COMPUTACIÓN PARALELA EN PYTHON SOBRE UN CLUSTER DE ALTO RENDIMIENTO

CARLOS ANTONIO GALÁN GUERRA
Estudiante de Ing. Sistemas

JOSÉ ORLANDO MALDONADO BAUTISTA
Director

Departamento de Ingenierías Eléctrica, Electrónica, Sistemas y Telecomunicaciones
Facultad de Ingenierías y Arquitectura
Ingeniería de sistemas
Universidad de Pamplona
Pamplona, Dic. 2016

Tabla de Contenido

Lista de Figuras.

Lista de Tablas

Lista de Anexos.

1. Resumen

Los clúster han aparecido como una herramienta computacional de alto rendimiento para la solución de problemas complejos en ciencias e ingeniería. En ese sentido, en el siguiente trabajo de grado presentamos resultados de la implementación de un clúster de alto rendimiento usando las potencialidades de Python en la computación de alto rendimiento. El procedimiento consiste en dividir el procesamiento en distintos equipos de cómputo conectados a través de una red, donde el rendimiento del proceso es proporcional a la cantidad de equipos que podamos agregar a la red. Se muestran los resultados de la evaluación del desempeño de la plataforma computacional, a partir de medidas de rendimiento como: eficiencia, escalabilidad y ganancia del sistema. De estos resultados de la configuración pudimos caracterizar y optimizar los tiempos de duración del procesamiento de datos y/o información.

Estos resultados fueron obtenidos aplicados a un problema en particular de segmentación de imágenes, sin embargo puede ser aplicado a otros tipos de problemas de las ingenierías y las ciencias naturales que requieran gran capacidad de procesamiento, tiempos accesibles y bajo costo.

2. Introducción

2.1. Planteamiento y Justificación del Problema

En la actualidad es conocido que los equipos de cómputo han aumentado su capacidad de procesamiento considerablemente, sin embargo también es cierto que la solución de cierto tipo de problemas conlleva un alto costo computacional, lo que implica largos tiempos de procesamiento si se tratan de resolver en una máquina convencional, junto a la programación secuencial tradicional. Por ejemplo, el análisis de imágenes de alta resolución, espectral y/o espacial puede requerir altos tiempos de procesamiento. Una alternativa a las limitaciones antes expuestas, son los clúster de alto rendimiento, que reúnen conjuntos de máquinas, o bien con configuraciones domésticas, o bien servidores de mayor capacidad, que permiten mejorar los tiempos de procesamiento. En la mayoría de los casos, el desarrollo de algoritmos de computación paralela y distribuida, requiere invertir gran cantidad de tiempo en el estudio de lenguajes especializados o extensiones y API's sobre lenguajes de uso general, tiempo que puede ser invertido en el análisis específico del problema. En este punto, surge la idea de utilizar herramientas que impliquen una menor curva de aprendizaje y el menor esfuerzo del programador en la paralelizacion de su código. Python y sus herramientas para la computación paralela parece una alternativa viable, debido a las características del lenguaje como su versatilidad (múltiples aplicaciones), legibilidad del código y facilidad de aprendizaje. Por tanto se quiere evaluar el desempeño que se obtenga al implementar alguna de

las herramientas de programación paralela ofrecidas por Python, sobre un clúster de alto rendimiento.

2.2. Objetivos

2.2.1. Objetivo General

Diseño de un clúster de alto rendimiento para la resolución de problemas de análisis de imágenes usando computación paralela con Python

2.2.2. Objetivos Específicos

- Estudio de potencialidades que ofrece Python en la computación de alto rendimiento.
- Desarrollo de un clúster de alto rendimiento usando el modulo “Parallel Python”.
- Evaluación del desempeño de la plataforma computacional, a partir de medidas de rendimiento como: eficiencia, escalabilidad y ganancia del sistema.

2.3. Enfoque Metodológico

Por la naturaleza de la presente investigación se adopta un enfoque cuantitativo, por lo cual se han abordado las siguientes etapas: **[1]**

Planteamiento del problema, que se ha establecido mediante la descripción del problema, la justificación de la investigación y la formulación de objetivos de la misma.

