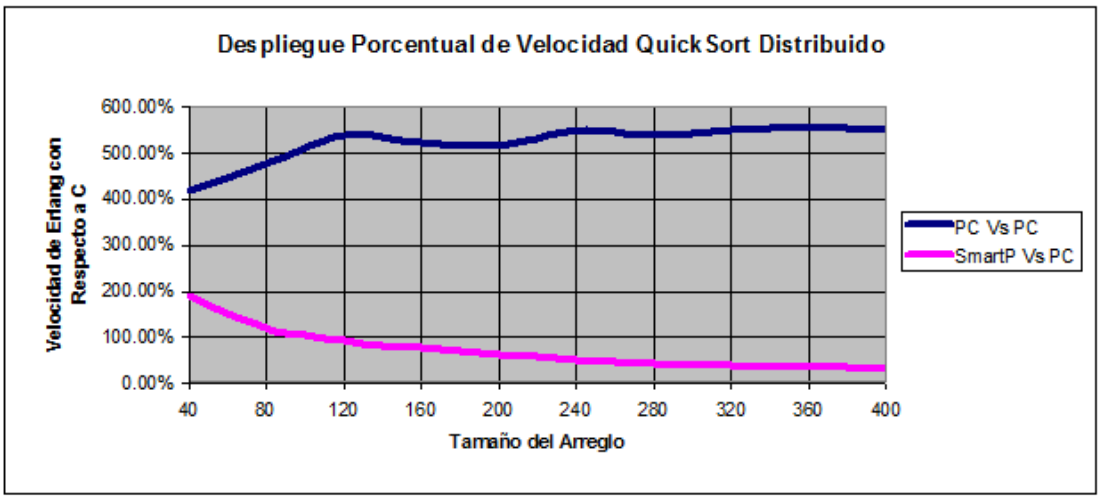


Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
PC Mejor	No	No	No	No	No	No	No	No	No	No
	40	80	120	160	200	240	280	320	360	400

Figura No. 4.2.4 Tendencia de Gasto de Memoria en Pruebas Paralelas de Erlang con Respecto a C.

En la figura 4.2.4 se muestra una brecha considerable que se mantiene para todos los tamaños de las pruebas, en donde el manejo de memoria es mejor en el smartphone que en la PC por parte de Erlang. Por ejemplo, para un tamaño de prueba de 40 elementos se tiene que Erlang gasta poco más de 25% de la memoria que se gasta en C solucionando el problema en la PC, mientras tanto en el smartphone, Erlang gasta un poco más de una veinteava parte de las memoria que se gasta en C solucionando el problema.

En la figura 4.2.5 se aprecia un análisis para el caso de las pruebas distribuidas parecido al que se tiene en la figura 4.2.1.

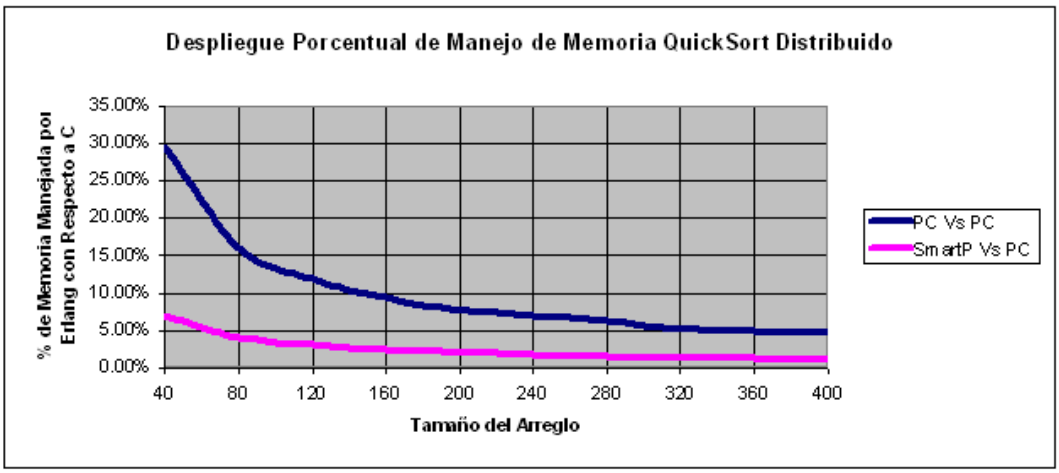


Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Vs PC Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
Pc VS Pc Mejor	No	No	No	No	No	No	No	No	No	No
	40	80	120	160	200	240	280	320	360	400

Figura No. 4.2.5 Tendencia de Velocidad en Pruebas Distribuidas de Erlang con Respecto a C.

En la figura 4.2.5 se puede apreciar que Erlang ofrece una ventaja considerable en la velocidad cuando se habla de procesos distribuidos usando el Smartphone, y entre más grande el problema mejor los resultados al compararse con lo que se obtiene de las pruebas en la PC, aún para problemas pequeños se tiene una ventaja considerable. Por ejemplo para un tamaño de prueba de 40 elementos se tiene que Erlang dura más de 4 veces lo que se dura en C solucionando el problema entre dos PC, mientras tanto en la versión distribuida entre el smartphone y la PC, Erlang dura 2 veces lo que dura C solucionando el problema.

En la figura 4.2.6 se aprecia un análisis para el caso de las pruebas distribuidas parecido al que se tiene en la figura 4.2.2.



Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
Pc Vs SmartP Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
Pc Vs PC Mejor	No	No	No	No	No	No	No	No	No	No
	40	80	120	160	200	240	280	320	360	400

Figura No. 4.2.6 Tendencia de Gasto de Memoria en Pruebas distribuidas de Erlang con Respecto a C.

En la figura 4.2.6 existe una brecha considerable que se mantiene para todos los tamaños de las pruebas, en donde el manejo de memoria es mejor en el smartphone que en la PC por parte de Erlang. Se tiene un caso muy similar al que se maneja para el caso paralelo. Por ejemplo, para un tamaño de prueba de 40 elementos se tiene que Erlang dura un poco más de una cuarta parte de lo que se dura en C solucionando el problema de manera distribuida entre dos PC, mientras tanto en la el smartphone dura un poco más de una veinteava parte de lo que dura C solucionando el problema.

4.3 Gráficos del resto de las pruebas

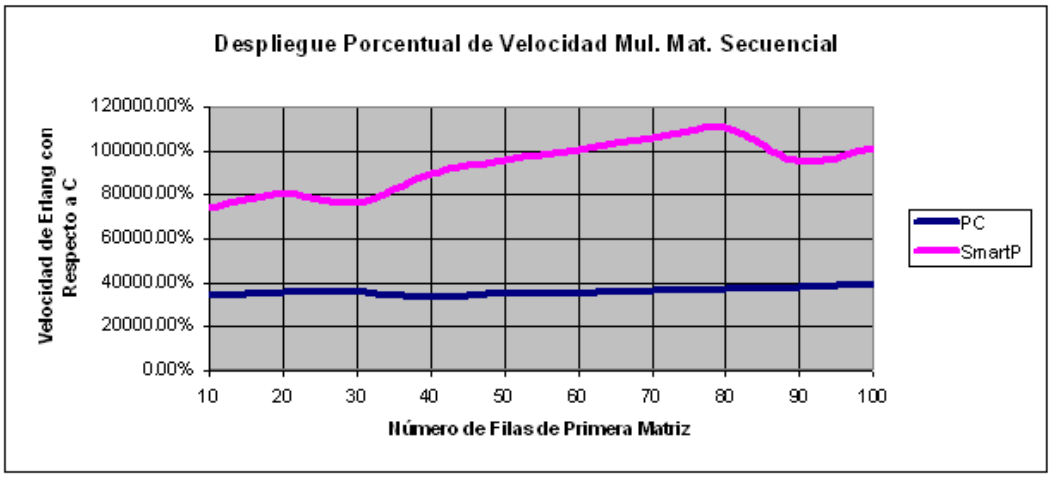
A continuación se indican los gráficos del resto de las pruebas, en las conclusiones se estarán tomando estos datos para generar deducciones acerca del comportamiento

mostrado del modelo de actores con respecto a C en los smartphones. Es importante aclarar que cuando en los gráficos se indica SmartP es el smartphone básico, y cuando se indica SmartP QC es el smartphone más complejo que usa 4 núcleos (Quadcore). Los gráficos explicados en la sección 4.2 son cuadros válidos resultantes de la investigación que no se mostrarán de nuevo en este apartado porque ya fueron mostrados y explicados como ayuda para entender los cuadros que aparecen en esta sección.

4.3.1 Pruebas usando el smartphone básico

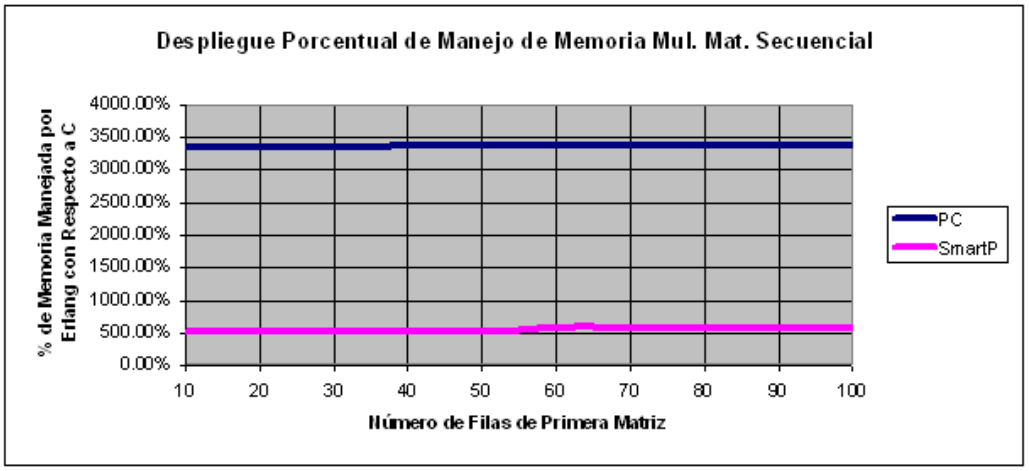
4.3.1.1 Multiplicación de matrices

Las matrices multiplicadas en este apartado para el algoritmo secuencial son (formato Filas * Columnas): la primera= N * 20 y la segunda = 20* 100, en donde N es variado dependiendo del tamaño de la prueba que muestre el gráfico. En el ambiente paralelo se ejecuta con los mismos tamaños de matrices a 4 hilos.



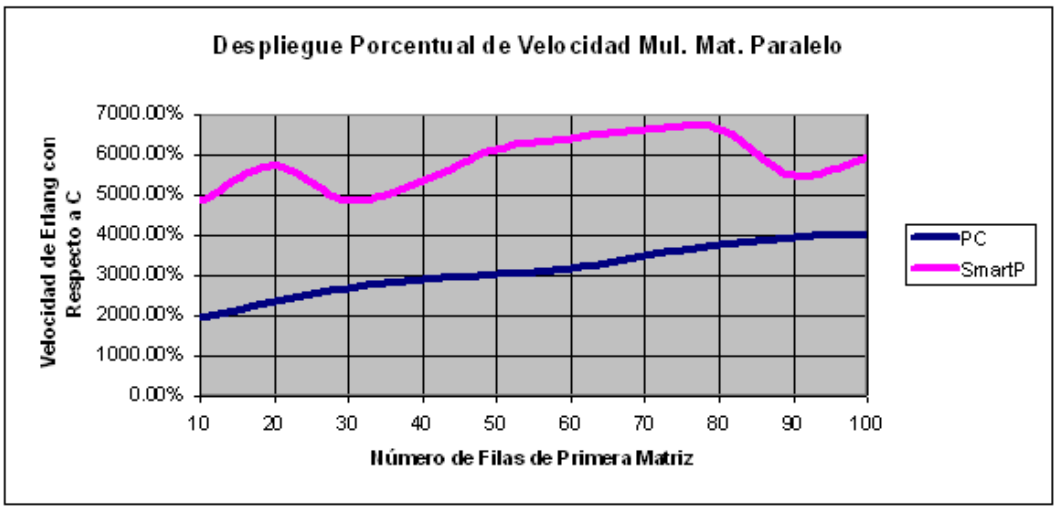
Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	No	No	No	No	No	No	No	No	No	No
PC Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
	10	20	30	40	50	60	70	80	90	100

Figura No. 4.3.1.1.1 Velocidad en Pruebas Secuenciales de Multiplicación de Matrices.



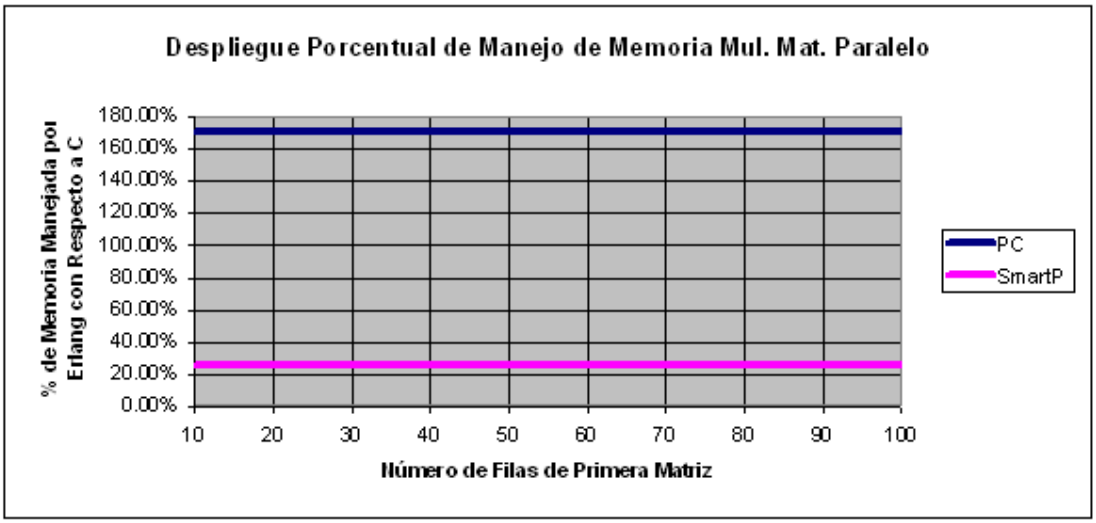
Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
PC Mejor	No	No	No	No	No	No	No	No	No	No
	10	20	30	40	50	60	70	80	90	100

Figura No. 4.3.1.1.2 Gasto de Memoria en Pruebas Secuenciales de Multiplicación de Matrices.



Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	No	No	No	No	No	No	No	No	No	No
PC Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
	10	20	30	40	50	60	70	80	90	100

Figura No. 4.3.1.1.3 Velocidad en Pruebas Paralelas de Multiplicación de Matrices.

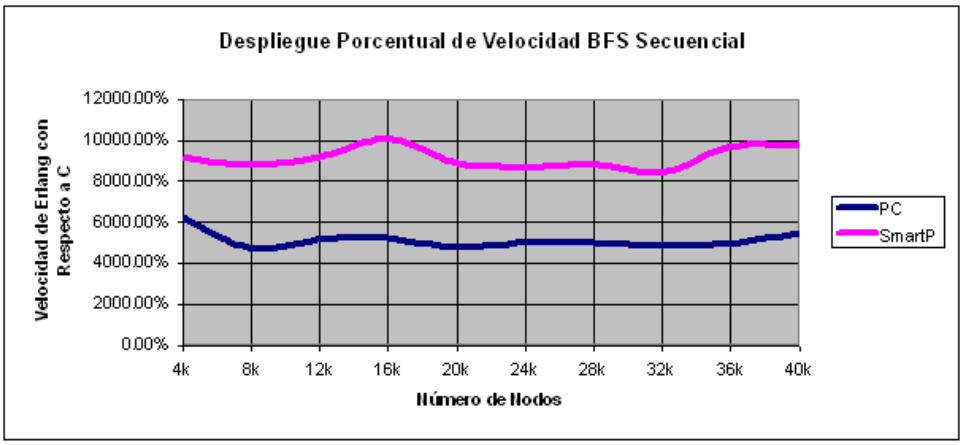


Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
PC Mejor	No	No	No	No	No	No	No	No	No	No
	10	20	30	40	50	60	70	80	90	100

Figura No. 4.3.1.1.4 Gasto de Memoria en Pruebas Paralelas de Multiplicación de Matrices.

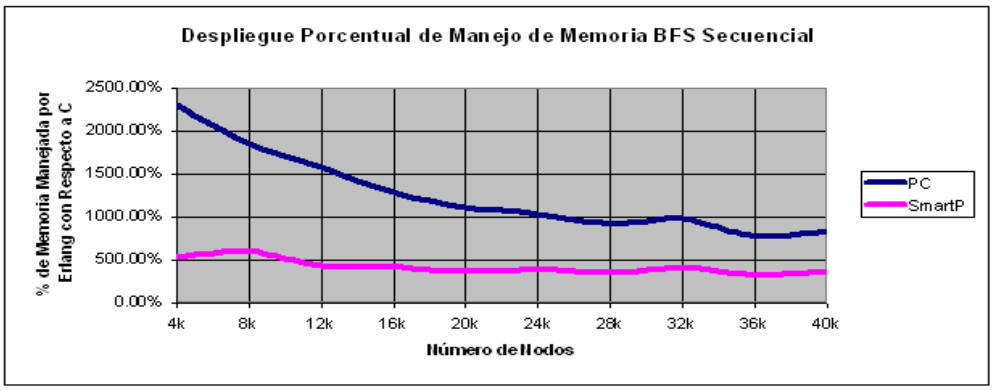
4.3.1.2 Búsqueda en anchura (BFS)

Los grafos tienen un tamaño en nodos como lo especifica el gráfico; los grafos fueron construidos con un promedio de arcos de 4 por nodo, estos arcos apuntan a nodos escogidos de manera aleatoria.



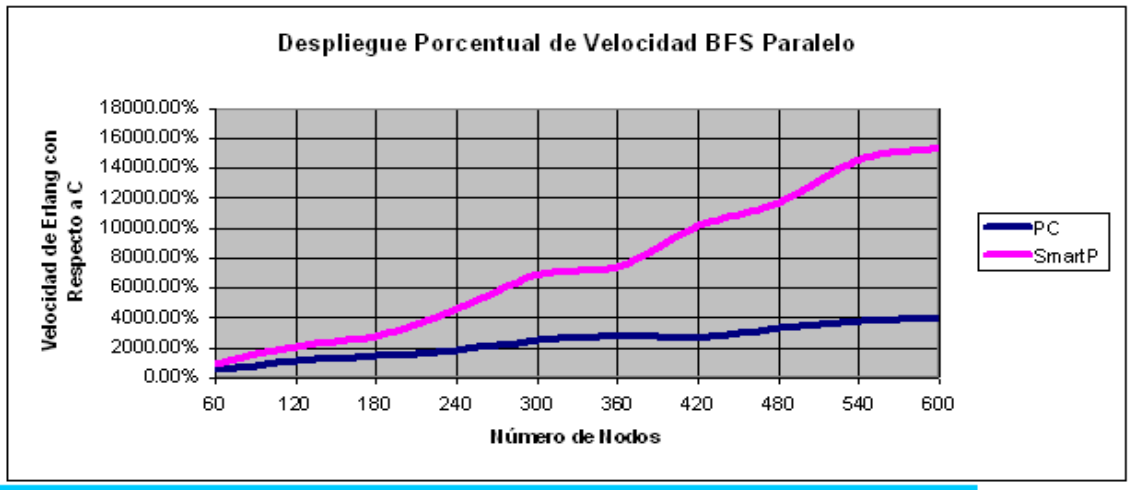
Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	No	No	No	No	No	No	No	No	No	No
PC Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
	4k	8k	12k	16k	20k	24k	28k	32k	36k	40k

Figura No. 4.3.1.2.1 Velocidad en Pruebas Secuenciales de BFS.



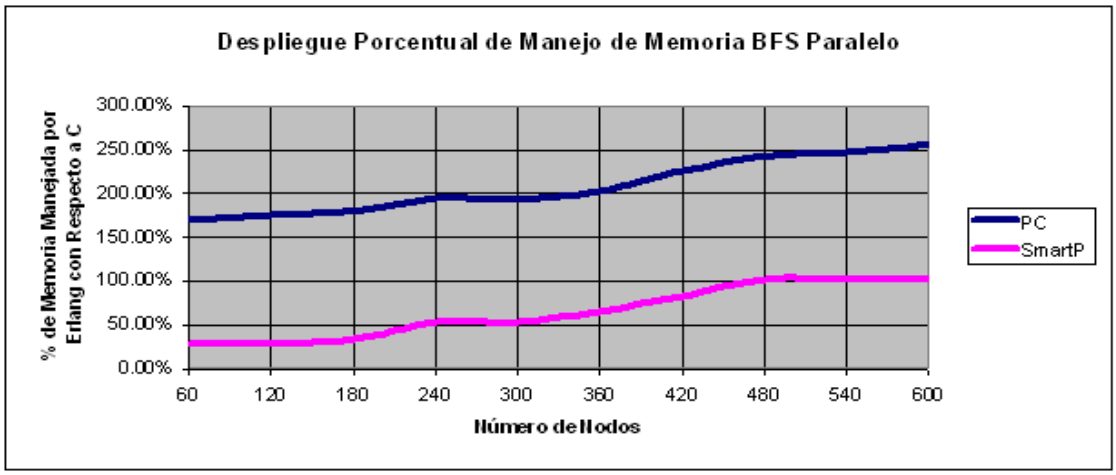
Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
PC Mejor	No	No	No	No	No	No	No	No	No	No
	4k	8k	12k	16k	20k	24k	28k	32k	36k	40k

Figura No. 4.3.1.2.2 Gasto de Memoria en Pruebas Secuenciales de BFS.



Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	No	No	No	No	No	No	No	No	No	No
PC Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
	60	120	180	240	300	360	420	480	540	600

Figura No. 4.3.1.2.3 Velocidad en Pruebas Paralelas de BFS.



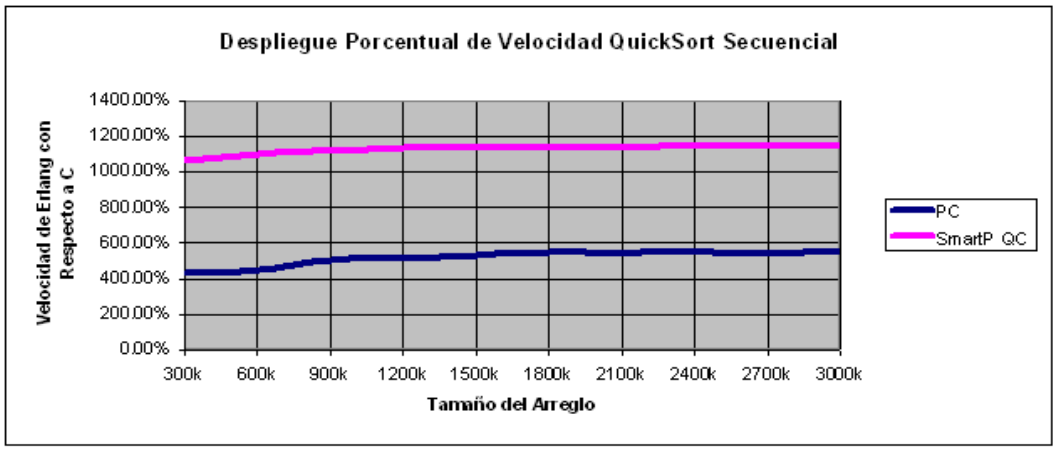
Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
PC Mejor	No	No	No	No	No	No	No	No	No	No
	60	120	180	240	300	360	420	480	540	600

Figura No. 4.3.1.2.4 Gasto de Memoria en Pruebas Paralelas de BFS.

4.3.2 Pruebas usando el smartphone de cuatro núcleos o Quadcore

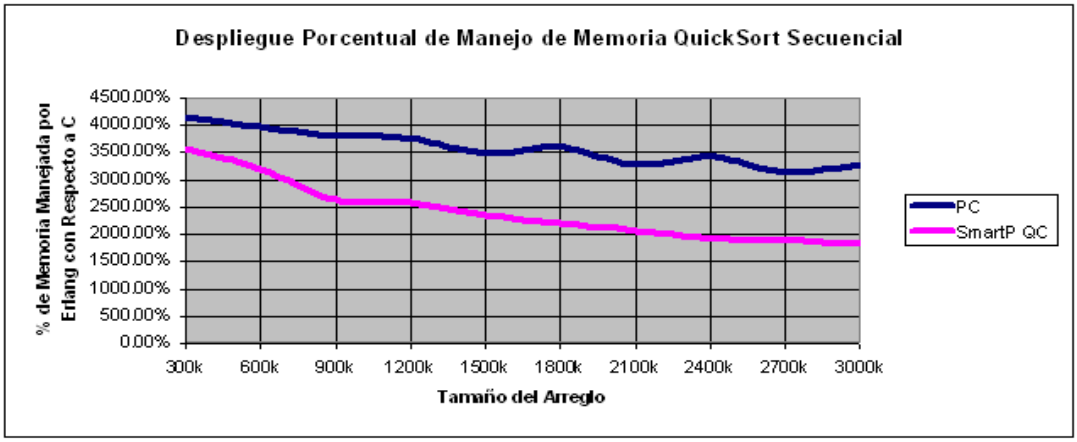
4.3.3.1 Quicksort

El tamaño del arreglo lo especifica cada gráfico, es el número de elementos a ordenar.



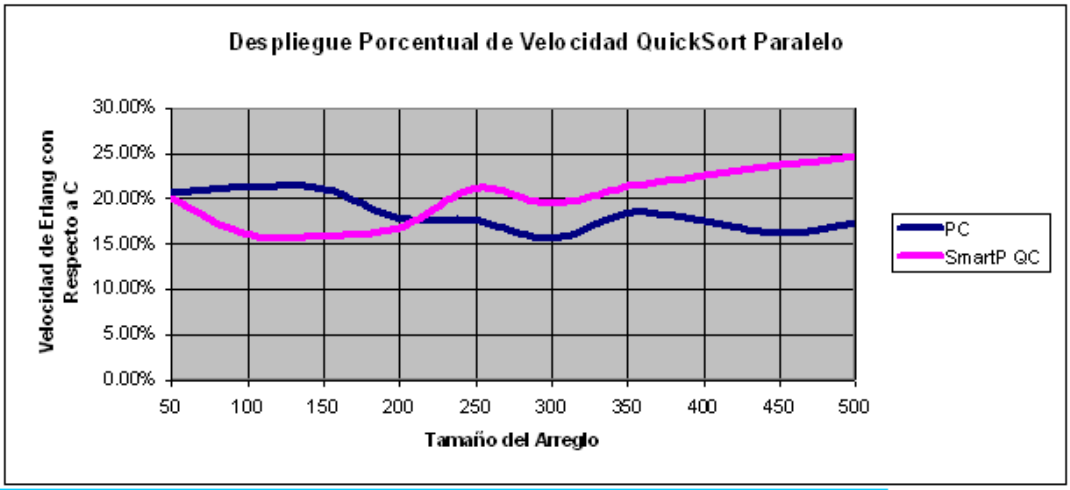
Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	No	No	No	No	No	No	No	No	No	No
PC Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
	300k	600k	900k	1200k	1500k	1800k	2100k	2400k	2700k	3000k

Figura No. 4.3.3.1.1 Velocidad en Pruebas Secuenciales de Quicksort.



Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
PC Mejor	No	No	No	No	No	No	No	No	No	No
	300k	600k	900k	1200k	1500k	1800k	2100k	2400k	2700k	3000k

Figura No. 4.3.3.1. Gasto de Memoria en Pruebas Secuenciales de Quicksort.

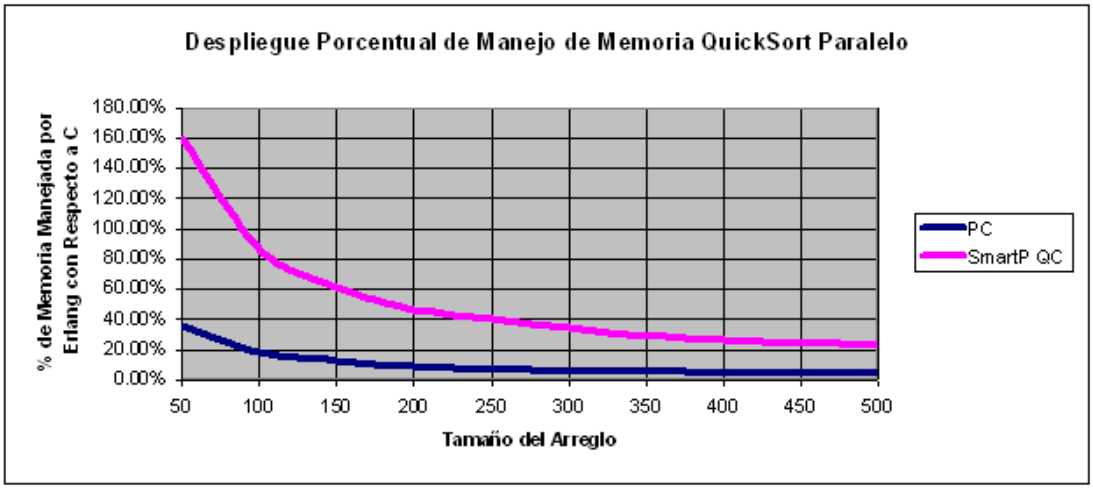


Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	NSPA	Si	Si	NSPA	Si	Si	Si	Si	Si	Si
SmartP Mejor	NSPA	Si	Si	NSPA	No	No	No	No	No	No
PC Mejor	NSPA	No	No	NSPA	Si	Si	Si	Si	Si	Si
	50	100	150	200	250	300	350	400	450	500

Nota: NSPA =No se puede asegurar

Figura No. 4.3.3.1.3 Velocidad en Pruebas Paralelas de Quicksort.

Se hace notar que en la Figura 4.3.3.1.3 Erlang dura menos que C, esto es por que según pruebas realizadas, los hilos en C consumen mucho más recursos que los procesos en Erlang. Al consumir más recursos se consume más tiempo en el Quicksort paralelo en C, ya que en este algoritmo se abren muchos hilos en C (e igual cantidad de procesos en la versión hecha en Erlang). En algunas pruebas hechas en el smartphone básico para saber la capacidad de crear procesos con Erlang se pudo crear más de 39 mil procesos en la ejecución del Quicksort paralelo para un arreglo de 40 mil elementos, mientras que en C no se pudo sobrepasar los 400 hilos por que se consumía toda la memoria del dispositivo.

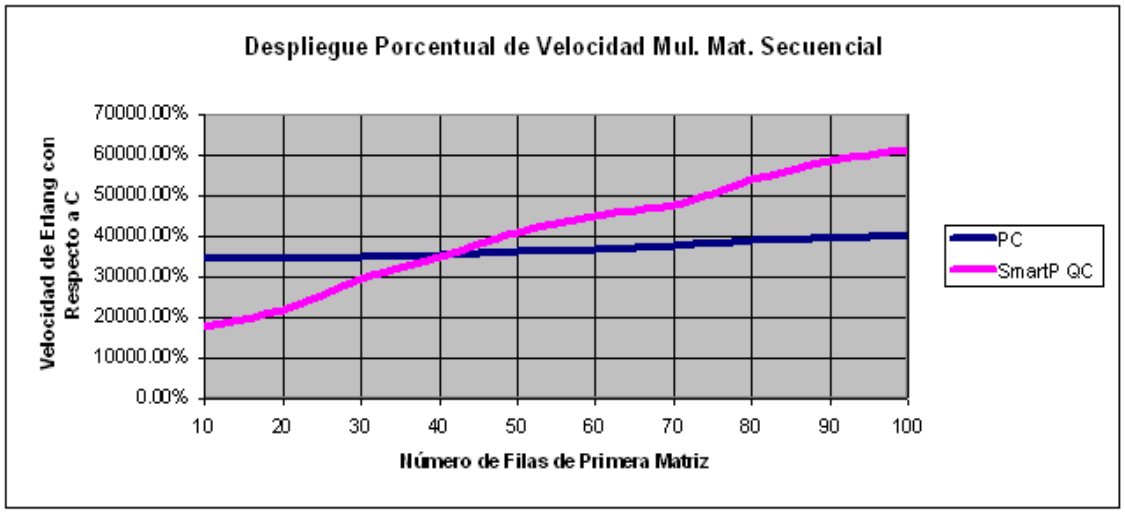


Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	No	No	No	No	No	No	No	No	No	No
PC Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
	50	100	150	200	250	300	350	400	450	500

Figura No. 4.3.3.1. Gasto de Memoria en Pruebas Paralelas de Quicksort.

4.3.3.2 Multiplicación de matrices

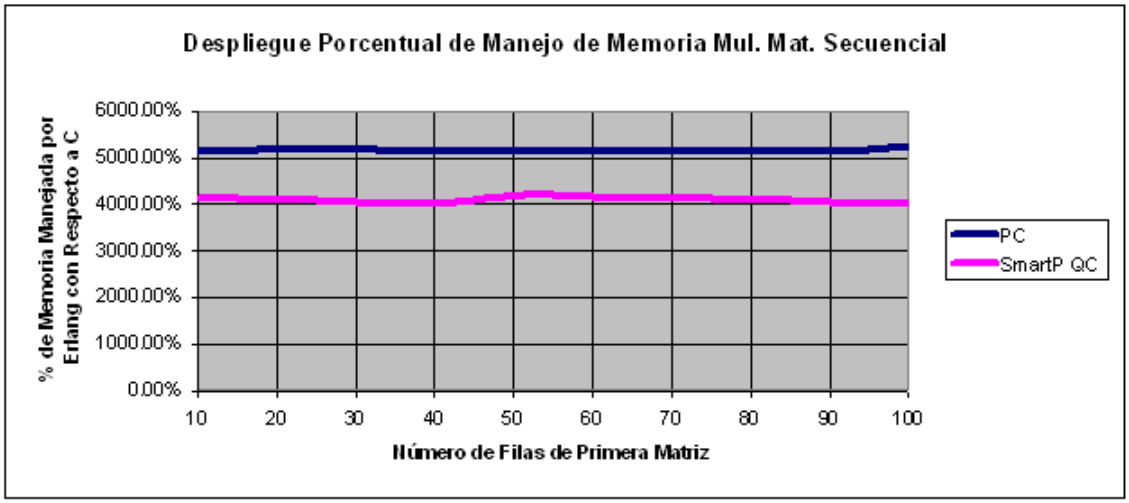
Las matrices multiplicadas en este aparatado para el algoritmo secuencial son (formato Filas * Columnas): la primera= N * 100 y la segunda = 100* 100, en donde N es variado dependiendo del tamaño de la prueba que muestre el gráfico. En el ambiente paralelo se ejecuta con los mismos tamaños de matrices (con excepto de N) a 4 hilos.



Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	NSPA	Si	Si	Si	Si	Si	Si
SmartP Mejor	Si	Si	Si	NSPA	No	No	No	No	No	No
PC Mejor	No	No	No	NSPA	Si	Si	Si	Si	Si	Si
	10	20	30	40	50	60	70	80	90	100

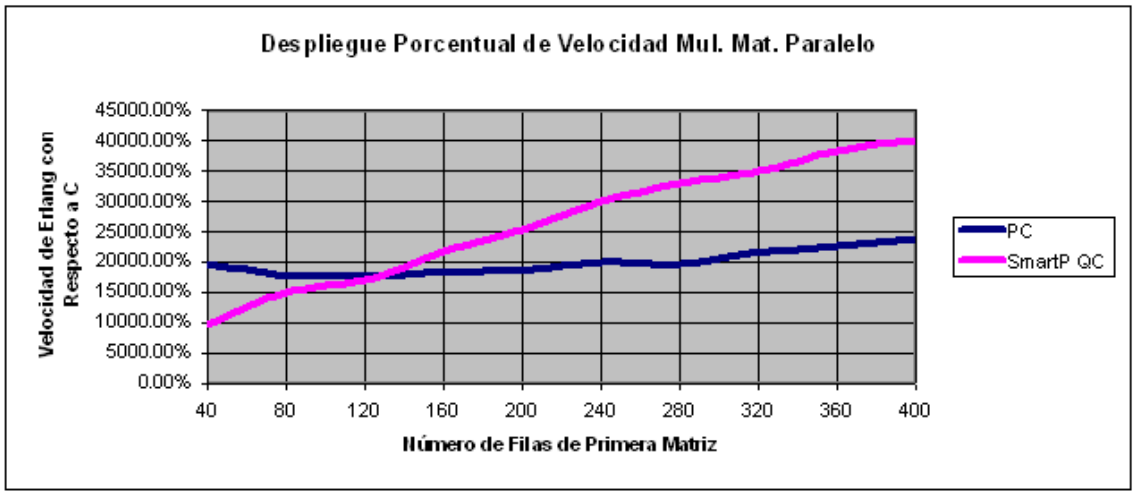
Nota: NSPA =No se puede asegurar

Figura No. 4.3.3.2.1 Velocidad en Pruebas Secuenciales de Multiplicación de Matrices.



Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
PC Mejor	No	No	No	No	No	No	No	No	No	No
	10	20	30	40	50	60	70	80	90	100

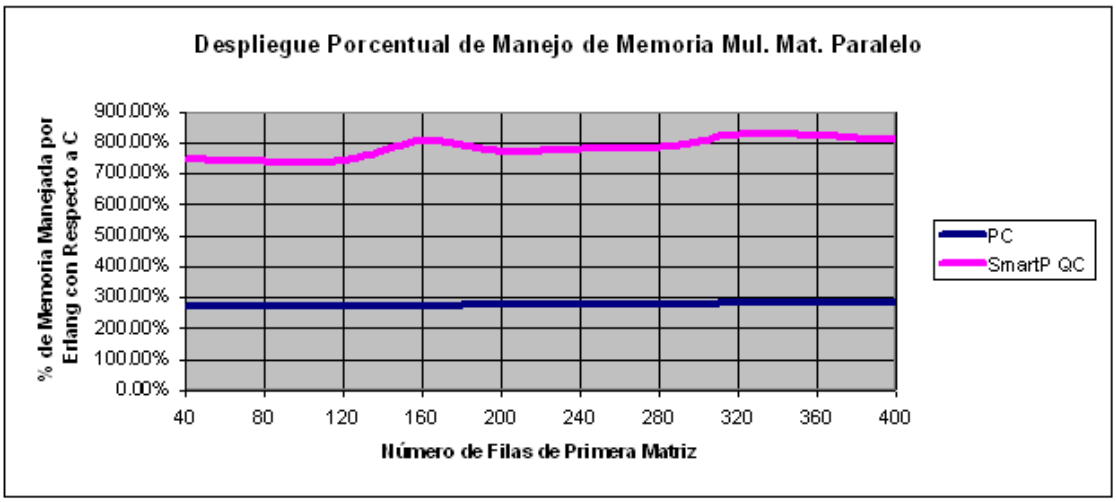
Figura No. 4.3.3.2.2 Gasto de Memoria en Pruebas Secuenciales de Multiplicación de Matrices.



Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	NSPA	NSPA	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	Si	NSPA	NSPA	No	No	No	No	No	No	No
PC Mejor	No	NSPA	NSPA	Si	Si	Si	Si	Si	Si	Si
	40	80	120	160	200	240	280	320	340	400

Nota: NSPA=No se puede asegurar

Figura No. 4.3.3.2.3 Velocidad en Pruebas Paralelas de Multiplicación de Matrices.

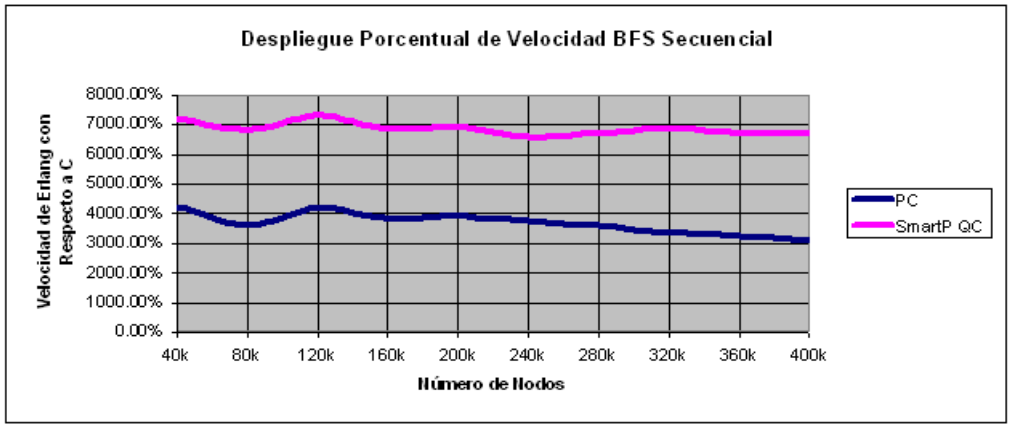


Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	No	No	No	No	No	No	No	No	No	No
PC Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
	40	80	120	160	200	240	280	320	340	400

Figura No. 4.3.3.2.4 Gasto de Memoria en Pruebas Paralelas de Multiplicación de Matrices.

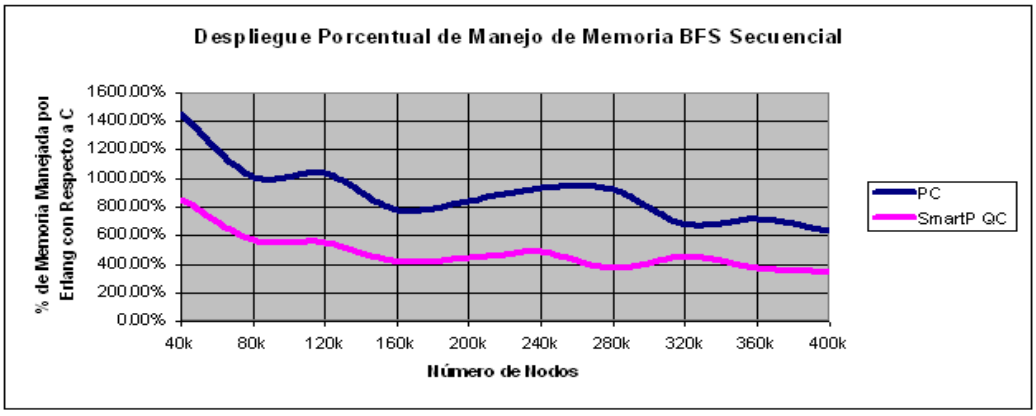
4.3.3.3 Búsqueda en anchura (BFS)

Los grafos tienen un tamaño en nodos, como lo especifica el gráfico, los grafos fueron construidos con un promedio de arcos de 4 por nodo, estos arcos apuntan a nodos escogidos de manera aleatoria.



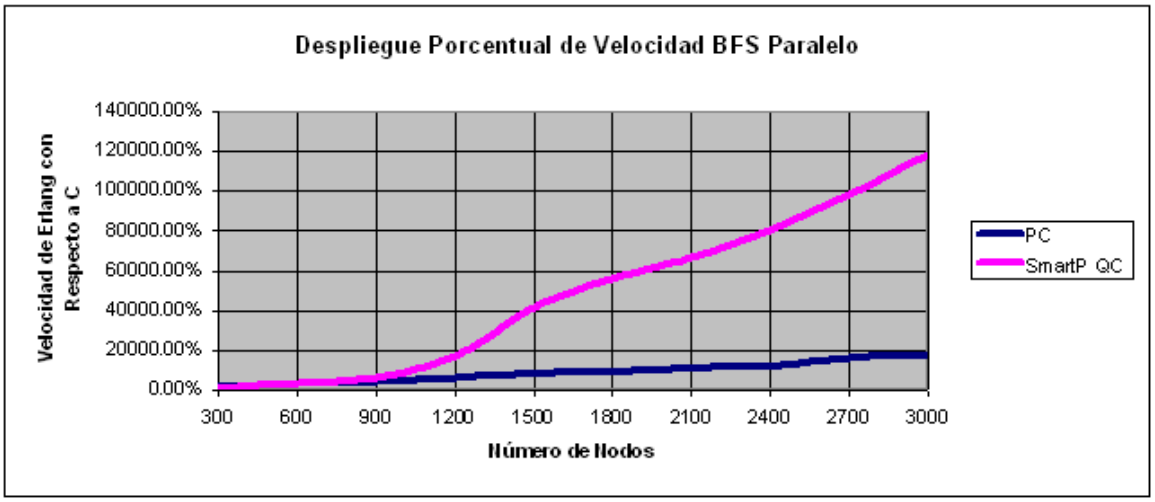
Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	No	No	No	No	No	No	No	No	No	No
PC Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
	40k	80k	120k	160k	200k	240k	280k	320k	360k	400k

Figura No. 4.3.3.3.1 Velocidad en Pruebas Secuenciales de BFS.



Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
PC Mejor	No	No	No	No	No	No	No	No	No	No
	40k	80k	120k	160k	200k	240k	280k	320k	360k	400k

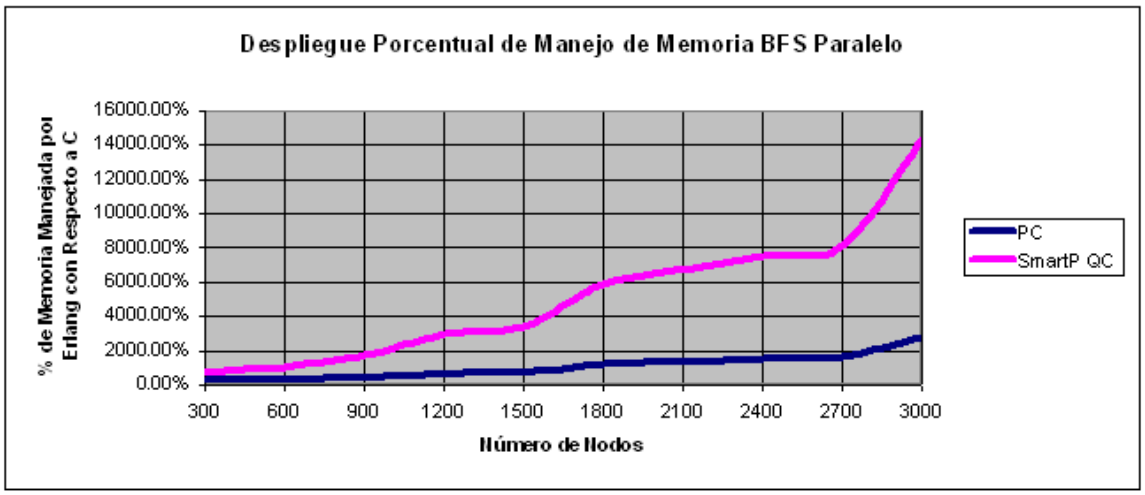
Figura No. 4.3.3.3.2 Gasto de Memoria en Pruebas Secuenciales de BFS.



Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	NSPA	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	Si	NSPA	No	No	No	No	No	No	No	No
PC Mejor	No	NSPA	Si	Si	Si	Si	Si	Si	Si	Si
	300	600	900	1200	1500	1800	2100	2400	2700	3000

Nota: NSPA=No se puede asegurar

Figura No. 4.3.3.3.3 Velocidad en Pruebas Paralelas de BFS.



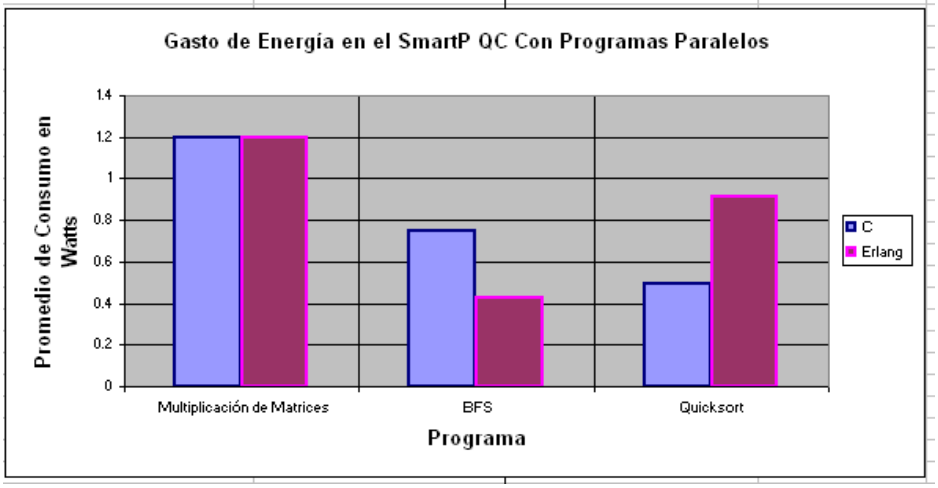
Resultados de Prueba de Contraste de Hipótesis										
Medias son Distintas	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
SmartP Mejor	No	No	No	No	No	No	No	No	No	No
PC Mejor	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
	300	600	900	1200	1500	1800	2100	2400	2700	3000

Figura No. 4.3.3.3.4 Gasto de Memoria en Pruebas Paralelas de BFS.

4.3.3.4 Pruebas de gasto de energía

Es importante también comparar el gasto de energía que implica ejecutar los algoritmos bajo el modelo de actores, ya que los smartphones dependen de una batería para funcionar y este tipo de medición ya ha sido considerado en estudios de este tipo [10]. Estas pruebas se realizan usando los mismos algoritmos con la intención de ver el gasto de energía entre C y Erlang. Los resultados se indican en el cuadro 4.3.3.4.1. Las pruebas son realizadas en paralelo a 4 hilos en el caso de multiplicación de matrices y BFS, en el caso de Quicksort se usan múltiples hilos según fue programado este algoritmo. En el gráfico se indica el promedio de gasto de energía obtenido con una aplicación para Android llamada PowerTutor usada en estudios de este tipo [10].

Tipo de Programa Paralelo	Características de la Prueba	Resultados de la Prueba	
Multiplicación de Matrices		Pruebas C	Pruebas Erlang
	Consumo en Watts	1.2	1.2
	Tamaño de matrices	4000 * 1000, 1000 * 1000	400 * 100, 100 * 200
	Tiempo de corrida en segundos	131.00	196.70
	Memoria consumida en KB	40,192.00	44,516.00
BFS			
	Consumo en Watts	0.75	0.43
	Número de nodos del grafo	1,000,000.00	3,000.00
	Tiempo de corrida en segundos	8.49	13.54
	Memoria consumida en KB	271,208.00	962,644.00
Quicksort			
	Consumo en Watts	0.49	0.92
	Tamaño de arreglo	11,000.00	160,000.00
	Tiempo de corrida en segundos	2.20	3.79
	Memoria consumida en KB	1,938,940.00	121,544.00



Cuadro No. 4.3.3.4.1 Gasto de Energía.

CAPÍTULO 5. CONCLUSIONES

Con el análisis realizado se procede a indicar el rechazo o aceptación de la hipótesis de esta tesis, esto se indica junto con las conclusiones a las que se llega en el apartado "Conclusiones y descubrimientos". Posteriormente se indican limitantes encontradas y futuras investigaciones a raíz de los hallazgos de esta investigación.

5.1 Conclusiones y descubrimientos

Al inicio este documento se plantea los siguientes aspectos a ser analizados para rechazar o aceptar la hipótesis:

1. Se indicará si se puede o no usar lenguajes concurrentes basados en el modelo de actores en los dispositivos móviles conocidos como smartphones.
2. Se indicará si este modelo es o no apropiado en los smartphones, esto desde el punto de vista de desempeño en ejecución de programas hechos bajo el modelo de actores, y se analizará la conveniencia para el desarrollador de aplicaciones bajo este modelo.
3. Se indicará las características encontradas bajo las cuales se puede tener mejores resultados y qué tan buenos son esos resultados usando el modelo de actores.

Analizando el primer aspecto, se tiene que al lograr migrar un lenguaje basado en el modelo de actores y ejecutar de manera satisfactoria programas en los tres diferentes escenarios planteados en esta tesis, se concluye que este modelo se puede usar en los dispositivos conocidos como smartphones.

Para el segundo aspecto, se tiene un resumen de lo que se aprecia en los gráficos de las secciones 4.2 y 4.3, agrupado por smartphone en el cuadro 5.1.

Smartphone	**Prueba**	**Resultado con Velocidad de Ejecución**	**Resultado con Manejo de Memoria**
Básico	Quicksort Secuencial	Malo	Bueno
	Quicksort Paralelo	Bueno	Bueno
	Quicksort Distribuido	Bueno	Bueno
	Multiplicación Matrices Secuencial	Malo	Bueno
	Multiplicación Matrices Paralelo	Malo	Bueno
	BFS Secuencial	Malo	Bueno
	BFS Paralelo	Malo	Bueno
Quadcore	Quicksort Secuencial	Malo	Bueno
	Quicksort Paralelo	Similar	Malo
	Multiplicación Matrices Secuencial	Malo	Bueno
	Multiplicación Matrices Paralelo	Malo	Malo
	BFS Secuencial	Malo	Bueno
	BFS Paralelo	Malo	Malo

Cuadro No. 5.1 Resumen Gráficos Finales de Prueba.

Según se muestra en el cuadro 5.1, para lo que respecta a velocidad casi siempre el smartphone presenta problemas, y solo en los casos en que en los procesos paralelos tienen poca carga de procesamiento y muchos hilos, entonces hay un buen desempeño en el smartphone básico y en el smartphone avanzado. El manejo de memoria no es bueno siempre, casi siempre es bueno en el smartphone básico, mientras que en el smartphone avanzado se tiene que a veces es bueno y a veces malo. En cuando al gasto de energía que

se observa en el cuadro 4.3.3.4.1, se observa un promedio de gasto parecido en C y Erlang en la prueba más larga, que fue la de multiplicación de matrices. Según el gráfico, hay gastos de energía que a veces eran más y a veces menos en los demás algoritmos usados para la prueba, por lo que se asume que los gastos de energía del modelo de actores es aceptable y comparable con el de las aplicaciones en C.

Por lo tanto, aunque los gastos de energía sí son aceptables, la respuesta para el segundo aspecto es que desde punto de vista de desempeño, las pruebas en esta investigación no pudieron demostrar que el modelo de actores fuera apropiado. Modificaciones a la implementación del modelo de actores que puedan ser implementadas en Erlang Embedded podrían indicar alguna diferencia en estos resultados, sin embargo, como se aclara anteriormente en esta investigación, se deja por fuera ese desarrollo porque no es un proyecto que sea mantenido en la actualidad, ya que se encuentra descontinuado.

Para el tercer y último aspecto, se tienen varias conclusiones, se debe considerar otros factores importantes para ver si conviene o no usar el modelo de actores. Uno de los más importantes es el tiempo de desarrollo, por ejemplo el tiempo necesario en esta tesis para programar y probar los algoritmos de pruebas en C tomó cuatro veces el tiempo que se usó para programar y probar los mismos algoritmos en Erlang. El ahorro en esta investigación en cuanto a tiempo de desarrollo y pruebas fue de un 75% de tiempo al programar en Erlang, por lo que se concluye que sí hay una ganancia sustancial en tiempo de desarrollo de las aplicaciones usando el modelo de actores.

Con respecto al número de líneas de código fuente, hay información disponible que indica que en Erlang se ahorra un 75% de líneas de código o más [19] [2], lo cual es un factor de peso a considerarse también. Si se revisa el cuadro 3.1.1, se observa que el total de líneas de los diferentes programas en C es 1804 mientras, que en Erlang es de 769. Esto indica que en C se tiene 2.34 veces más código fuente que en Erlang, lo cual muestra un ahorro de más del 57% de código en Erlang. Con ello que se concluye que sí hay un ahorro importante de cantidad de código fuente.

Otro beneficio que se observa es la disponibilidad de poder crear una cantidad muy grande de procesos en Erlang. Por ejemplo, en algunas pruebas alternas hechas en el smartphone básico para saber la capacidad de crear procesos con Erlang, se pudo crear más de 39 mil procesos en la ejecución del Quicksort paralelo para un arreglo de 40 mil elementos, mientras que en C no se pudo sobrepasar los 400 hilos por que se consumía toda la memoria del dispositivo. Por lo tanto, se concluye que aplicaciones que necesiten crear una gran cantidad de procesos paralelos se ven muy beneficiadas en el modelo de actores.

Si se observa las conclusiones para los 3 aspectos analizados hasta aquí, los experimentos realizados no respaldan la hipótesis planteada inicialmente: "El modelo de actores es un modelo robusto para ser utilizado en los smartphone en programación concurrente". Básicamente porque las medidas de desempeño en el segundo aspecto no dieron buenos resultados, pero debido a que hay aplicaciones como Whatsapp [30] ya funcionando para smartphones construidas usando el modelo de actores, se concluye que la utilidad del

modelo no depende de que las aplicaciones sean corridas en un smartphone o PC, sino que más bien depende del tipo de aplicación que se está ejecutando.

Otra conclusión es con respecto a las aplicaciones que mas podrían beneficiarse del modelo de actores. Estas aplicaciones son aquellas en donde por su complejidad en el uso paralelo de múltiples procesos es mejor manejarlas a un alto nivel de abstracción minimizando los errores típicos de programación concurrente, en donde un gasto mayor de memoria no sea un problema y en donde se tengan tiempos cortos de ejecución en los procesos. Tal es el caso de servicios para aplicaciones web o aplicaciones de control de múltiples dispositivos o sensores, esto debido a que el modelo según el cuadro 5.1 es bueno en velocidad de ejecución de programas que manejan muchos hilos con procesamiento liviano en el Smartphone, aunque con un gasto de memoria mayor a lo esperado.

Con respecto a este tipo de investigación en particular, se concluye adicionalmente que el riesgo de cometer errores a la hora de manipular muchos datos de prueba en este tipo de investigaciones es grande. Este tipo de investigaciones están muy expuestas al error humano, por lo que se recomienda automatizar lo más posible el análisis de información. En esta investigación se logró automatizar parte de la manipulación de datos con hojas de cálculo que leen datos de otras hojas de cálculo de manera automática para generar gráficos, pero se recomienda automatizar aún más el proceso con programas de análisis de datos a partir de archivos de información generados desde los programas de prueba, con el

fin de minimizar la manipulación y con ello evitar el error humano en el análisis de los resultados.

5.2 Limitaciones enfrentadas durante la investigación

Una de las limitantes más importantes es la poca existencia de programas para probar el modelo de actores y compararlo con C. No se logró encontrar pares de programas, uno hecho en C y otro echo en Erlang, que siguieran la misma estructura y que sirvieran para establecer comparaciones entre el paradigma de hilos y el modelo de actores.

Otra limitante muy importante la constituye el tiempo que consume este tipo de investigaciones, se tiene que con una prueba se generaba el dato de consumo de memoria y velocidad de ejecución de la misma, por lo tanto cada prueba ayuda a alimentar datos para 2 gráficos. En la sección 4.2 y 4.3 se muestran 26 gráficos, cada par de ellos se generó a partir de 1 400 pruebas, lo que implica 18 200 pruebas. Esto acarrea mucho trabajo por el número de pruebas y por el análisis que se hace de las mismas montando los datos en hojas de cálculo que generen los gráficos.

Por último, otra limitante encontrada es los recursos materiales para hacer pruebas, ya que después del análisis hecho y según las conclusiones obtenidas, se considera importante hacer otro tipo de pruebas en donde intervengan muchos dispositivos a la vez. El contar con muchos dispositivos no es común para este tipo de pruebas.

5.3 Futuras investigaciones

Para trabajos futuros es necesario probar el modelo en ambientes más diversos en donde varios nodos participen, ya que en el presente estudio se evalúa la utilidad del modelo de actores corriendo principalmente en un solo nodo de aplicaciones de procesamiento intensivo. Según las debilidades del modelo expuestas en el apartado 5.1, surge como proyecto futuro el modificar el modelo de actores para lograr mejores resultados en los smartphones en aplicaciones de procesamiento intensivo. Esta investigación aporta gran cantidad de recursos para probar las mejoras que se pudieran dar al modelo.

CAPÍTULO 6. BIBLIOGRAFÍA

[1] Agha G. "Actors: A Model of Concurrent Computation in Distributed Systems". MIT Press. *Cambridge, MA, USA ©1986,* https://www.cypherpunks.to/erights/history/actors/AITR-844.pdf

[2] Armstrong J, “Comparing number of lines of code Erlang <-> C”, 19 Feb 2004 , http://erlang.org/pipermail/erlang-questions/2004-February/011647.html

[3] Armstrong J., Virding R., Wikström C., Williams M., *Concurrent Programming in Erlang*, 2nd Edition, Prentice Hall, 1996, http://www.erlang.org/download/erlang-book-part1.pdf

[4] Bustard D., “Concepts of Concurrent Programming”, Carnegie Mellon University - Software Engineering Institute, , April 1990, http://resources.sei.cmu.edu/asset_files/CurriculumModule/1990_007_001_15815.pdf

[5] Canavos G., *Probabilidad y Estadística Aplicaciones y Métodos***,** México, McGraw-Hill, 1988.

[6] Casas J., *Inferencia Estadística Para Economía y Administración de Empresas***,** Editorial Centro de Estudios Ramón Areces S. A., 1988.

[7] Cunningham & Cunningham Inc., *Actors Model*, December 4 2014, http://c2.com/cgi/wiki?ActorsModel

[8] Cunningham & Cunningham Inc, *Erlang Language*. Junio de 2007. http://c2.com/cgi/wiki?ErlangLanguage

[9] Debian Home Site, https://www.debian.org/distrib/packages

[10] Denti M., Nurminen J., "Performance and Energy-Efficiency of Scala on Mobile Devices", Aalto University School of Science, May 19, 2013, http://cse.aalto.fi/en/midcom-serveattachmentguid-1e38756839a9dea875611e3b6c5bdaeedf47f937f93/denti_ngmast.pdf

[11] Domhan T., "Augmented Reality on Android Smartphones", Universidad Baden-Wuerttemberg , Alemania, 8 Junio 2010.