Desarrollo de la perspectiva teórica, realizada mediante la revisión de la literatura para establecer un contexto, un marco teórico y un estado del arte.

La descripción del alcance de la investigación. Esta llegará a ser de tipo descriptivo, pues llega con la aplicación de medidas de rendimiento de los algoritmos sobre el clúster implementado y análisis de los resultados. Aunque se llega mediante el análisis a tratar de explicar los resultados obtenidos, para los objetivos de este proyecto no se alcanza a realizar una explicación estructurada en rigor.

Formulación de la hipótesis de investigación: A saber: “Es posible obtener un mejor rendimiento, en términos de: Aceleración, Eficiencia y Escalabilidad al implementar los algoritmos de computación paralela en un clúster del alto rendimiento con la librería Parallel Python, frente a las versiones secuenciales de los algoritmos en un único nodo”

Diseño de investigación: De tipo experimental, las variables a medir serán precisamente Celeridad, Eficiencia y Escalabilidad, aplicada a cada uno de los algoritmos implementados.

La Selección de la muestra, recolección y análisis de datos, y presentación de resultados: Se explican en el apartado 6. Evaluación del rendimiento.

3. Marco Teórico

3.1. Computación de Alto Rendimiento

La computación de alto rendimiento surge de la creciente demanda de mayores unidades de procesamiento a la hora de plantear soluciones a problemas complejos de carácter computacional que consumen recursos como tiempo, entre otros. Para lograr cumplir esta finalidad la computación de alto rendimiento se apoyó en distintas tecnologías computacionales como lo son las supercomputadoras, clúster o también la computación paralela. La mayoría de las ideas actuales de la computación distribuida se han basado en la computación de alto rendimiento. Los avances en ese campo se presentaron en mayor medida a nivel de hardware, lo que motivó la necesidad de idear nuevas herramientas y metodologías que permitieran su implementación y uso de una manera eficiente [2].

3.1.1. Computación Paralela

Desde el año 2002 la mejora del rendimiento de un solo procesador ha disminuido alrededor del 20% anual, cifra que contrasta con el 50% anual que se venía reflejando desde 1986 [3].

De otro lado, la computación paralela se ha convertido en el componente clave de la computación de alto rendimiento (*HPC*) desde la década de los noventa y a mediano plazo parece que este argumento no sufrirá variaciones. Cuando se requiere explotar esta tecnología se necesita tener una buena compresión de las arquitecturas de estas y además también se

hace necesario saber cómo escribir programas paralelos que aprovechen estas tecnologías. La programación y en general la computación paralela debería ser una parte básica de la formación en un futuro, esta debería abarcar transversalmente cada área del conocimiento y romper con el mito de que la programación solo debe enseñarse en el área de las tecnologías y afines [4].

Sin embargo la computación paralela es difícil de definir con precisión, ya que este concepto abarca varios niveles. Existe el nivel donde varias instrucciones pueden estar siendo procesadas simultáneamente en la *CPU*, a este se le denomina paralelismo de nivel de instrucción y está fuera del control explícito del cliente. En el otro extremo está el tipo de paralelismo donde más de una secuencia de instrucciones es manejada por múltiples procesadores, este tipo de paralelismo es programado típicamente por el usuario [5].

3.1.2. Programación en Paralelo

El concepto de la programación en paralelo es derivado del surgimiento de las arquitecturas y hardware para la computación en paralelo. A medida que estas fueron evolucionando con el paso del tiempo, fue surgiendo la idea de poder aprovechar todas las cualidades del hardware a través del software, mediante la programación paralela. La finalidad y lo que promete esencialmente la programación paralela es proporcionar beneficios en términos de precios, rendimiento, velocidad y escalabilidad. La optimización de estos factores brinda la posibilidad o capacidad de poder manejar

problemas más grandes o más intensivos en términos computacionales que antes [4].

Lo que hace realmente atractiva la programación en paralelo es la esperanza de que los tiempos para resolver problemas computacionales disminuirán al compás del paso del tiempo, esto como consecuencia de los constantes avances que se producen en hardware, más y mejores procesadores, aumento de memoria y otros más que aumentan masivamente la potencia computacional disponible a un precio moderado. Además, la programación paralela introduce fuentes adicionales de complejidad: si tuviéramos que programar al más bajo nivel, no solo aumentaría el número de instrucciones ejecutadas, si no que también tendríamos que gestionar explícitamente la ejecución de miles de procesadores y coordinar millones de interacciones entre procesadores. Por lo tanto la abstracción y la modularidad en la programación son al menos tan importantes como en la programación secuencial [6].