[12] Fielding R., "Architectural Styles and the Design of Network-based Software Architectures", University Of California, Irvine, 2000, https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[13] Frequently Asked Questions: Intel® Multi-Core Processor Architecture, "The Move to Multi-Core Architecture Explained, Why is Intel implementing multi-core architectures across its product line?", Intel Developer Zone, https://software.intel.com/en-

us/articles/frequently-asked-questions-intel-multi-core-processor-architecture#_The_move_to_dual/multi-core%20explain

[14] Gray J., “Why Do Computers Stop and What Can Be Done About It?”, Tandem Computers, June 1985, http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf

[15] Hewitt C., Bishop P., Steiger R.,"A Universal Modular Actor Formalism for Artificial Intelligence". MIT Artificial Intelligence Laboratory, 1973, http://dli.iiit.ac.in/ijcai/IJCAI-73/PDF/027B.pdf

[16] Hewitt C., “Viewing control structures as patterns of passing messages”, MIT Artificial Intelligence Laboratory, December 1976, https://www.cypherpunks.to/erights/history/actors/AIM-410.pdf

[17] Huang Z., Hui P., Peylo C., Chatzopoulos, “Mobile Augmented Reality Survey: A Bottom-up Approach”, Cornell University Library, 18 Sep 2013

[18] John-Paul Bader, *Why Erlang?*, April 22, 2012 http://smyck.net/2012/04/22/why-erlang/

[19] Jones R., “Rewriting Playdar: C++ to Erlang, massive savings”, 21 October 2009. http://www.metabrew.com/article/rewriting-playdar-c-to-erlang-massive-savings

[20] Karmani R., Agha G., “Actors”, Open Systems Laboratory - Department of Computer Science - University of Illinois at Urbana-Champaign, http://web.cs.ucla.edu/~palsberg/course/cs239/papers/karmani-agha.pdf

[21] Kilic O. *Erlang Embedded*, Erlang Solutions, October 31, 2013, http://www.erlang-embedded.com/

[22] Khalilieh H., Kafri N., Mohammad R., “Performance Evaluation of Message Passing vs. Multithreading Parallel Programming Paradigms on Multi-core Systems”, The Society of Digital Information and Wireless Communications, 2014 (ISSN: 2220-9085), http://dx.doi.org/10.17781/p009

[23] Madduri K., “Scaling up graph algorithms on emerging multicore systems“, Lawrence Berkeley National Laboratory, http://www.graphanalysis.org/SIAM-AN09/04_Madduri.pdf

[24] Mendenhall W., Beaver R., Beaver B., *Introducción a la probabilidad y estadística*, México, Cengage Learning, 2006

[25] Miller A., “Understanding actor concurrency, Part 1: Actors in Erlang”, *JavaWorld*, Feb 24, 2009, http://www.javaworld.com/article/2077999/java-concurrency/understanding-actor-concurrency--part-1--actors-in-erlang.html

[26] Moon S., *TCP/IP socket programming in C*, Dec 24, 2011, http://www.binarytides.com/socket-programming-c-linux-tutorial/

[27] Nortier B., "Parallel Quicksort in Erlang", 21st Century Code Works, Thursday, 24 April 2008, http://21ccw.blogspot.com/2008/04/parallel-quicksort-in-erlang.html

[28] Nvidia, "The Benefits of Multiple CPU Cores in Mobile Devices", nvidia whitepaper, https://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf, 2010 NVIDIA Corporation.

[29] O'Toole J., "Mobile apps overtake PC Internet usage in U.S.", CNNMoney (New York), February 28, 2014, http://money.cnn.com/2014/02/28/technology/mobile/mobile-apps-Internet/

[30] Open-source Erlang, http://www.erlang.org

[31] Open-source Erlang, "Supervision Principles", http://www.erlang.org/documentation/doc-4.9.1/doc/design_principles/sup_princ.html

[32] Parallel Programming Laboratory, "Parallel Languages/Paradigms: Charm++ Parallel Programming with Migratable Objects", Department of computing Science – University of Illinois, http://charm.cs.illinois.edu/research/charm

[33] Piedra R., “A Parallel Approach for Matrix Multiplication on the TMS320C4x DSP“, Texas Instruments, February 1994, http://www.ti.com/lit/an/spra107/spra107.pdf

[34] Purcell A., “Smartphone grids – the future for distributed computing?”, International science grid this week, September 12, 2012, http://www.isgtw.org/feature/smartphone-grids-%E2%80%93-future-distributed-computing

[35] Rogozhin K, “Controlling memory consumption with Intel® Threading Building Blocks (Intel® TBB) scalable allocator”, Intel Developer Zone, Tue 03/18/2014, , https://software.intel.com/en-us/articles/controlling-memory-consumption-with-intel-threading-building-blocks-intel-tbb-scalable

[36] Scala, http://www.scala-lang.org/

[37] Siek J., “Review of Elementary Graph Theory”, Indiana University, http://www.boost.org/doc/libs/1_59_0/libs/graph/doc/graph_theory_review.html

[38] Székely A., Talanow R., Bágyi P., “Smartphones, tablets and mobile applications for radiology”, European Journal of Radiology (82) (2013) 829-836,

[39] Sourceforge, http://sourceforge.net/projects/linwizard/

[40] The Society of Digital Information and Wireless Communications (SDIWC), http://www.sdiwc.net/

[41] Theriault, D. G., "A Primer for the Act-1 Language", MIT Artificial Intelligence Laboratory, April 1982, http://dspace.mit.edu/handle/1721.1/5675

[42] Typesafe Inc. , "Fault Tolerance", http://doc.akka.io/docs/akka/snapshot/scala/fault-tolerance.html

[43] University of California, *Structure and Interpretation of Computer Programs, Chapter 4: Distributed and Parallel Computing*, Abril 2015, http://wla.berkeley.edu/~cs61a/fa11/lectures/communication.html

[44] Wagner D, Schmalstieg D., "Making Augmented Reality Practical on Mobile Phones, Part 1", Graz University of Technology, http://www.computer.org/cms/ComputingNow/homepage/2009/0609/rW_CG_MakingAugmentedReality.pdf

[45] WiseGEEK, "What Is a Personal Digital Assistant?", http://www.wisegeek.org/what-is-a-personal-digital-assistant.htm

CAPÍTULO 7. APÉNDICES

7.1 Glosario

Algoritmo: El algoritmo es un conjunto de pasos, instrucciones o acciones que se deben seguir para resolver un problema.

Aplicación de cálculo intensivo: Son programas que por su complejidad requieren el uso pronunciado del procesador o procesadores para ejecutar una serie de cálculos numéricos o lógicos.

Aplicaciones web: Las aplicaciones web reciben este nombre porque se ejecutan en Internet.

Arreglo (programación): Es una colección de datos del mismo tipo. Sirve para manejar un número "n" de elementos en común.

BFS o búsqueda en anchura: Es un algoritmo para recorrer grafos, es una forma sistemática de encontrar todos los vértices alcanzables de un grafo desde un vértice de origen.

C (lenguaje): C es un lenguaje de programación de propósito general que ofrece economía sintáctica, control de flujo y estructuras sencillas y un buen conjunto de operadores. No es un lenguaje de muy alto nivel y más bien un lenguaje pequeño, sencillo y no está especializado en ningún tipo de aplicación.

Coeficiente de variabilidad: El coeficiente de variación es una medida de dispersión que describe la cantidad de variabilidad en relación con la media.

Computadora de escritorio: Es un tipo de computadora personal, diseñada y fabricada para ser instalada en una ubicación fija, como un escritorio o mesa, a diferencia de otras computadoras, como las portátiles, notebooks, netbooks, laptops o ultrabooks.

Crosscompiling: Se refiere a compilar usando un compilador que crea código ejecutable para una plataforma diferente a la usada para ejecutar el compilador.

Crossover (cable): Es un cable de red con una configuración especial que permite conectar dos computadoras entre sí para pasar información, también sirve para la comunicación entre routers y firewalls.

Desktop: Consultar Computadora de escritorio.

Distribuido (algoritmo): Algoritmo que ejecuta procesos en varios dispositivos o computadores conectados entre sí para un fin en común.

Embedded (Erlang): Versión del lenguaje Erlang construido para correr en dispositivos empotrados. Los sistemas empotrados son sistemas de computación diseñado para realizar una o algunas pocas funciones dedicadas, frecuentemente en un sistema de computación en

tiempo real. Al contrario de lo que ocurre con los ordenadores de propósito general (como por ejemplo, una computadora personal o PC), que están diseñados para cubrir un amplio rango de necesidades, los sistemas embebidos se diseñan para cubrir necesidades específicas.

Erlang (lenguaje): Lenguaje de programación declarativo basado en el modelo de actores.

FIFO (cola de mensajes): Se refiere a una cola de mensajes en donde el primero que se recibe es el primero que se atiende y sale de la cola. Funciona diferente de una pila (LIFO) en donde el último recibido es el primero en atenderse y salirse de la cola.

GPS (aplicaciones): Aplicación que usa el sistema de posicionamiento global (GPS), el cual es un sistema que permite determinar en todo el mundo la posición de un objeto (una persona, un vehículo) con una precisión de hasta centímetros (si se utiliza GPS diferencial), aunque lo habitual son unos pocos metros de precisión.

Gráfico en 3D: Este tipo de gráficos se originan mediante un proceso de cálculos matemáticos sobre entidades geométricas tridimensionales producidas en un ordenador.

Grafo: Es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto. Típicamente, un grafo se representa gráficamente como un conjunto de puntos (vértices o nodos) unidos por líneas (aristas).

Hilos (Programación): Un hilo de ejecución, hebra o subproceso, es la unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo. Un hilo es simplemente una tarea que puede ser ejecutada al mismo tiempo con otra tarea.

Hoja de cálculo: Una hoja de cálculo es un tipo de documento que permite manipular datos numéricos y alfanuméricos dispuestos en forma de tablas compuestas por celdas, las cuales se suelen organizar en una matriz bidimensional de filas y columnas.

Iteración (en informática): Significa el acto de repetir un proceso con el objetivo de alcanzar una meta deseada, objetivo o resultado. A cada repetición del proceso también se le denomina una "iteración", y los resultados de una iteración se utilizan como punto de partida para la siguiente iteración.

Linux: Es uno de los términos empleados para referirse a la combinación del núcleo o *kernel* libre similar a Unix denominado Linux con el sistema operativo GNU. Es un Sistema Operativo como MacOS, DOS o Windows.

Matriz (en informática): Estructura de datos usada para manejar los datos que comúnmente se manejan en una matriz matemática, la matriz matemática en general es un conjunto ordenado en una estructura de filas y columnas.

Memoria: La memoria es el dispositivo que retiene, memoriza o almacena datos informáticos durante algún intervalo de tiempo.

Memoria virtual: La memoria virtual es una técnica de gestión de la memoria que permite que el sistema operativo disponga, tanto para el software de usuario como para sí mismo, de mayor cantidad de memoria de la que esté disponible físicamente.

Métrica (en informática): Hace referencia a la medición del software con base en parámetros predeterminados, como puede ser el número de líneas de código de que consta o el volumen de documentación asociada.

Multiplicación de matrices: La multiplicación o producto de matrices es la operación de composición efectuada entre dos matrices, o bien la multiplicación entre una matriz y un arreglo según unas determinadas reglas. La definición de la multiplicación de matrices indica una multiplicación renglón-por-columna, donde las entradas en el renglón *i* de A son multiplicadas por las entradas correspondientes en el renglón *j* de B y luego se suman los resultados.

Paradigma de Programación: Un paradigma de programación es una propuesta tecnológica adoptada por una comunidad de programadores y desarrolladores cuyo núcleo central es incuestionable en cuanto que únicamente trata de resolver uno o varios problemas claramente delimitados. La resolución de estos problemas debe suponer consecuentemente un avance significativo en al menos un parámetro que afecte a la ingeniería de software.

Paralelo (algoritmo): Es un algoritmo que puede ser ejecutado por partes en el mismo instante de tiempo por varias unidades de procesamiento (las cuales se encuentran en el mismo dispositivo y no suponen una red de computadores), para finalmente unir todas las partes y obtener el resultado correcto.

PC: Consultar Computadora de escritorio.

Procesador (en informática): El elemento que interpreta las instrucciones y procesa los datos de los programas de computadora.

PThreads: Es una biblioteca que cumple los estándares POSIX y que nos permite trabajar con distintos hilos de ejecución (threads) al mismo tiempo.

Quadcore: Indica que el dispositivo posee cuatro procesadores.

Quicksort: Es un algoritmo de ordenación considerado entre los más rápidos y eficientes basado en la técnica de divide y vencerás.

Realidad aumentada: Es el término que se usa para definir una visión a través de un dispositivo tecnológico, directa o indirecta, de un entorno físico del mundo real, cuyos elementos se combinan con elementos virtuales para la creación de una realidad mixta en tiempo real.

Secuencial (algoritmo): Son un conjunto de pasos, procedimientos y acciones que se deben ejecutar de manera ordenada para solucionar un problema. Su funcionamiento se basa en ejecutar la primera instrucción y así sucesivamente hasta llegar al final. En estos algoritmos no se puede ejecutar más de un paso a la vez.

Sensor: Es un dispositivo capaz de detectar magnitudes físicas o químicas, llamadas variables de instrumentación, y transformarlas en variables eléctricas.

Sistema Operativo: Es un programa o conjunto de programas de un sistema informático que gestiona los recursos de hardware y provee servicios a los programas de aplicación.

SmarP: Es el smartphone básico con un solo procesador que se usa en esta investigación.

SmarP QC: Es el smartphone avanzado con 4 procesadores que se usa en esta investigación.

Smartphone: Es un término comercial para denominar a un teléfono móvil que ofrece más funciones que un teléfono móvil común. Permite la instalación de programas para incrementar sus posibilidades, como el procesamiento de datos y la conectividad. Estas aplicaciones pueden ser desarrolladas por el fabricante del dispositivo, por el operador o por un tercero.

Socket: Es un método para la comunicación entre un programa del cliente y un programa del servidor en una red. Un socket se define como el punto final en una conexión. Los

sockets se crean y se utilizan con un sistema de peticiones o de llamadas de función a veces llamados interfaz de programación de aplicación de sockets (API, application programming interface).

Tablet: es una computadora (ordenador) portátil más grande que un smartphone pero más pequeña que una netbook. Se caracteriza por contar con pantalla táctil: esto quiere decir que para utilizar la tablet no se necesita mouse (ratón) ni teclado.

TMEF (Tiempo medio entre fallos): El TMEF es el tiempo medio entre cada ocurrencia de una parada específica por fallo (o avería) de un proceso.

TMRF (Tiempo de recuperación de un fallo): Es el tiempo promedio que toma reparar algo después de una falla.

Ubuntu: Es un sistema operativo basado en GNU/Linux y que se distribuye como software libre.

7.2 Programas de pruebas

7.2.1 Programa de pruebas en lenguaje C

7.2.1.1 Quicksort

Tesis_qsort.c

```
/* Hecho por Hugo Mora.

Compilation example:
  gcc -lpthread -Wall -std=c99 tesis_qsort.c misc_functions.c -otesis_qsort_exe.o

Use examples:
  ./tesis_qsort_exe.o 10 20000000 1
  ./tesis_qsort_exe.o 10 5000000 2
  ./tesis_qsort_exe.o 6 5000 3 127.0.0.1 5081 127.0.0.1 5085 --client
  ./tesis_qsort_exe.o 6 5000 4 127.0.0.1 5085 127.0.0.1 5081 --server
*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <string.h>
#include <errno.h>

#include <sys/types.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <strings.h>

#include <sys/resource.h>
#include <unistd.h>
#include "misc_functions.h"

#include <arpa/inet.h>
#include <signal.h>

//for sequential and parallel implementation
void swap(int lyst[], int i, int j);
int partition(int lyst[], int lo, int hi);
void quicksortHelper(int lyst[], int lo, int hi);
void quicksort(int lyst[], int size);
int isSorted(int lyst[], int size);

//for parallel implementation
void parallelQuicksort(int lyst[], int size, int tlevel);
void *parallelQuicksortHelper(void *threadarg);

struct thread_data{
  int *lyst;
  int low;
  int high;
  int level;
};

typedef struct {
  int sc;
} param_thread_server;

```

```
56  /********************************** distributed quicksort
    ******************************/
57
58  void* quick_sort_server();
59  void end_quick_sort_server ();
60  void* sort_in_this_server(void* sockt_data_id);
61  void* sort_request_to_other_server(void* threadarg_struc);
62  void distributedQuicksort (void* threadarg_struc);
63  int local_server_mounted;
64  pthread_mutex_t mutex_server_mounted;
65  /*****************************************************************************************
66
67  int Num_hilos;
68  int Num_elem_vector;
69  pthread_mutex_t mutexsum;
70  int program;
71  int *lyst;
72
73
74  char* remote_server;
75  char* local_server;
76  int local_port;
77  int remote_port;
78  int *lystbck;
79  int thread_level;
80
81  int main (int argc, char *argv[])
82  {
83    struct timeval start, end;
84    double diff;
85    int rc;
86
87    srand(time(NULL)); //seed random
88
89    if (argc <=3)
90    {
91      printf("Uso: <programa> <nivel máximo de árbol> <Num_elem_vector elementos> <tipo
      algoritmo:
92      1=quicksort secuencial,2=quicksort paralelo, 3=quicksort distribuido, 4=modo
      servidor para quicksort distribuido>
93      [<ip servidor local> <puerto servidor local> <ip servidor remoto> <puerto
      servidor remoto>] \n");
94      exit(-1);
95    }
96    program = atoi(argv[3]);
97    if (((program == 3 || program == 4) && argc == 8) || (program>=1 && program <=2 &&
    argc == 4))
98    {
99      Num_elem_vector = atoi(argv[2]);
100     thread_level = atoi(argv[1]);
101
102     if (argc == 8)
103     {
104       local_server = argv[4];
105       local_port = atoi(argv[5]);
```

```
106       remote_server = argv[6];
107       remote_port = atoi(argv[7]);
108     }
109   }
110   else
111   {
112     printf("Uso: <programa> <Num_elem_vector elementos> <tipo algoritmo:1=quicksort
      secuencial,2=quickort paralelo,
113       3=quicksort distribuido, 4=modo servidor para quicksort distribuido> [<puerto
      servidor local>
114       <ip servidor remoto> <puerto servidor remoto>] \n");
115     exit(-1);
116   }
117
118   if (program <= 3)
119   {
120     lystbck = (int *) malloc(Num_elem_vector*sizeof(int));
121     lyst = (int *) malloc(Num_elem_vector*sizeof(int));
122
123     for(int i = 0; i < Num_elem_vector; i ++)
124     {
125       lystbck[i] = rand();
126     }
127     memcpy(lyst, lystbck, Num_elem_vector*sizeof(int));
128
129   }
130
131   if (program==1)
132   {
133     gettimeofday(&start, NULL);
134     quicksort(lyst, Num_elem_vector);
135     gettimeofday(&end, NULL);
136
137     if (!isSorted(lyst, Num_elem_vector))
138     {
139       printf("Error, list is not sorted by Quicksort.\n");
140     }
141
142     diff = ((end.tv_sec * 1000000 + end.tv_usec)
143         - (start.tv_sec * 1000000 + start.tv_usec))/1000000.0;
144     printf("Sequential quicksort took: %lf sec. and memory= %ld KB\n", diff,
      get_my_peak_used_mem());
145   }
146
147
148   //Now, parallel quicksort.
149
150   if (program==2)
151   {
152     Num_hilos = 0;
153
154     gettimeofday(&start, NULL);
155     parallelQuicksort(lyst, Num_elem_vector, thread_level);
156     gettimeofday(&end, NULL);
157
```

```
158     if (!isSorted(lyst, Num_elem_vector))
159     {
160       printf("Error, list is not sorted by parallel Quicksort.\n");
161     }
162
163     diff = ((end.tv_sec * 1000000 + end.tv_usec)
164         - (start.tv_sec * 1000000 + start.tv_usec))/1000000.0;
165     printf("Parallel quicksort took: %lf sec. # threads= %d and memory= %ld KB\n",
        diff, Num_hilos, get_my_peak_used_mem());
166   }
167
168
169   if (program==3)
170   {
171
172     Num_hilos = 0;
173     /************* mounting local server *************************/
174     pthread_t thread_id;
175
176     pthread_mutex_lock (&mutex_server_mounted);
177     local_server_mounted = -1;
178     pthread_mutex_unlock (&mutex_server_mounted);
179
180     if((rc=pthread_create( &thread_id , NULL, quick_sort_server, NULL)))
181     {
182           printf("Error; return code from pthread_create() is %d\n", rc);
183           exit(-1);
184           }
185
186     /***************************************/
187     while (1)
188     {
189       pthread_mutex_lock (&mutex_server_mounted);
190
191       if (local_server_mounted==0)
192       {
193         pthread_mutex_unlock (&mutex_server_mounted);
194         break;
195       }
196       else
197         pthread_mutex_unlock (&mutex_server_mounted);
198
199     }
200
201
202     struct thread_data thread_data_tmp;
203
204     thread_data_tmp.lyst = lyst;
205     thread_data_tmp.low = 0;
206     thread_data_tmp.high = Num_elem_vector - 1;
207     thread_data_tmp.level = thread_level;
208
209
210     gettimeofday(&start, NULL);
211
```

```
212     distributedQuicksort((void *) &thread_data_tmp);
213
214     gettimeofday(&end, NULL);
215
216     if (!isSorted(lyst, Num_elem_vector))
217     {
218       printf("Error, list is not sorted by distributed Quicksort.\n");
219     }
220
221     diff = ((end.tv_sec * 1000000 + end.tv_usec)
222         - (start.tv_sec * 1000000 + start.tv_usec))/1000000.0;
223     printf("Distributed quicksort took: %lf sec. # threads= %d and memory= %ld KB\n",
      diff, Num_hilos, get_my_peak_used_mem());
224     end_quick_sort_server ();
225   }
226
227   if (program==4)
228   {
229     printf("Servidor quicksort esperando solicitudes.\n");
230     quick_sort_server();
231
232
233   }
234   if (program <= 3)
235   {
236     free(lyst);
237     free(lystbck);
238   }
239
240
241   return 0;
242 }
243
244 void quicksort(int lyst[], int size)
245 {
246   quicksortHelper(lyst, 0, size-1);
247 }
248
249 void quicksortHelper(int lyst[], int lo, int hi)
250 {
251   if (lo >= hi) return;
252   int b = partition(lyst, lo, hi);
253   quicksortHelper(lyst, lo, b-1);
254   quicksortHelper(lyst, b+1, hi);
255 }
256
257 void swap(int lyst[], int i, int j)
258 {
259   int temp = lyst[i];
260   lyst[i] = lyst[j];
261   lyst[j] = temp;
262 }
263
264 int partition(int lyst[], int lo, int hi)
265 {
```

```
266   int b = lo;
267   int r = (int) (lo + (hi-lo + 1)*(1.0*rand()/RAND_MAX));
268   int pivot = lyst[r];
269   swap(lyst, r, hi);
270   for (int i = lo; i < hi; i ++)
271   {
272     if (lyst[i] < pivot)
273     {
274       swap(lyst, i, b);
275       b ++;
276     }
277   }
278   swap(lyst, hi, b);
279   return b;
280 }
281
282 void parallelQuicksort(int lyst[], int size, int tlevel)
283 {
284   int rc;
285   void *status;
286
287   pthread_attr_t attr;
288   pthread_attr_init(&attr);
289   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
290
291   struct thread_data td;
292   td.lyst = lyst;
293   td.low = 0;
294   td.high = size - 1;
295   td.level = tlevel;
296
297   pthread_t theThread;
298   Num_hilos = Num_hilos + 1;
299   rc = pthread_create(&theThread, &attr, parallelQuicksortHelper,
300             (void *) &td);
301   if (rc)
302   {
303         printf("ERROR; return code from pthread_create() is %d\n", rc);
304
305         exit(-1);
306   }
307
308   pthread_attr_destroy(&attr);
309   rc = pthread_join(theThread, &status);
310   if (rc)
311   {
312     printf("ERROR; return code from pthread_join() is %d\n", rc);
313     exit(-1);
314   }
315
316 }
317
318 void *parallelQuicksortHelper(void *threadarg)
319 {
320   int mid, t, rc;
```

```
321   void *status;
322
323   struct thread_data *my_data;
324   my_data = (struct thread_data *) threadarg;
325
326   if (my_data->level <= 0 || my_data->low >= my_data->high)
327   {
328     quicksortHelper(my_data->lyst, my_data->low, my_data->high);
329     pthread_exit(NULL);
330     return NULL;
331   }
332
333   pthread_attr_t attr;
334   pthread_attr_init(&attr);
335   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
336
337   mid = partition(my_data->lyst, my_data->low, my_data->high);
338
339   struct thread_data thread_data_array[2];
340
341   for (t = 0; t < 2; t ++)
342   {
343     thread_data_array[t].lyst = my_data->lyst;
344     thread_data_array[t].level = my_data->level - 1;
345   }
346   thread_data_array[0].low = my_data->low;
347   thread_data_array[0].high = mid-1;
348   thread_data_array[1].low = mid+1;
349   thread_data_array[1].high = my_data->high;
350
351   pthread_t threads[2];
352
353
354   for (t = 0; t < 2; t ++)
355   {
356     if (thread_data_array[t].low < thread_data_array[t].high)
357     {
358       pthread_mutex_lock (&mutexsum);
359       Num_hilos = Num_hilos + 1;
360       pthread_mutex_unlock (&mutexsum);
361
362       rc = pthread_create(&threads[t], &attr, parallelQuicksortHelper,
363                 (void *) &thread_data_array[t]);
364
365       if (rc)
366       {
367             printf("ERROR; return code from pthread_create() is %d\n", rc);
368             exit(-1);
369           }
370     }
371   }
372
373
374   pthread_attr_destroy(&attr);
375   for (t = 0; t < 2; t ++)
```

```
376   {
377     if (thread_data_array[t].low < thread_data_array[t].high)
378     {
379       rc = pthread_join(threads[t], &status);
380       if (rc)
381       {
382         printf("ERROR; return code from pthread_join() is %d\n", rc);
383         exit(-1);
384       }
385     }
386   }
387
388   pthread_exit(NULL);
389 }
390
391
392 int isSorted(int lyst[], int size)
393 {
394   for (int i = 1; i < size; i ++)
395   {
396     if (lyst[i] < lyst[i-1])
397     {
398       printf("Error at loc %d, %d < %d \n", i, lyst[i], lyst[i-1]);
399       return 0;
400     }
401   }
402   return 1;
403 }
404
405 /********************************** distributed with sockets
    ******************************************/
406
407 void* sort_request_to_other_server (void* threadarg_struc) {
408   struct sockaddr_in server_addr;
409   struct hostent *hp;
410   int sd, i;
411   struct thread_data* parameters;
412   int* buffer_int2;
413
414   sd = socket(AF_INET, SOCK_STREAM, 0);
415   hp = gethostbyname (remote_server);
416   bzero((char *)&server_addr, sizeof(server_addr));
417   server_addr.sin_family = AF_INET;
418
419   memcpy (&(server_addr.sin_addr), hp-> h_addr_list[0], hp->h_length);
420   server_addr.sin_port = htons(remote_port);
421
422   parameters = ((struct thread_data *) threadarg_struc);
423   int buff_size = ((parameters->high - parameters->low) +1) + 1; //buffer to order +
    level
424
425   char* buffer_read = (char *) malloc(buff_size*sizeof(int));
426   int* buffer_int = (int *) malloc(buff_size*sizeof(int));
427
428   for (i=0; i <  buff_size-1; i++)
```

```
429   {
430     buffer_int[i] = htonl(parameters->lyst[ parameters->low + i]);
431   }
432
433   buffer_int[buff_size-1] = htonl(parameters->level);
434
435   if (connect(sd, (struct sockaddr *) &server_addr, sizeof(server_addr))< 0)
436   {
437     perror("connect failed. Error");
438     exit(-1);
439   }
440
441   if( send(sd, (char *) &buffer_int[0], buff_size * sizeof(int), 0) < 0) // envía la
      petición
442   {
443     puts("Send failed");
444               exit(-1);
445         }
446
447   if( recv(sd, (char *) buffer_read, buff_size * sizeof(int), 0) < 0) // recibe el
      vector ordenado
448   {
449     puts("recv failed");
450               exit(-1);
451         }
452   buffer_int2 = (int*)buffer_read;
453
454   for (i=0; i <  buff_size-1; i++)
455   {
456       parameters->lyst[parameters->low + i] = ntohl(buffer_int2[i]);
457   }
458
459   close (sd);
460   free(buffer_read);
461   free(buffer_int);
462   pthread_exit(NULL);
463 }
464
465 void* sort_in_this_server(void* sockt_data_id){
466   param_thread_server* par;
467   int* buffer_int;
468   int i, bytes_readed, client_sock;
469   struct thread_data read_data;
470
471   pthread_detach(pthread_self());
472   par = (param_thread_server*) sockt_data_id;
473   client_sock = par->sc;
474   char* buffer= (char *) malloc((Num_elem_vector + 1)*sizeof(int)); //list + level
475
476   bytes_readed=recv(client_sock, (char *) buffer, ((Num_elem_vector +
      1)*sizeof(int)), 0);
477
478   if(bytes_readed < 0)
479   {
480     puts("recv failed");
```

```
481     exit(-1);
482         }
483
484   int buff_size = bytes_readed / sizeof(int);
485
486   read_data.lyst =  (int *) malloc((buff_size - 1)* sizeof(int));;
487   buffer_int = (int *) buffer;
488
489   for (i = 0; i < buff_size - 1; i++)
490   {
491     read_data.lyst[i] = ntohl(buffer_int[i]);
492   }
493
494   read_data.level = ntohl(buffer_int[buff_size-1]);
495   read_data.low = 0;
496   read_data.high =  buff_size-1 -1;
497
498   distributedQuicksort ((void* )&read_data);
499
500   for (i = 0; i < buff_size - 1; i++)
501   {
502     buffer_int[i] = htonl(read_data.lyst[i]);
503   }
504
505   buffer_int[buff_size - 1] = htonl(read_data.level);
506
507   if (send(client_sock, &buffer_int[0],  sizeof(int)*buff_size, 0)< 0)
508   {
509             puts("Send failed");
510             exit(-1);
511         }
512
513   free(buffer);
514   free(read_data.lyst);
515
516   pthread_exit(NULL);
517 }
518
519 void distributedQuicksort(void *threadarg_struc)
520 //this is NOT to be used in a pthread
521 //the whole parameter's list is sorted, the buffer is passed to 2 sub calls with
    indexes adjusted
522 //in the parameters to order the sub parts by sort_request_to_other_server
523 {
524   int mid, t, rc;
525   void *status;
526
527   struct thread_data* my_data;
528   my_data = (struct thread_data *) threadarg_struc;
529
530   if (my_data->level <= 0 || my_data->low >= my_data->high)
531   {
532     quicksortHelper(my_data->lyst, my_data->low, my_data->high);
533     return;
534   }
```

```
535
536   pthread_attr_t attr;
537   pthread_attr_init(&attr);
538   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
539
540   mid = partition(my_data->lyst, my_data->low, my_data->high);
541   struct thread_data thread_data_array[2];
542
543   for (t = 0; t < 2; t ++)
544   {
545     thread_data_array[t].lyst = my_data->lyst;
546     thread_data_array[t].level = my_data->level - 1;
547   }
548   thread_data_array[0].low = my_data->low;
549   thread_data_array[0].high = mid-1;
550   thread_data_array[1].low = mid+1;
551   thread_data_array[1].high = my_data->high;
552
553   pthread_t threads[2];
554
555
556   for (t = 0; t < 2; t ++)
557   {
558     if (thread_data_array[t].low < thread_data_array[t].high)
559     {
560       pthread_mutex_lock (&mutexsum);
561       Num_hilos = Num_hilos + 1;
562       pthread_mutex_unlock (&mutexsum);
563
564       rc = pthread_create(&threads[t], &attr, sort_request_to_other_server,
565                 (void *) &thread_data_array[t]);
566
567       if (rc)
568       {
569             printf("ERROR; return code from pthread_create() is %d\n", rc);
570             exit(-1);
571           }
572     }
573   }
574
575   pthread_attr_destroy(&attr);
576   for (t = 0; t < 2; t ++)
577   {
578     if (thread_data_array[t].low < thread_data_array[t].high)
579     {
580       rc = pthread_join(threads[t], &status);
581       if (rc)
582       {
583         printf("ERROR; return code from pthread_join() is %d\n", rc);
584         exit(-1);
585       }
586       pthread_detach(threads[t]);
587     }
588   }
589   return ;
```

```
590 }
591
592
593 void* quick_sort_server(){
594 //this is to be used in a pthread
595 //server tp be mounted in a phtread to take orders, when one order arrives a new
    pthread is created to solve the request
596   struct sockaddr_in server_addr;
597   struct  sockaddr* client_addr;
598   int socket_desc, client_sock, c, rc;
599   param_thread_server* param_a_servir;
600
601   socket_desc = socket(AF_INET, SOCK_STREAM, 0);
602
603   if (socket_desc == -1)
604   {
605     printf("Could not create socket");
606     exit(-1);
607   }
608
609   server_addr.sin_family = AF_INET;
610   server_addr.sin_addr.s_addr = INADDR_ANY;
611   server_addr.sin_port = htons(local_port);
612
613   if (bind(socket_desc, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0)
614   {
615     perror("bind failed. Error");
616     exit(-1);
617   }
618
619   listen(socket_desc, 100);
620
621   pthread_t thread_id;
622   c = sizeof(struct sockaddr);
623
624   pthread_mutex_lock (&mutex_server_mounted);
625   local_server_mounted = 0;
626   pthread_mutex_unlock (&mutex_server_mounted);
627
628   while(1)
629   {
630     client_addr = malloc(sizeof( struct  sockaddr));
631
632
633
634     client_sock = accept(socket_desc, client_addr, (socklen_t*)&c);
635     pthread_mutex_lock (&mutex_server_mounted);
636
637     if (local_server_mounted != 0)
638     {
639       pthread_mutex_unlock (&mutex_server_mounted);
640       break;
641     }
642     else
643       pthread_mutex_unlock (&mutex_server_mounted);
```

```
644
645
646     param_a_servir = (param_thread_server*)malloc(sizeof(param_thread_server));
647     param_a_servir->sc =  client_sock;
648
649     if( (rc=pthread_create( &thread_id , NULL, sort_in_this_server, (void*)
        param_a_servir)))
650     {
651           printf("ERROR; return code from pthread_create() is %d\n", rc);
652           exit(-1);
653           }
654
655
656   }
657
658   if (client_sock  < 0)
659   {
660     perror("accept has failed");
661     exit(-1);
662   }
663
664   close(socket_desc);
665   pthread_mutex_lock (&mutex_server_mounted);
666   local_server_mounted = -2;
667   pthread_mutex_unlock (&mutex_server_mounted);
668
669   pthread_exit(NULL);
670 }
671
672 void end_quick_sort_server ()
673 {
674 //this is to finish the execution of the server
675   struct sockaddr_in server_addr;
676   struct hostent *hp;
677   int sd;
678
679   sd = socket(AF_INET, SOCK_STREAM, 0);
680   hp = gethostbyname (local_server);
681   bzero((char *)&server_addr, sizeof(server_addr));
682   server_addr.sin_family = AF_INET;
683
684   memcpy (&(server_addr.sin_addr), hp-> h_addr_list[0], hp->h_length);
685   server_addr.sin_port = htons(local_port);
686
687   pthread_mutex_lock (&mutex_server_mounted);
688   local_server_mounted = -1;
689   pthread_mutex_unlock (&mutex_server_mounted);
690
691   if (connect(sd, (struct sockaddr *) &server_addr, sizeof(server_addr))< 0)
692   {
693     perror("connect failed. Error");
694     exit(-1);
695   }
696
697   while (1)
```

```
698   {
699     pthread_mutex_lock (&mutex_server_mounted);
700
701     if (local_server_mounted==-2)
702     {
703       pthread_mutex_unlock (&mutex_server_mounted);
704       break;
705     }
706     else
707       pthread_mutex_unlock (&mutex_server_mounted);
708
709   }
710   close (sd);
711   return;
712 }
713 /*****************************************************************************************
714
715
```

Misc_functions.c

```c
//Hecho por Hugo Mora.
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "misc_functions.h"

/*
gcc -c misc_functions.c -cmisc_functions.o
 */
void
lr_init(struct line_reader *lr, FILE *f)
{
  lr->f = f;
  lr->buf = NULL;
  lr->siz = 0;
}

/*
 Usado para leer la memoria ocuapada por un proceso
 */
char *next_line(struct line_reader *lr, size_t *len)
{
  size_t newsiz;
  int c;
  char *newbuf;

  *len = 0;
  for (;;) {
    c = fgetc(lr->f);
    if (ferror(lr->f))
      return NULL;

    if (c == EOF) {
      if (*len == 0)
        return NULL;
      else
        return lr->buf;
    } else {
      if (*len == lr->siz) {
        newsiz = lr->siz + 4096;
        newbuf = realloc(lr->buf, newsiz);
        if (newbuf == NULL)
          return NULL;
        lr->buf = newbuf;
        lr->siz = newsiz;
      }
      lr->buf[(*len)++] = c;

      if (c == '\n')
        return lr->buf;
    }
  }
}

```

```
56  void
57  lr_free(struct line_reader *lr)
58  {
59    free(lr->buf);
60    lr->buf = NULL;
61    lr->siz = 0;
62  }
63
64  char *strdup (const char *s) {
65      char *d = malloc (strlen (s) + 1);
66      if (d == NULL) return NULL;
67      strcpy (d,s);
68      return d;
69  }
70
71
72
73  long get_my_peak_used_mem(void)
74  {
75    char *vmpeak;
76
77    size_t len;
78    char *line;
79    long result;
80
81
82    FILE *f;
83    struct line_reader lr;
84
85    int p_id;
86    char file_name_pid[128];
87
88    vmpeak = NULL;
89
90    p_id = (int) getpid();
91
92    sprintf(file_name_pid, "/proc/%d/status", p_id);
93    f = fopen(file_name_pid, "r");
94
95    if (!f) return -1;
96
97    lr_init(&lr, f);
98    while (!vmpeak)
99    {
100     if (!(line = next_line(&lr, &len)))
101     {
102       return -1;
103     }
104
105     if (!strncmp(line, "VmPeak:", 7))
106     {
107       vmpeak = strdup(&line[7]);
108     }
109   }
110   free(line);
```

```
111
112   fclose(f);
113
114   len = strlen(vmpeak);
115   vmpeak[len - 4] = 0;
116
117   result = atol(vmpeak);
118
119
120   free(vmpeak);
121
122   return result;
123 }
```

Misc_functions.h

```
1  struct line_reader {
2    FILE  *f;
3    char  *buf;
4    size_t   siz;
5  };
6
7  void
8  lr_init(struct line_reader *lr, FILE *f);
9  char *next_line(struct line_reader *lr, size_t *len);
10 void lr_free(struct line_reader *lr);
11 char *strdup (const char *s);
12 long get_my_peak_used_mem(void);
13
```

7.2.1.2 Multiplicación de matrices

Matmul.c

```
1   //Por Hugo Mora
2
3   //multiplicación de matrices
4   //gcc -lpthread misc_functions.c  matmul.c -omatmul.o
5   //./matmul.o 1 4 2 2 3 2
6
7   #include <pthread.h>
8   #include <stdlib.h>
9   #include <stdio.h>
10  #include "misc_functions.h"
11
12  int num_of_thrd;
13
14  int **A, **B, **C;
15  int NumRowsMatA, NumColsMatAorRowsMatB, NumColsMatB;
16
17  void create_matrix(void **mat, int rows, int cols)
18  {
19    int i, j, val = 0;
20    int **m;
21
22
23    m = (int **) malloc(sizeof(int) * rows);
24    for (i = 0; i < rows; i++)
25        m[i] = (int *) malloc(sizeof(int) * cols);
26
27    for (i = 0; i < rows; i++)
28        for (j = 0; j < cols; j++)
29    {
30          m[i][j] = val++;
31    }
32    *mat = (void *)m;
33  }
34
35  void destroy_matrix(int **m, int rows)
36  {
37    int i, j;
38
39    for (i = 0; i < rows; i++)
40        free(m[i]);
41    free(m);
42  }
43
44  void print_matrix(int **m, int rows, int cols)
45  {
46    int i, j;
47
48    for (i = 0; i < rows; i++) {
49      printf("\n\t| ");
50      for (j = 0; j < cols; j++)
51        printf("%d ", m[i][j]);
52      printf("|");
53    }
54  }
55
```

```
56  void* multiply(void* part)
57  {
58    int s = *((int *) part);
59
60    int from = s * (NumRowsMatA/num_of_thrd);
61    int to = from + (NumRowsMatA/num_of_thrd);
62    int remainder = NumRowsMatA%num_of_thrd;
63
64    if (remainder >0)
65      if (s < remainder)
66      {
67      from = from + s;
68      to = to+s+1;
69      }
70      else
71      {
72      from = from + remainder;
73      to = to + remainder;
74      }
75
76
77    int i,j,k;
78
79    for (i = from; i < to; i++)
80    {
81      for (j = 0; j < NumColsMatB; j++)
82      {
83        C[i][j] = 0;
84        for ( k = 0; k < NumColsMatAorRowsMatB; k++)
85    {
86            C[i][j] += A[i][k]*B[k][j];
87    }
88
89      }
90    }
91    return 0;
92  }
93
94  int main(int argc, char* argv[])
95  {
96    int row1, col1, row2, col2, local_part, rc, program;
97    int* threads_parts;
98
99    struct timeval start, end;
100   double diff;
101
102   pthread_t* threads;  // pointer to a group of threads
103   int i;
104
105   if (argc!=7)
106   {
107     printf("Formato: %s  <tipo algoritmo:1=secuencial,2=paralelo> <row1>
108       <col1> <row2> <col2> <#threads>\n",argv[0]);
109     exit(-1);
110   }
```

```
111   program = atoi(argv[1]);
112   row1 = abs(atoi(argv[2]));
113   col1 = abs(atoi(argv[3]));
114   row2 = abs(atoi(argv[4]));
115   col2 = abs(atoi(argv[5]));
116   if (col1 != row2)
117   {
118     printf("\n Wrong parameters (col1 must be equal to row2).\n");
119     exit(0);
120   }
121   num_of_thrd = abs(atoi(argv[6]));
122
123   if (num_of_thrd > row1)
124     num_of_thrd = row1;
125
126   NumRowsMatA = row1;
127   NumColsMatAorRowsMatB = col1;
128   NumColsMatB = col2;
129
130
131   create_matrix((void **)&A, NumRowsMatA, NumColsMatAorRowsMatB);
132   create_matrix((void **)&B, NumColsMatAorRowsMatB, NumColsMatB);
133   create_matrix((void **)&C, NumRowsMatA, NumColsMatB);
134
135   if (program==2)
136   {
137     threads = (pthread_t*) malloc((num_of_thrd)*sizeof(pthread_t));
138     threads_parts = (int*) malloc((num_of_thrd)*sizeof(int));
139
140     pthread_attr_t attr;
141     pthread_attr_init(&attr);
142     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
143
144     gettimeofday(&start, NULL);
145
146     for (i = 0; i < (num_of_thrd); i++)
147     {
148       threads_parts[i] = i;
149       rc = pthread_create (&threads[i], &attr, multiply, (void*)&threads_parts[i]);
150
151       if (rc !=0)
152       {
153         perror("Can't create thread");
154         free(threads);
155         destroy_matrix(A, NumRowsMatA);
156         destroy_matrix(B, NumColsMatAorRowsMatB);
157         destroy_matrix(C, NumRowsMatA);
158         free(threads_parts);
159         exit(-1);
160       }
161     }
162
163     for (i = 0; i < num_of_thrd; i++)
164       pthread_join (threads[i], NULL);
165     gettimeofday(&end, NULL);
```

```
166   }
167   else
168   {
169     num_of_thrd = 1;
170     threads_parts = (int*) malloc((num_of_thrd)*sizeof(int));
171     threads_parts[0] = 0;
172
173     gettimeofday(&start, NULL);
174     multiply(&threads_parts[0]);
175     gettimeofday(&end, NULL);
176   }
177
178   if (program==2)
179     free(threads);
180   destroy_matrix(A, NumRowsMatA);
181   destroy_matrix(B, NumColsMatAorRowsMatB);
182   destroy_matrix(C, NumRowsMatA);
183   free(threads_parts);
184
185   diff = ((end.tv_sec * 1000000 + end.tv_usec)
186         - (start.tv_sec * 1000000 + start.tv_usec))/1000000.0;
187   if (program==2)
188     printf("Parallel matrix multiplication in c took: %lf sec. and memory= %ld KB\n",
      diff,
189       get_my_peak_used_mem());
190   else
191     printf("Sequential matrix multiplication in c took: %lf sec. and memory= %ld
      KB\n", diff,
192       get_my_peak_used_mem());
193
194   return 0;
195 }
196
```

7.2.1.3 Búsqueda en anchura (BFS)

Bfs.c

```
1   //Por Hugo Mora
2   //gcc -lpthread -std=c99 misc_functions.c adj_mat.c queue_man.c bfs.c -obfs.o
3   //./bfs.o 1 graph100k4.txt 100000 100000 2
4
5   #include <pthread.h>
6   #include <stdlib.h>
7   #include <stdio.h>
8   #include <sys/time.h>
9   #include "adj_mat.h"
10  #include "queue_man.h"
11  #include "misc_functions.h"
12
13  #define NOT_VISITED -1
14
15  int serial_bfs(int n, int source, int max_queue_size);
16  int parallel_bfs(int n, int source, int max_num_threads, int max_quue_size);
17  void* bfs_worker(void* queue_slice);
18
19  int max_q_part_size;
20  int min_q_part_size;
21  int num_max_q_part_size_threads;
22  int num_min_q_part_size_threads;
23
24
25  Queue_type global_queue;
26
27  int *global_distances;
28  int *global_parent_array;
29
30  Adj_Mat *adj_mat;  //matriz de adyacencia
31
32  pthread_mutex_t mutex_bfs; //usado para zona crítica de los hilos
33
34  /*****************************************/
35  int main(int argc, char *argv[]) {
36    char *filename;
37    int nodes;
38    int max_threads, max_queue_size, program;
39
40    struct timeval start, end;
41    double diff;
42
43    if (argc!=6)
44    {
45      printf("Parámetros: <tipo algoritmo:1=secuencial,2=paralelo> <Nombre Archivo>
46        <Número de nodos> <Tamaño máximo de cola> <Número máximo de hilos>.\n");
47      exit(-1);
48    }
49
50    program = atoi(argv[1]);
51    filename = (argv[2]);
52    nodes = atoi(argv[3]);
53    max_queue_size = atoi(argv[4]);
54    max_threads = atoi(argv[5]);
55
```

```
56
57    adj_mat =  read_adj_mat(filename, nodes);
58
59
60    gettimeofday(&start, NULL);
61
62    if (program==2)
63      parallel_bfs(nodes, 0, max_threads, max_queue_size); //souce es siempre el nodo 0
        o inicial
64    else
65      serial_bfs(nodes, 0, max_queue_size); //souce es siempre el nodo 0 o inicial
66
67    deinit_adj_mat(adj_mat);
68
69    gettimeofday(&end, NULL);
70
71    diff = ((end.tv_sec * 1000000 + end.tv_usec)
72          - (start.tv_sec * 1000000 + start.tv_usec))/1000000.0;
73    if (program==2)
74      printf("Parallel BFS in c took: %lf sec. and memory= %ld KB\n", diff,
        get_my_peak_used_mem());
75    else
76      printf("Sequential BFS in c took: %lf sec. and memory= %ld KB\n", diff,
        get_my_peak_used_mem());
77
78    return 0;
79  }
80
81  /*****************************************/
82
83
84  /*****************************************/
85  /*el orden con que quedan puede que no sea igual al del BFS secuencial pero las
86  oleadas si quedan correctamente indicadas en cuanto a la distancia del nodo origen,
87  esto por que al ser paralelo puede que por ejemplo el segundo elemento de la cola de
    una oleada
88  se procese antes que el primer elemento de la cola de esa oleada.*/
89
90  int parallel_bfs(int n, int source, int max_num_threads, int max_queue_size) {
91
92    int distance = 0;
93    int thread_count;
94    int batch_size;
95    int rc;
96    pthread_t* threads;
97    int* threads_slices;
98
99    init_que(&global_queue, max_queue_size);
100
101   global_distances = (int *) malloc(n*sizeof(int));
102   global_parent_array= (int *) malloc(n*sizeof(int));
103
104   for (int i=0; i<n; i++) {
105     global_distances[i] = NOT_VISITED;
106     global_parent_array[i] = NOT_VISITED;
```

```
107   }
108   add_to_queue(&global_queue, source);
109   global_distances[source] = 0;
110   global_parent_array[source] = source;
111
112   pthread_attr_t attr;
113   pthread_attr_init(&attr);
114   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
115
116
117   while (global_queue.actual_size != 0) {
118     distance++;
119     batch_size = global_queue.actual_size;
120
121     if (max_num_threads < batch_size) {
122        thread_count = max_num_threads;
123     }
124     else {
125        thread_count = batch_size;
126     }
127
128     //cuantos nodos se deja cada hilo de la cola, puede que unos hilos tengan un nodo
129     //mas que otros si no es exacta la div de nodos en la cola con los hilos
130     min_q_part_size = batch_size/thread_count;
131     max_q_part_size = min_q_part_size + 1;
132
133     //estimo cuantos hilos tendran mas nodos que otros
134     num_max_q_part_size_threads = batch_size - (thread_count * min_q_part_size);
135
136     //estimo cuantos hilos tendran menos nodos que otros
137     num_min_q_part_size_threads = thread_count - num_max_q_part_size_threads;
138
139     threads = (pthread_t*) malloc((thread_count)*sizeof(pthread_t));
140     threads_slices = (int*) malloc((thread_count)*sizeof(int));
141
142     for (int i=0; i<(thread_count); i++) {
143       threads_slices[i] = i;
144       rc = pthread_create(&threads[i], &attr, bfs_worker,
145             (void *) &threads_slices[i]);
146       if (rc !=0)
147       {
148         perror("Can't create thread");
149         free(threads);
150         deinit_que(&global_queue);
151         exit(-1);
152       }
153     }
154
155     for (int i = 0; i < thread_count; i++)
156       pthread_join (threads[i], NULL);
157
158     free(threads);
159     free(threads_slices);
160     //hasta que los hilos terminen con su trabajo se toca la cola para quitar todos
      los nodos ya procesados
```

```
161     queue_pop_many(&global_queue, batch_size);
162
163   }
164
165   deinit_que(&global_queue);
166
167   return 0;
168 }
169 /****************************************/
170 //trabajador montado en un hilo pthread
171 void* bfs_worker(void* queue_slice) {
172
173 int chunk_size;
174 int start_index;
175 int parent;
176 int child;
177 int int_queue_slice;
178 int distance;
179
180
181   int_queue_slice =  *((int *)queue_slice);
182
183   //cálculo en donde arranca la parte de nodos para este hilo
184   if (int_queue_slice < num_max_q_part_size_threads) {
185      chunk_size = max_q_part_size;
186      start_index = int_queue_slice*max_q_part_size;
187   }
188   else {
189     chunk_size = min_q_part_size;
190     start_index = num_max_q_part_size_threads*max_q_part_size + (int_queue_slice-
191       num_max_q_part_size_threads)*min_q_part_size;
192   }
193
194   for (int j=start_index; j<start_index+chunk_size; j++) {
195     parent = read_queue_pos(&global_queue, j); //leo el nodo de la cola
196     distance = global_distances[parent] + 1;
197     for (int k=1; k < adj_mat->adj_list[parent][0]; k++) { //para todos los nodos
      adyacentes a este nodo
198       child = adj_mat->adj_list[parent][k]; //lee este nodo adyacente
199       pthread_mutex_lock (&mutex_bfs);
200       if (global_distances[child] == NOT_VISITED || global_distances[child] >
        distance) {
201          add_to_queue(&global_queue, child); //ponerlo de ultimo en la cola
202          global_distances[child] = distance;
203          global_parent_array[child] = parent;
204
205       }
206       pthread_mutex_unlock (&mutex_bfs);
207      }
208   }
209
210   return 0;
211 }
212
213
```

```
214 /*****************************************/
215 /*bfs secuencial*/
216 int serial_bfs(int n, int source, int max_queue_size) {
217
218   Queue_type global_queue;
219   int parent, child;
220   int distance = 0;
221
222   global_distances = (int *) malloc(n*sizeof(int));
223   global_parent_array= (int *) malloc(n*sizeof(int));
224
225   init_que(&global_queue, max_queue_size);
226
227   for (int i=0; i<n; i++) {
228     global_distances[i] = NOT_VISITED;
229      global_parent_array[i] = NOT_VISITED;
230   }
231
232   add_to_queue(&global_queue, source);
233
234   global_distances[source] = 0;
235   global_parent_array[source] = source;
236
237   while (global_queue.actual_size != 0) {
238     parent = queue_pop(&global_queue); //aquí se puede imprimir el nodo si se quiere
239     distance = global_distances[parent] + 1;
240     for (int i = 1; i < adj_mat->adj_list[parent][0]; i++) {
241      child = adj_mat->adj_list[parent][i]; //deme el nodo conectado al parent
242      if (global_distances[child] == NOT_VISITED || global_distances[child] >
       distance) {
243
244       add_to_queue(&global_queue, child); //ponerlo de último en la cola
245       global_distances[child] = distance; //indicar que es visitado al ponerle una
        distancia
246       global_parent_array[child] = parent;
247
248      }
249     }
250   }
251   return 0;
252 }
253 /*****************************************/
254
```

Adj_mat.c

```c
//Por Hugo Mora
//gcc -std=c99 adj_mat.c

#include <stdlib.h>
#include <stdio.h>
#include "adj_mat.h"
#include <string.h>

// Leer la lista de adyacencia de un archivo, el formato del archivo es parecido al de la
//lista de adyacencia por cada linea, el número es el fuente y los que le sigen separados
//por punto y coma son los destinos a los que se puede llegar desde ese nodo
Adj_Mat *read_adj_mat(char *filename, int n) {
  Adj_Mat *adj_mat;
  char line [10000];
  char *token;
  int source_node, dest_node;

  adj_mat = (Adj_Mat*) malloc(sizeof (Adj_Mat));
  int edge_count = 0;
  adj_mat->n = n;
  adj_mat->directed = 0;

  init_adj_mat(adj_mat, n);

  FILE *file = fopen ( filename, "r" );
  if (file != NULL) {
    while(fgets(line,sizeof line,file)!= NULL) {
      token = (char*) strtok (line, ";");
      source_node = atoi(token);

      token = (char*) strtok(NULL, ";");
      while (token != NULL)
      {
        dest_node=atoi(token);
        if (source_node != dest_node ) { //no se admite arcos a si mismo
          if (!is_linked(adj_mat, source_node, dest_node)) {
            insert_list_edge(adj_mat, source_node, dest_node);
            edge_count++;
          }
          //el archivo debe tener el arco en sentido opuesto también, pero aun asi
          //mejor lo agrego de una vez por que el achivo puede tener algun error
          if (!is_linked(adj_mat, dest_node, source_node) && !(adj_mat->directed))
             insert_list_edge(adj_mat, dest_node, source_node);
        }
        else
          printf("Ciclo en nodo: %d.\n", source_node);

        token = (char*) strtok(NULL, ";");
      }
    }
    fclose(file);
    adj_mat->edge_count = edge_count;
  }
```

```
54    else {
55      perror(filename);
56    }
57
58    return adj_mat;
59  }
60
61  /*****************************/
62  //inicializa el arreglo de listas de adyacencia de la matriz recibida
63  void init_adj_mat(Adj_Mat *adj_mat, int n) {
64    adj_mat->adj_list = (int **) malloc(n*sizeof(int *));
65    for (int i=0; i < n; i++) {
66      adj_mat->adj_list[i] = (int *) malloc(sizeof(int));
67      adj_mat->adj_list[i][0] = 1;
68    }
69  }
70
71  /*****************************/
72  //vacea el arreglo de listas de adyacencia de la matriz recibida
73  void deinit_adj_mat(Adj_Mat *adj_mat) {
74     int n = adj_mat->n;
75
76     for (int i=0; i<n; i++)
77       free(adj_mat->adj_list[i]);
78    free(adj_mat->adj_list);
79
80  }
81
82  /*****************************/
83  //inserta un arco dirigido del nodo #u al nodo #v en
84  //el arreglo de listas de adyacencia de la matriz recibida
85  void insert_list_edge(Adj_Mat *adj_mat, int u, int v) {
86    int old_list_size = adj_mat->adj_list[u][0];
87    int new_list_size = old_list_size + 1;
88    int *new_edge_list = (int *) malloc(new_list_size*sizeof(int));
89
90    for (int i=0; i<old_list_size; i++)
91       new_edge_list[i] = adj_mat->adj_list[u][i];
92    new_edge_list[old_list_size] = v;
93    adj_mat->adj_list[u] = new_edge_list;
94    adj_mat->adj_list[u][0] = new_list_size;
95  }
96
97  /*****************************/
98  /* Indica si un el nodo #u tiene un arco al nodo #v, devuelve 1 si lo hay, cero si no
    lo hay*/
99  int is_linked(Adj_Mat *adj_mat, int u, int v) {
100    int **adj_list = adj_mat->adj_list;
101    int contains = 0;
102
103   int degree = (adj_list[u][0])-1;
104   for (int i=1; i <= degree; i++) {
105       if (adj_list[u][i]==v){
106       contains = 1;
107       break;
```

```
108     }
109   }
110   return contains;
111 }
112
113 /*****************************/
114 /*imprime la lista de adyacencia para todos los nodos*/
115 void print_adj_mat(Adj_Mat *adj_mat, int n) {
116    int **adj_list = adj_mat->adj_list;
117
118   printf("Adj list: ");
119   for (int i = 0; i < n; i++) {
120      printf("\nnode %d", i);
121      //la columna tiene la cantidad de arcos de ese nodo
122      int vertex_degree = adj_list[i][0] - 1;
123      printf(", num vert %d: ", vertex_degree);
124      for (int j = 1; j <= vertex_degree; j++) {
125        printf("%d ", adj_list[i][j]);
126     }
127   }
128   printf("\n");
129 }
130 /*****************************/
131
```

Adj_mat.h

```
1  //Por Hugo Mora
2  typedef struct {
3     int n;
4     int edge_count;
5     int directed;
6     int **adj_list;
7  } Adj_Mat;
8
9  Adj_Mat *read_adj_mat(char *filename, int n);
10 void init_adj_mat(Adj_Mat *adj_mat, int n);
11 void deinit_adj_mat(Adj_Mat *adj_mat);
12 void insert_list_edge(Adj_Mat *adj_mat, int u, int v);
13 int is_linked(Adj_Mat *adj_mat, int u, int v);
14 void print_adj_mat(Adj_Mat *adj_mat, int n);
15
```

Queue_man.c

```c
//Por Hugo Mora
/*este es un manejador de colas fifo*/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "queue_man.h"

void init_que(Queue_type *q, int maximun_size) {
  q->maximun_size = maximun_size;
  q->actual_size = 0;
  q->first_mem_pos = (int *) malloc(sizeof(int));
  q->first_elem_pos = 0;
}

void deinit_que(Queue_type *q) {
  q->maximun_size = 0;
  q->actual_size = 0;
  free(q->first_mem_pos);
  q->first_elem_pos = 0;
}

void add_to_queue(Queue_type *q, int item) {

  if ((q->first_elem_pos + q->actual_size + 1) <= q->maximun_size) {
    if ((q->first_elem_pos + q->actual_size) > 1)
      q->first_mem_pos = realloc(q->first_mem_pos,
        (q->first_elem_pos+q->actual_size + 1) * sizeof(int));
    q->first_mem_pos[(q->first_elem_pos+q->actual_size) % q->maximun_size] = item;
    q->actual_size++;
  }
  else {
    printf( "Queue is full.\n", item);
  }

}

int queue_pop(Queue_type *q) {
  int item = -1;
  if (q->actual_size > 0) {
    item = q->first_mem_pos[q->first_elem_pos];
    q->first_elem_pos = (q->first_elem_pos+1) % q->maximun_size; //ciclica
    q->actual_size--;
  }
  else
    printf( "Queue with no items.\n");
  return item;
}

/*vacean una parte de la cola pero no retorna el pedazo vaceado*/
void queue_pop_many(Queue_type *q, int chunk_size) {
   if (q->actual_size >= chunk_size) {
    q->first_elem_pos = (q->first_elem_pos+chunk_size) % q->maximun_size;
    q->actual_size = q->actual_size - chunk_size;
   }
```

```
56   else
57     printf( "Can not pop %d items from a queue with only %d elements.\n", chunk_size,
       q->actual_size);
58 }
59
60 int read_queue_pos(Queue_type *q, int index) {
61    if (index >= 0 && index < q->actual_size) {
62      return q->first_mem_pos[(q->first_elem_pos + index) % q->maximun_size];
63   }
64   else {
65      printf( "Queue index %d is out of limits.\n", index);
66     return -1;
67   }
68 }
69
70 void print_queue(Queue_type *q) {
71   printf("Queue_type: ");
72   for (int i=0; i<q->actual_size; i++) {
73     printf("%d ", (q->first_mem_pos[(q->first_elem_pos+i) % q->maximun_size]));
74   }
75   printf("\n");
76 }
77
78
```

Queue_man.c

```
1  //Por Hugo Mora
2
3  typedef struct {
4    int maximun_size;
5    int actual_size;
6    int *first_mem_pos;
7    int first_elem_pos;
8  } Queue_type;
9
10 void init_que(Queue_type *q, int maximun_size);
11 void deinit_que(Queue_type *q);
12
13 void add_to_queue(Queue_type *q, int item);
14 int queue_pop(Queue_type *q);
15 void queue_pop_many(Queue_type *q, int chunk_size);
16
17 int read_queue_pos(Queue_type *q, int index);
18 void print_queue(Queue_type *q);
19
20
```

7.2.1.4 Generador de grafos para el (BFS)

Graph_gen_print.c

```
1  //Por Hugo Mora
2  //Imprime la lista de adyacencia del grafo aleatorio conectado, cada conección se
   supone bidireccional
3  //y por tanto no es necesario imprimir la conección inversa del nodo destino hacia el
   nodo fuente.
4
5  //gcc graph_gen.c -ograph_gen.o
6  //./graph_gen.o 100000 4 > graph100k4.txt
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <time.h>
11
12 int main(int argc, char* argv[]){
13
14   int numberOfVertices, prob_limit, prob, expected_conecc;
15   int vertexCounter,edgeCounter;
16
17   if (argc!=3){
18     printf("Uso: %s <#nodos (1 - N)> <promedio de nodos conectados por nodo
       (0-#nodos)> \n",argv[0]);
19     exit(-1);
20   }
21
22   numberOfVertices = abs(atoi(argv[1]));
23   expected_conecc = abs(atoi(argv[2]));
24
25   if( numberOfVertices == 0){
26     printf("Uso: %s <#nodos (1 - N)> <promedio de nodos conectados por nodo
       (0-#nodos)> \n",argv[0]);
27     exit(-1);
28   }
29   printf("Total Nodos = %d, promedio de nodos conectados por nodo =
     %d\n",numberOfVertices, expected_conecc);
30
31   srand ( time(NULL) );
32   prob_limit = RAND_MAX * (expected_conecc * 1.0/numberOfVertices);
33
34   printf("[prob conecc cada nodo= %d / %d]\n", prob_limit, RAND_MAX);
35   vertexCounter = 0;edgeCounter = 0;
36   for (vertexCounter = 0; vertexCounter < numberOfVertices; vertexCounter++){
37     printf("%d",vertexCounter);
38     for (edgeCounter=0; edgeCounter < numberOfVertices; edgeCounter++){
39       if (edgeCounter != vertexCounter){ //no se permiten conecciones a si mismo
40         prob = rand();
41         if (prob <= prob_limit)
42           printf(";%d", edgeCounter);
43       }
44     }
45     printf("\n");
46   }
47   return 0;
48 }
49
```

7.2.2 Programa de pruebas en lenguaje Erlang

7.2.2.1 Quicksort

Tesis_qsort.erl

```
%Por Hugo Mora
%erl -compile tesis_qsort
%erl -sname b -noshell -run tesis_qsort main 1 20000 a@pc b@pc -s init stop

-module(tesis_qsort).
-export([qsort/1, qsort_and_metrics/1, main_internal/4, main/1, p_qsort/1,
split/3,merge/1, dist_qsort/3, qwait/0, peval/4, distributed_qsort_and_metrics/3]).


%%%%%%%%%%%% normal quicksort
qsort([]) ->
    [];
qsort([H | T]) ->
    qsort([ X || X <- T, X < H ]) ++ [H] ++ qsort([ X || X <- T, X >= H ]).

qsort_and_metrics(Lista) ->
  T1 = now(),
  Lr = qsort(Lista),
  T2 = now(),
  [Time|_] = io_lib:format("~.6f", [timer:now_diff(T2, T1)/1000000]),
  {Lr, Time, misc_functions:get_my_peak_used_mem()}.



%%%%%%%%%%%% paralel quicksort
p_qsort(L) ->
  Self = self(),
  Ref = erlang:make_ref(), % make an unique id
  spawn(fun() ->
  split(L, Self, Ref)
  end),
  merge(Ref).

split([], Parent, Ref) ->
  Parent ! {Ref, 0, []};

split([C|[]], Parent, Ref) ->
  Parent ! {Ref, 0, [C]};

split([C|[C2|[]]], Parent, Ref) ->
  if
    C =< C2 -> Parent ! {Ref, 0, [C|[C2]]};
    true -> Parent ! {Ref, 0, [C2|[C]]}
  end;


split([Pivot | T], Parent, Ref) ->
  Self = self(),
  Ref1 = erlang:make_ref(),
  Ref2 = erlang:make_ref(),
  spawn(fun() ->
      split([X || X <- T, X < Pivot], Self, Ref1)
    end),
  spawn(fun() ->
      split([X || X <- T, not (X < Pivot)], Self, Ref2)
```

```
56      end),
57    TupIzq = merge(Ref1),
58    TupDer = merge(Ref2),
59    Parent ! {Ref, element(1, TupIzq) + element(1, TupDer) + 2,
60      element(2, TupIzq) ++ [Pivot] ++ element(2, TupDer)}.
61
62  merge(Ref) ->
63    receive
64      {Ref, Num_hijos, Value} ->
65        {Num_hijos, Value}
66    end.
67
68  parallel_qsort_and_metrics(Lista) ->
69    T1 = now(),
70    Lr = p_qsort(Lista),
71    T2 = now(),
72    [Time|_] = io_lib:format("~.6f", [timer:now_diff(T2, T1)/1000000]),
73    {Lr, Time, misc_functions:get_my_peak_used_mem()}.
74
75  %%%%%%%%% distributed quicksort
76  %%tesis_qsort:dist_qsort([3,2,1,566,43,22,453,32],a@pc, b@pc)
77
78  dist_qsort([], _, _) -> [];
79  dist_qsort([Pivot|Rest], Left_node, Right_node) ->
80      Left = [X || X <- Rest, X < Pivot],
81      Right = [Y || Y <- Rest, Y >= Pivot],
82      peval(Left, Left_node, Right_node,Left_node) ++ [Pivot] ++ peval(Right,
      Left_node,
83        Right_node, Right_node).
84
85
86  peval(Lista, Left_node, Right_node, Node) ->
87      Pid = spawn(Node, tesis_qsort, qwait, []),
88      Pid ! {self(), Lista, Left_node, Right_node},
89      receive
90          {Pid, R} -> R
91      end.
92
93  qwait() ->
94      receive
95          {From,Lista, Left_node, Right_node} ->
96              From ! {self(), dist_qsort(Lista, Left_node, Right_node)};
97        finished ->
98              io:format("qwait finished~n", [])
99      end.
100
101 distributed_qsort_and_metrics(Lista, Nodo1, Nodo2) ->
102   T1 = now(),
103   Lr = dist_qsort(Lista, Nodo1, Nodo2),
104   T2 = now(),
105   [Time|_] = io_lib:format("~.6f", [timer:now_diff(T2, T1)/1000000]),
106   {Lr, Time, misc_functions:get_my_peak_used_mem()}.
107
108 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
109
```

```
110 main_internal(Program, NumElems, Nodo1, Nodo2) ->
111   {A1,A2,A3} = now(),
112   random:seed(A1, A2, A3),
113   List=[random:uniform(134217728) || _ <- lists:seq(1, NumElems)], %integer = 32 bits
114   case Program of
115     1 ->  {NList, Time, Used_mem} = qsort_and_metrics(List),
116       Is_ordered = misc_functions:is_ordered(NList),
117       if Is_ordered ->
118           io:format("Sequential quicksort in erlang took: ~s sec. and memory = ~w
          KB~n", [Time, Used_mem]);
119         true -> io:format("List is NOT ordered")
120       end;
121     2 ->  {Threads_and_NList, Time, Used_mem} = parallel_qsort_and_metrics(List),
122       {NThreads, NList}= Threads_and_NList,
123       Is_ordered = misc_functions:is_ordered(NList),
124       if
125         Is_ordered ->
126           io:format("Parallel quicksort in erlang took: ~s sec.  # threads= ~w and
          memory = ~w KB~n",
127           [Time, NThreads, Used_mem]);
128         true -> io:format("List is NOT ordered")
129       end;
130     3 ->  {NList, Time, Used_mem} = distributed_qsort_and_metrics(List, Nodo1,
      Nodo2),
131       Is_ordered = misc_functions:is_ordered(NList),
132       if
133         Is_ordered ->
134           io:format("Distributed quicksort in erlang took: ~s sec. and memory = ~w
          KB~n", [Time, Used_mem]);
135         true -> io:format("List is NOT ordered")
136       end;
137
138
139     _Else -> io:format("Uso: erl -noshell -run -sname <nombre nodo local> tesis_qsort
      main
140       <tipo algoritmo:1=quicsort secuencial, 2=quisort paralelo, 3=quicksort
        distribuido>  <num elementos>
141       <Nombre nodo local> <Nombre nodo remoto> -s init stop")
142   end.
143
144 main(ParamList) -> %[Program, NumElems, Nodo1, Nodo2]
145   Prog_int = list_to_integer(lists:nth(1, ParamList)),
146   NumElems_int = list_to_integer(lists:nth(2, ParamList)),
147   Nodo1 = list_to_atom(lists:nth(3, ParamList)),
148   Nodo2 = list_to_atom(lists:nth(4, ParamList)),
149   main_internal(Prog_int, NumElems_int, Nodo1, Nodo2).
150
```

Misc_functions.erl

```
%Por Hugo Mora
%Usado para leer la memoria ocuapada por un proceso
%erl -compile misc_functions
% erl -noshell -s misc_functions get_my_peak_used_mem -s init stop

-module(misc_functions).
-export([readlines/1, find_line/2, get_my_peak_used_mem/0, is_ordered/1]).

%to get a list of lines of the file
readlines(FileName) ->
    {ok, Device} = file:open(FileName, [read]),
    try get_all_lines(Device)
      after file:close(Device)
    end.

get_all_lines(Device) ->
    case io:get_line(Device, "") of
        eof  -> [];
        Line -> [Line |get_all_lines(Device)]
    end.

% to get the firts line with a word in a list of strings
find_line([], _) ->
    "";

find_line([C|R], Word) ->
    Pos=string:str(C, Word),
    if
      Pos >= 1 -> C;
      true -> find_line(R, Word)
    end.

get_kb_number(Line) ->
  Num=string:sub_word(Line, 2),
  if
    Num == "" ->
      0;
    true ->
      {IntNum, _} = string:to_integer(Num),
      IntNum
  end.

%Get the peak used memory of the current process
get_my_peak_used_mem() ->
    Proc=os:getpid(),
    Lines=readlines("/proc/" ++ Proc ++"/status"),
    Line=find_line(Lines, "VmPeak:"),
    get_kb_number(Line).

%testing if a list is ordered
is_ordered([]) -> true;
is_ordered([_C|[]]) -> true;

is_ordered([C|[C2|R]]) ->
    if
```

```
56      C =< C2 -> is_ordered([C2|R]);
57
58      true -> false
59    end.
```

7.2.2.2 Multiplicación de matrices

Matmul.erl

```
%Por Hugo Mora
% multiplicación de matrices
%erl -compile matmul

-module(matmul).

-export([mat_mul_parallel/4, mat_mul_seq/3, allowed_workers/2, worker/3, getResults/1, getResults/3, matrix_from_list_matrixes/2,
    matrix_create/2, matrix_create/3, mat_mul/2, matrix_show/1, matrix_show/2,
    matrix_getXY/3, num_rows_to_process/3,
    matrix_parts/2, main_internal/5, main/1]).

%%%%%%%%%%%%%%%
%Multiplica 2 matrices, NumProcs = número de procesos que se desea que abrir, si la primera matriz
%tiene menos filas que el número de procesos deseado entonces el número de procesos deseado se diminuye
%automaticamente para procesar una fila por proceso de la prmera matriz, si la cantidad de filas
%dividido entre la cantidad de procesos no da un número entero entonces a los primeros procesos
%se les asigna una fila mas hasta agotar el remanente.

mat_mul_parallel(NumProcs, A_rows, A_cols, B_cols) ->
    A = matrix_create(A_rows, A_cols),
    B = matrix_create(A_cols, B_cols),
    T1 = now(),
    NumWorkers = allowed_workers(A, NumProcs),
    RowDistributions = matrix_parts(A, NumWorkers),
    Workers = [
        spawn(fun() -> worker(SubMat, B, NumWorker) end)
            ||
        {NumWorker, SubMat} <- lists:zip(lists:seq(1, NumWorkers), RowDistributions)
    ],
    [Worker ! {calc, self()} || Worker <- Workers], %envía la orden de calcular
    Result = getResults(NumProcs),
    T2 = now(),
    [Time|_] = io_lib:format("~.6f", [timer:now_diff(T2, T1)/1000000]),
    {Result, Time}.

mat_mul_seq(A_rows, A_cols, B_cols) ->
    A = matrix_create(A_rows, A_cols),
    B = matrix_create(A_cols, B_cols),
    T1 = now(),
    Result = mat_mul(A, B),
    T2 = now(),
    [Time|_] = io_lib:format("~.6f", [timer:now_diff(T2, T1)/1000000]),
    {Result, Time}.

%%%%%%%%%%%%%%%
%%%%%%%% como máximo retorna el número de workers igual al número
%de filas de la matriz, si NumProcs no es mayor al número de filas retorna entonces NumProcs
%NumProcs=número de procesos como máximo que se desea abrir.
allowed_workers(Matrix, NumProcs) ->
```

```
49      Rows = length(Matrix),
50      if
51          NumProcs > Rows ->
52              Rows;
53          true ->
54              NumProcs
55      end.
56
57  %%%%%%%%%%%%%%%%%%%%%%%%%%%%
58  %% Usado para invocar el ordenado de cada submtriz con la segunda matriz entera
59  worker(A, B, Id) ->
60      receive
61          {calc, Parent} ->
62              Result = mat_mul(A, B),
63              Parent ! {done, Result, Id, self()}
64      end.
65
66  %%%%%%%%%%%%%%%%%%%%%%%%%%%%
67  %%%Concatena los resultados
68
69  getResults(NumProcs) ->
70    L_results = [[[]] || _ <- lists:seq(0, NumProcs - 1)],
71        getResults(NumProcs, NumProcs, L_results).
72
73  getResults(0, _NumProcs, Results) ->
74        N_Results = matrix_from_list_matrixes(Results, []),
75        N_Results;
76
77  getResults(ProcRestantes, NumProcs, Results) ->
78      receive
79          {done, Result, IdRemote, _From} ->
80            N_results = if (IdRemote == NumProcs)
81          -> lists:sublist(Results,IdRemote - 1) ++ [Result];
82          true -> if (IdRemote == 1)
83              -> [Result] ++ lists:nthtail(IdRemote,Results);
84              true -> lists:sublist(Results,IdRemote - 1) ++ [Result] ++
85                lists:nthtail(IdRemote,Results)
86             end
87                      end,
88          getResults(ProcRestantes - 1, NumProcs, N_results)
89      end.
90
91  %%%%%%%%%%%%%%%%%%%%%%%%%%%%
92
93  %dada una lista de matrices la transforma en una matriz
94  matrix_from_list_matrixes([], Result) ->
95      Result;
96  matrix_from_list_matrixes([H|R], Result) ->
97      matrix_from_list_matrixes(R, Result ++ H).
98
99
100 %%%%%%%%%%%%%%%%
101 %asigna las matrices pero con números que no son aleatorios
102 matrix_create(Rows, Columns) ->
103     matrix_create(Rows, Columns, []).
```

```
104
105 matrix_create(0, _, Matrix) ->
106     Matrix;
107 matrix_create(Rows, Columns, Matrix) ->
108     L = [(Rows-1) * Columns + X || X <- lists:seq(0, Columns - 1)],
109     matrix_create(Rows - 1, Columns, [L | Matrix]).
110
111 %%%%%%%%%%%%%%%%%
112 %%%% Multiplica 2 matrices, el # de columnas de la matriz A debe ser igual al número
    de filas de las matriz B
113
114 mat_mul(A, B) ->
115     A_cols = lists:seq(1, length(hd(A))),
116     A_rows = lists:seq(1, length(A)),
117     B_cols = lists:seq(1, length(hd(B))),
118     [
119         [
120             lists:sum([matrix_getXY(A, K, J) * matrix_getXY(B, J, I) || J <- A_cols])
121                 ||
122             I <- B_cols
123         ]
124             ||
125         K <- A_rows
126     ].
127
128 %%%%%%%%%%%%%%%%%
129 %%%%%%%%Imprime la matriz
130
131 matrix_show(Matrix) ->
132     matrix_show(Matrix, "~p ").
133
134 matrix_show(Matrix, Format) ->
135     Rows = [fun() -> [io:format(Format, [Item]) || Item <- Row], io:format("~n") end
    || Row <- Matrix],
136                             %cada función imprime una fila de la matriz
137     [Row() || Row <- Rows]. %ejecuta cada una de las funciones de la lista de
    funciones
138
139 %%%%%%%%%%%%%%%%%%%%%%%%%%%
140 %%%%%%% para obtener el Item de la matriz en la fila I y columna J
141 matrix_getXY(M, I, J) ->
142     lists:nth(J, lists:nth(I, M)).
143
144 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
145 %le asigna una fila mas a los werkers en caso de que las filas totales no calcen de
    manera
146 %exacta (al repartir) con los workers totales
147 %esa sobreasignación se da hasta agotar el remanente de filas.
148 num_rows_to_process(RowsTotal, ProcNum, NumProcs) ->
149     RowsBase = RowsTotal div NumProcs,
150     Rows_Rem= RowsTotal rem NumProcs,
151     if
152         ProcNum < Rows_Rem->
153             RowsBase + 1;
154         true ->
```

```
155             RowsBase
156     end.
157
158 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
159 %parte la matriz, cada parte tiene un número de filas continuas,
160 %cada una de estas partes le va a tocar a un worker,
161 %devuelve las particiones junto con una lista de rows id, eso row id lo que indica es
    la id
162 %de la última fila de la partición dentro de la matriz,
163 %por ejemplo, si la id es la 6, indica que en la matriz esa partición termina en la
    fila 6
164 %(considerando numeración de filas de 1 a N)
165 matrix_parts(Matrix, NumWorkers) ->
166     RowsNum = length(Matrix),
167     RowDistributions = [num_rows_to_process(RowsNum, WorkerNum,
168       NumWorkers) || WorkerNum <- lists:seq(0, NumWorkers - 1)],
169     matrix_parts(Matrix, RowDistributions, []).
170
171 matrix_parts(Matrix, [Size | RowDistributions], Parts) ->
172     {Part, Remainder} = lists:split(Size, Matrix),
173     matrix_parts(Remainder, RowDistributions, [Part | Parts]);
174
175 matrix_parts([], [], Parts) ->
176     lists:reverse(Parts).
177 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
178
179 main_internal(Program, NumProcs, A_rows, A_cols, B_cols) ->
180      case Program of
181       1 -> {_ResultMatrix, Time} = mat_mul_seq(A_rows, A_cols, B_cols),
182          Used_mem = misc_functions:get_my_peak_used_mem(),
183          io:format("Sequential matrix multiplication in erlang took: ~s sec. and
           memory = ~w KB~n", [Time, Used_mem]);
184       2 -> {_ResultMatrix, Time} = mat_mul_parallel(NumProcs, A_rows, A_cols,
          B_cols),
185          Used_mem = misc_functions:get_my_peak_used_mem(),
186          io:format("Parallel matrix multiplication in erlang took: ~s sec. and memory
           = ~w KB~n", [Time, Used_mem]);
187       _Else -> io:format("Uso: erl -noshell -run matmul main <tipo
          algoritmo:1=Multiplicación matrices secuencial,
188                   2=Multiplicación matrices paralelo> <columnas matriz 1> <filas matriz
                      1>
189                   <columnas matriz 2> -s init stop\n", [])
190      end.
191
192 main(ParamList) -> %[Program, NumProcs, A_rows, A_cols, B_cols]
193   Program = list_to_integer(lists:nth(1, ParamList)),
194   NumProcesos = list_to_integer(lists:nth(2, ParamList)),
195   A_rows = list_to_integer(lists:nth(3, ParamList)),
196   A_cols = list_to_integer(lists:nth(4, ParamList)),
197   B_cols = list_to_integer(lists:nth(5, ParamList)),
198   main_internal(Program, NumProcesos, A_rows, A_cols, B_cols).
199
```

7.2.2.3 Búsqueda en anchura (BFS)

Bfs.erl

```
%Por Hugo Mora
%se maneja una lista de adjacencia la cual es una lista de conjuntos indexada por número de nodo
%en donde los nodos son enumerados del 0 al N-1 siendo N el número de nodos, en el archivo

 -module(bfs).
-export([create_ad_ma/2, ad_ma_read_each_file_line/2, ad_ma_process_text_line/2, ad_ma_add_XY/3, ad_ma_add_List_XY/2, ad_ma_print/2, printArr/2,
  ad_ma_printTmp/2, printArrTmp/2, updateDist/3, updateParentsList/3,
  updateParents/3, update_parent_array_using_array_index_list/3, seq_stage/6,
  sequential_bfs/2, bfs_worker/2, await_responses/3, orderWork/4,
  synchronizer/6,processCoord/3, start_parallel/3, start_seq/2, main_internal/4,
  main/1]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%este bloque es el que lee el archivo de coordenadas y llena la matriz de adjacencia recivida
create_ad_ma(FileName, Num_Nodes) ->
  {ok, Device} = file:open(FileName, [read]),
  AdjMat = array:new([{size,Num_Nodes},{fixed,true},
    {default,ordsets:new()}]),

  NewAdjMat = ad_ma_read_each_file_line(Device, AdjMat),
  file:close(Device),
      NewAdjMat.

ad_ma_read_each_file_line(Device, AdjMat) ->
    case io:get_line(Device, "") of
        eof  -> file:close(Device),
            AdjMat;
        Line ->  Line_no_spaces = string:strip(Line),
           N_AdjMat = ad_ma_process_text_line(Line_no_spaces, AdjMat),
                ad_ma_read_each_file_line(Device,N_AdjMat)
    end.

ad_ma_process_text_line(Line, AdjMat) ->
  case string:len(Line) of
    0 -> AdjMat;
    1 -> AdjMat; %puede que la linea solo tenga el caracter de retorno por ser una linea vacía
    _ ->
      L_nums = lists:map(fun(X) -> {Int, _} = string:to_integer(X),
            Int end, string:tokens(Line, ";")),
      ad_ma_add_List_XY(L_nums, AdjMat)

  end.

%monta el vertice de X a Y y de Y a X en la matriz de adjacencia recivida como parámetro
%las coordenadas se supone que parten de cero en adelante
 ad_ma_add_XY(NumX, NumY, AdjMat) ->
     XSet = array:get(NumX, AdjMat),
     %%no se repiten nodos en los destinos del nodo
```

```
48      AdjMatUpdated = array:set(NumX, ordsets:add_element(NumY, XSet), AdjMat),
49      YSet = array:get(NumY, AdjMat),
50      %devuelve la matriz de adjacencia actualizada
51      array:set(NumY, ordsets:add_element(NumX, YSet), AdjMatUpdated).
52
53
54  %usando una lista de números, actualiza la lista de adjacencia asumiendo
55  %que el primer número es la coordenada X y los siguientes son las coordenadas
56  %Y a las que se puede llegar desde X
57  ad_ma_add_List_XY([], AdjMat) -> AdjMat;
58
59  ad_ma_add_List_XY([_|[]], AdjMat) -> AdjMat;
60
61  ad_ma_add_List_XY([X|[Y|Rest]], AdjMat) ->
62      N_AdjMat = ad_ma_add_XY(X, Y, AdjMat),
63      ad_ma_add_List_XY([X|Rest], N_AdjMat).
64
65  %%%%%%%%%%%%%%%%%%%%%%%%%%
66  %%Imprime una matrix con los números de fila
67   ad_ma_printTmp(Matrix, 1) ->
68    io:format("1 ~w ~n", [array:get(array:size(Matrix)-1, Matrix)]);
69
70   ad_ma_printTmp(Matrix, Row) ->
71    io:format("~w ~w ~n", [size(Matrix) - Row, array:get(array:size(Matrix) - Row,
     Matrix)]),
72    ad_ma_printTmp(Matrix, Row-1).
73
74
75   ad_ma_print(Matrix, SizeOfTheMatrix) -> ad_ma_printTmp(Matrix, SizeOfTheMatrix).
76
77
78   printArrTmp(Array, 1) ->
79    io:format("~w ", [array:get(array:size(Array)-1, Array)]);
80
81   printArrTmp(Array, Row) ->
82    io:format("~w ", [array:get(array:size(Array)-Row, Array)]),
83    printArrTmp(Array, Row-1).
84
85   printArr(Array, SizeOfTheArray) ->
86     io:format("Size ~w ~n", [SizeOfTheArray]),
87     printArrTmp(Array, SizeOfTheArray).
88
89
90  %%%%%%%%%%%%%%%%%%%%%%%%%%
91  %Actualiza la lista de distancias
92  %Distances=Lista de distancias del todos los nodos
93  %Depth=número que indica profundidad
94
95   updateDist([], _Depth, Distances) ->
96    Distances;
97
98   updateDist([Head|Rest], Depth, Distances) ->
99    UpdatedDistances = array:set(Head, Depth, Distances),
100   updateDist(Rest, Depth, UpdatedDistances).
101
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%
%Parent=número que biene a ser el padre
%Parents=arreglo numérico que indica distancias.

 updateParentsList([], _L_L_Children, ParentsArr) -> ParentsArr;

 updateParentsList(_L_parents, [], ParentsArr) ->  ParentsArr;

 updateParentsList([H_P|R_P], [H_C|R_C], ParentsArr) ->
         NewParentsArr = updateParents(H_P, H_C, ParentsArr),
         updateParentsList(R_P, R_C, NewParentsArr).

 updateParents(_Parent, [], ParentsArr) ->
  ParentsArr;

 updateParents(Parent, [Head|Rest], ParentsArr) ->
   UpdatedParentsArr = array:set(Head, Parent, ParentsArr),
   updateParents(Parent, Rest, UpdatedParentsArr).


 update_parent_array_using_array_index_list([], _SourceArray, TargetArray) ->
 TargetArray;

 update_parent_array_using_array_index_list([Index|Rest], SourceArray, TargetArray)->
    X_source = array:get(Index, SourceArray),
    X_target = array:get(Index, TargetArray),
    if
      (X_target == -1) or (X_source < X_target) -> %-1=default parent
       NewTargetArray = array:set(Index, X_source, TargetArray),
       update_parent_array_using_array_index_list(Rest, SourceArray,
       NewTargetArray);
      true -> update_parent_array_using_array_index_list(Rest, SourceArray,
      TargetArray)
    end.

%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%Se encarga de actualizar las distancias de todos los nodos con respecto a
%la raiz escogida, es algoritmo bfs secuencial
%Visited=lista con números que indica los nodos visitados ej: [1,4,3]
%Matrix = matriz de adyacencia
%Distances=Lista de distancias del todos los nodos, la posición en la lista indica el
# de nodo
%Targets=lista de los nodos a visitar (adyacentes), es la cola del BFS
%Depth=distancia actual
 seq_stage(_Matrix, _Visited, Distances, Parents, [], _Iteration) ->
  {Distances, Parents};

 seq_stage(Matrix, Visited, Distances, Parents, Targets, Depth) ->
   Next = lists:map(fun(I) ->
             AM = array:get(I, Matrix),
             AM
             end, Targets),
   Size = array:size(Matrix),
   ParentsTemp = array:new([{size,Size},{fixed,true},{default,-1}]),
   ParentsTemp2 = updateParentsList(Targets, Next, ParentsTemp),
```

```
153
154     NewVisited = ordsets:union(Visited, Targets),
155     Frontier = ordsets:from_list(lists:flatten(Next)), %los adyacentes de los targets
156     NotSeen = ordsets:subtract(Frontier, NewVisited),
157
158     NewParents=update_parent_array_using_array_index_list(NotSeen, ParentsTemp2,
        Parents),
159     Distances2 = updateDist(NotSeen, Depth, Distances), %se actualiza las distancias
        de los nuevos targets
160     seq_stage(Matrix, NewVisited, Distances2, NewParents, NotSeen, Depth+1).
161
162 %%%%%%%%%%%%%%%%%%%%%%%%%%%
163 %%%%%%%Se encarga de devolver la lista de distancias de todos los nodos con respecto
    a la raiz escogida
164 %Start = valor del nodo de salida
165  sequential_bfs(Matrix, Start) ->
166     Visited = ordsets:from_list([]),
167     Size = array:size(Matrix),
168     DistancesResult = array:new([{size,Size},{fixed,true},{default,-1}]),
169     DistancesResult2 = array:set(Start, 0, DistancesResult), %al nodo raiz le asigna
        una distancia de cero
170     ParentsResult = array:new([{size,Size},{fixed,true},{default,-1}]),
171     ParentsResult2 = array:set(Start, 0, ParentsResult), %al nodo raiz le asigna un
        padre de cero
172     seq_stage(Matrix, Visited, DistancesResult2, ParentsResult2, [Start], 1).
173
174 %%%%%%%%%%%%%%%%%%%%%%%%%%%
175 %%%Proceso que se encarga de sacar los nodos adjacentes de un nodo de la lista de
    adyacencia
176 %%Se manetiene en memoria esperando trabajo, manda el resultado cuando lo obtiene y
    continúa en memoria
177 %%hasta que se le indique terminar
178  bfs_worker(AdjList, Master) ->
179     receive
180       {explore, Node} ->
181         Neighbors = array:get(Node, AdjList),
182         Master ! {found, Node, Neighbors},
183         bfs_worker(AdjList, Master);
184       terminate ->
185        ok
186     end.
187
188 %%%%%%%%%%%%%%%%%%%%%%%%%%%
189 %% Una vez todos los trabajos encargados a los procesos se hayan terminado entonces
    deveulve
190 %el conjunto de todos los nodos adyacentes sin repetir encontrados para los nodos
    investigados
191 %espera un # de respuestas con un conjunto cada una y al final devuelve un conjunto
    de las respuestas sin
192 %elementos repetidos producto de la unión de todos los conjuntos dados por las
    respuestas.
193  await_responses(0, ParentsArray, Queue) -> {ParentsArray, Queue};
194
195  await_responses(Requests, ParentsArray, Queue) ->
196      receive
```

```
197       {found, Node, Neighbors} ->
198         NewParentsArray = updateParents(Node, Neighbors, ParentsArray),
199         await_responses(Requests-1, NewParentsArray, ordsets:union(Neighbors, Queue))
200      end.
201
202 %%%%%%%%%%%%%%%%%%%%%%%%%%
203 %Este despachador de trabajos se encarga de asignar a todos los procesos
204 %disponibles el trabajo necesario para resolver la búsqueda de nodos adyacentes
205 %para toda la lista de nodos recibida en Queue.
206
207 %Queue=Lista de nodos para los cuales hay que explorar los adjacentes de cada uno
208 %Counter=contador retornado con el número de procesos invocados
209 %RemWorkers=Número de procesos que esperan trabajo
210 %Workers es el número total del procesos para trabajo paralelo
211 %cuando no hay procesos que esperan
212 % trabajo se les vuelve a reciclar asignandoles mas trabajo,
213
214  orderWork(_WorkersList, _RemWorkers, [], Counter) -> Counter;
215
216  orderWork(WorkersList, [], Queue, Counter) ->
217       orderWork(WorkersList, WorkersList, Queue, Counter);
218
219  orderWork(WorkersList, [NextWorker|RemWorkers], [Head|Tail], Counter) ->
220    NextWorker ! {explore, Head},
221    orderWork(WorkersList, RemWorkers, Tail, Counter+1).
222
223 %%%%%%%%%%%%%%%%%%%%%%%%%%
224 %Función con algoritmo de bfs paralelo que usa los workers de manera paralela
225 %para obtener todas las oleadas
226
227  synchronizer(_WorkersList, [], _Visited, Distances, ParentArray, _Depth) ->
     {Distances, ParentArray};
228
229  synchronizer(WorkersList, Queue, Visited, Distances, ParentArray, Depth) ->
230    Requests = orderWork(WorkersList, WorkersList, Queue, 0),
231
232    Size = array:size(ParentArray),
233    ParentsResultTemp = array:new([{size,Size},{fixed,true},{default,-1}]),
234    {SourceParentArrayTemp, NextQueue} = await_responses(Requests, ParentsResultTemp,
       []),
235
236    NewVisited = ordsets:union(Visited, Queue),
237    Frontier = ordsets:from_list(lists:flatten(NextQueue)),
238    NotSeen = ordsets:subtract(Frontier, NewVisited),
239
240    NewParents=update_parent_array_using_array_index_list(NotSeen,
       SourceParentArrayTemp, ParentArray),
241    NewDistances = updateDist(NotSeen, Depth, Distances), %Se actualizan las
       distancias
242    synchronizer(WorkersList, NotSeen, NewVisited, NewDistances, NewParents, Depth+1).
243
244 %%%%%%%%%%%%%%%%%%%%%%%%%%
245 %Se encarga de invocar al sincroniador que obtiene todas las oleadas (resolver el
    bfs)
246 %montando primero los worker en memoria
```

```
247 %y diciéndole al sincroniador que los use para resolver el problema total
248 %Retorna la lista de distancias de los nodos con respecto al nodo origen
249
250 %NumWorkers=Número de procesos paralelos que se quiere
251 %Workers=arreglos de Pid de los procesos en memoria
252 %AsjList = matriz de adyacencia
253
254 processCoord(0, WorkersList, AdjList) ->
255    receive
256      {search, Origin, ResponseAddr} ->
257        Size = array:size(AdjList),
258        DistancesTmp = array:new([{size,Size},{fixed,true},{default,-1}]),
259        DistancesTmp2 = array:set(Origin, 0, DistancesTmp),
260        ParentsTmp = array:new([{size,Size},{fixed,true},{default,-1}]),
261        ParentsTmp2 = array:set(Origin, 0, ParentsTmp), %al nodo raiz se le asigna un
           padre de cero
262
263        {Distances, ParentArray} = synchronizer(WorkersList,
264          ordsets:from_list([Origin]),
265          ordsets:from_list([]),
266          DistancesTmp2,
267          ParentsTmp2,
268          1),
269        ResponseAddr ! {ok, Distances, ParentArray};
270      die ->
271       lists:map(fun(W) -> W ! terminate end, WorkersList)
272    end;
273
274  processCoord(NumWorkers, Workers, AdjList) ->
275    W = spawn(?MODULE, bfs_worker, [AdjList, self()]),
276    processCoord(NumWorkers-1, [W] ++ Workers, AdjList).
277
278 %%%%%%%%%%%%%%%%%%%%%%%%%
279 start_seq(FileName, Num_Nodes) ->
280    AM = create_ad_ma(FileName, Num_Nodes),
281    T1 = now(),
282    {ResultsDistancesArr, ResultsParentArray} = sequential_bfs(AM, 0), %prueba del
       secuencial.
283    T2 = now(),
284    [Time|_] = io_lib:format("~.6f", [timer:now_diff(T2, T1)/1000000]),
285    {ResultsDistancesArr, ResultsParentArray, Time}.
286
287  start_parallel(FileName, Num_Nodes, Num_Procs) ->
288    AM = create_ad_ma(FileName, Num_Nodes),
289          T1 = now(),
290    Syncer = spawn(?MODULE, processCoord, [Num_Procs, [], AM]), %usando Num_Procesos
       procesos
291    Syncer ! {search, 0, self()}, %obtener la solución del bfs con el nodo 0 como
       origen
292    {ResultsDistancesArr, ResultsParentArray} = receive
293       {ok, DistancesArr, ParentArray} -> {DistancesArr, ParentArray}
294        end,
295    Syncer ! die,
296    T2 = now(),
297      [Time|_] = io_lib:format("~.6f", [timer:now_diff(T2, T1)/1000000]),
```

```
298    {ResultsDistancesArr, ResultsParentArray, Time}.
299
300
301 %%%%%%%%%%%%%%%%%%%%%%%%%%
302 main_internal(Program, FileName, Num_Nodes, Num_Procs) ->
303      case Program of
304       1 -> {_ResultsDistancesArr, _ResultsParentArray, Time} = start_seq(FileName,
         Num_Nodes),
305          Used_mem = misc_functions:get_my_peak_used_mem(),
306          io:format("Sequential BFS in erlang took: ~s sec. and memory = ~w KB~n",
             [Time, Used_mem]);
307       2 -> {_ResultsDistancesArr, _ResultsParentArray, Time} =
         start_parallel(FileName, Num_Nodes, Num_Procs),
308          Used_mem = misc_functions:get_my_peak_used_mem(),
309          io:format("Parallel BFS in erlang took: ~s sec. and memory = ~w KB~n",
             [Time, Used_mem]);
310       _Else -> io:format("Uso: erl -noshell -run bfs main <tipo algoritmo:1=BFS
         secuencial,2=BFS paralelo>
311              <Nombre Archivo> <Num Nodos> <Num Procesos> -s init stop\n", [])
312      end.
313
314 main(ParamList) -> %[Program, FileName, Num_Nodes, Num_Procs]
315   Program = list_to_integer(lists:nth(1, ParamList)),
316   FileName = lists:nth(2, ParamList),
317   NumNodos = list_to_integer(lists:nth(3, ParamList)),
318   NumProcesos = list_to_integer(lists:nth(3, ParamList)),
319   main_internal(Program, FileName, NumNodos,NumProcesos).
320
```

7.3 Correos de validación de aporte por comunidades internacionales

7.3.1 The Society of Digital Information and Wireless Communications (SDIWC)

On Wed, Sep 16, 2015 at 10:10 PM, The International Journal of New Computer Architectures and Their Applications (IJNCAA) <ijncaa@sdiwc.net> wrote:

Dear Hugo:

THANK YOU for your email.

We think that it is important.

Good Luck.

Sincerely yours

Jackie Blanco

www.sdiwc.net

----- Original Message -----

From: Hugo Mora <mauriciomoraarley@gmail.com>

To: ijncaa@sdiwc.net

Date: Wed, 16 Sep 2015 20:14:48 -0600

Subject: I need your opinion

Hello, my name is Hugo Mora Arley, I am doing my master's degree in my country (Costa Rica), I read a study named "Performance Evaluation of

Message Passing vs. Multithreading Parallel Programming Paradigms on Multi-core Systems" (reference: *(IJNCAA) 4(2): 108-116*) and I think you are very familiar with this kind of studies comparing different programming paradigms, I am doing a research about a programming model named "Actors Model" in the Smartphones, the idea of this research is to give some data to the decision maker (programmer, company owner, etc) to evaluate if this model is adequate or not in terms of performance to develop his project in the Smartphone.

The Actors Model is becoming favorable because the tendency of the Smartphones to be a multi-core platform, so, the concurrent programming arises as the logical path to create apps and libraries. The benefits of the Actors Model to avoid typical problems of the concurrent programming could be very attractive for the decision maker, the problem is that there are no studies to measure if this model will work as expected in the Smartphones in terms of performance. My study will compare the performance obtained using the concurrent programming with the traditional threads model in C language and the Actors Model present in the Erlang language, the comparison will be done as described before in the Smartphone as the target platform.

This is a research I am doing in my university as my thesis, so this impartial research will be useful for every body and I consider that this

is an important study.

Please, I need to know if you consider that this study is an important contribution for decision makers from your point of view.

Thank you for your valuable help.

7.3.2 Investigadores en paradigmas de programación

On Thu, Sep 17, 2015 at 1:05 AM, Denti Mattia <mattia.denti@aalto.fi> wrote:

Hello Hugo,

I'm not very familiar with the Actors Model and the current behavior and performance of concurrent applications on mobile devices, and I think the main reason for the lack of studies is the amount of different devices in the market. There are too many chips, architectures, and hardware combinations to give a complete answer to any question in this domain.

However I can say that in my opinion studies on specific characteristics of applications on mobile devices are much needed.

Your research may not be able to cover everything (especially in the scope of a thesis), but for decision makers (and anyone else interested) any additional information is definitely valuable.

So, yes, I do think that this study can be an important contribution for decision makers.

BR,

Mattia

From: Hugo Mora [mauriciomoraarley@gmail.com]

Sent: Thursday, September 17, 2015 4:32 AM

To: Denti Mattia; Nurminen Jukka

Subject: I need your opinion

Hello, my name is Hugo Mora Arley, I am doing my master's degree in my country (Costa Rica), I read your study named "Performance and energy-efficiency of Scala on mobile devices" and I think you are very familiar with this kind of studies in mobile devices, I am doing a research about a programming model named "Actors Model" in the Smartphones, the idea of this research is to give some data to the decision maker (programmer, company owner, etc) to evaluate if this model is adequate or not in terms of performance to develop his project in the Smartphone.

The Actors Model is becoming favorable because the tendency of the Smartphones to be a multi-core platform, so, the concurrent programming arises as the logical path to create apps and libraries. The benefits of the Actors Model to avoid typical problems of the concurrent programming could be very attractive for the decision maker, the problem is that there are no studies to measure if this model will work as expected in the Smartphones in terms of performance. My study will compare the performance obtained using the concurrent programming with the traditional threads model in C language and the Actors Model present in the Erlang language, the comparison will be done as described before in the Smartphone as the target platform.

This is a research I am doing in my university as my thesis, so this impartial research will be useful for every body and I consider that this is an important study.

Please, I need to know if you consider that this study is an important contribution for decision makers from your point of view.

Thank you for your valuable help.

7.4 Pasos para migrar Erlang en el smartphone básico

A manera de guía se indica lo que se tuvo que hacer para migrar Erlang al smartphone básico en esta investigación, ya que fue el que más costo de instalar. Primero fue necesario instalar Linux en el smartphone, se instala una versión de Linux denominada Linwizard, esta versión está especialmente configurada para este dispositivo [39] pero se encontraba solo con lo básico. Posteriormente fue necesario instalar varios paquetes para que el entorno estuviera listo con el compilador gcc y las librerías necesarias para compilar el Erlang versión R12B-5, el cual se encuentra disponible en el sitio web de Erlang [30]. Se instala el compilador Gcc versión 4.3.2 junto con build-essential y el m4; para poder instalar esto se tuvo que instalar antes los siguientes paquetes según se demandara conforme se avanzara en la instalación o se tratara de compilar el Erlang:

- base
- libgcc
- libstd
- cpp
- libgomp1
- libmud
- libobj
- libgfortran
- gcc
- libstdc++6-4.1-dev (este se instala con las opciones -i --ignore-depends g++-4.3)
- g++

- gobjc
- gfortran

Uno de los problemas más graves es que el dispositivo no contaba con un repositorio de paquetes completo para que el instalador de paquetes instalar lo necesario con los paquetes dependientes así que la instalación fue uno por uno conforme se indicaban las dependencias. Los paquetes que sirvieron son los compatibles para la plataforma arm en el sitio oficial de Debian [008].

Una vez instalados los paquetes se tuvo que crear un archivo de memoria virtual para compilar el Erlang ejecutando los siguientes pasos:

1. dd if=/dev/zero of=/mnt/swapfile bs=1M count =200
2. chown root:root /mnt/swapfile
3. mkswap /mnt/swapfile
4. swapon /mnt/swapfile

Posteriormente descomprimió el archivo de Erlang en un directorio (instrucción = tar -xzf otp_src_R11B-5.tar.gz) y se hizo los siguientes pasos ya estando en ese directorio:

1. ./configure --without-ssl --without-java --disable-hipe
2. make

Para ejecutar estos 2 pasos el smartphone duró unos 36 minutos.

Posteriormente se deshabilita el archivo de memoria virtual con la siguiente instrucción:

1. swapoff /mnt/swapfile

Luego, una vez conectado con un cable usb el smartphone al PC, se usan las siguientes instrucciones en la SMARTPHONE y en la PC:

1. ifconfig usb0 up 192.168.2.200 #smartphone
2. ifconfig usb0 up 192.168.2.201 #pc

Con esto ya queda listo el ambiente para ejecutar las pruebas en el smartphone básico.