3.1.3. Arquitecturas

Al hablar de computación en paralelo se hace referencia al conjunto de técnicas usadas para obtener que un problema o tarea sea procesado paralelamente, esto es producto de hacer uso ya sea de hardware netamente paralelo o fue logrado mediante técnicas de programación [3].

El primer paso hacia la paralización de las arquitecturas de los computadores, se da con la aparición de los procesadores. Los procesadores utilizan unidades aritméticas lógicas segmentadas,

extendiendo el concepto de paralelismo al tratamiento de grandes volúmenes de datos. La contribución de la arquitectura al aumento de las prestaciones de los sistemas de cómputo ha venido de la mano del paralelismo y ha logrado que estos conceptos se extiendan y sean adoptados tanto en uso civil como también en el campo militar.

3.1.3.1. Hardware Paralelo

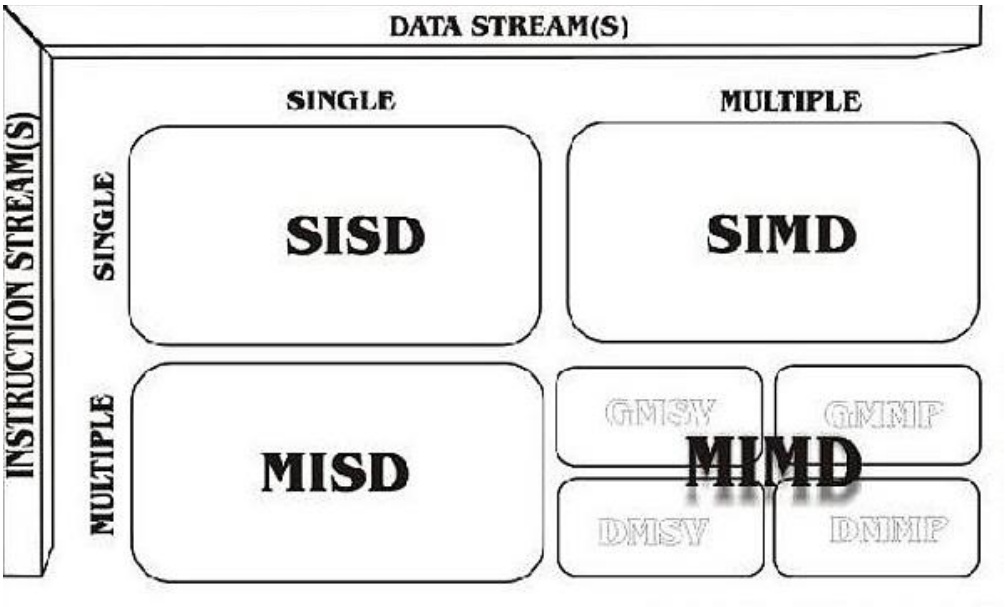

Figura (1) *Esquema de hardware paralelo (Flynn,* [7]*)*

3.1.3.1.1. Sistemas SIMD

En la computación paralela, La taxonomía de Flynn [7] es la más usada frecuentemente para clasificar las arquitecturas de las computadoras. Esta clasificación es realizada de acuerdo al número de flujo de instrucciones y el número de datos que puede procesar simultáneamente.

Los sistemas SIMD (del inglés: *Single Instruction, Multiple Data*), son sistemas paralelos que como su nombre lo indica operan muchos flujos de datos simultáneamente aplicando las mismas instrucciones a estos. Los sistemas *SIMD* poseen algunas características particulares como:

- Todas las unidades de procesamiento ejecutan la misma instrucción.
- Cada unidad procesa un flujo de datos totalmente distinto al que se ejecuta en las otras unidades.
- Todas las unidades operan simultáneamente.

3.1.3.1.2. Sistemas MIMD

Múltiples instrucciones, múltiples datos o MIMD (del inglés: *Multiple Instruction, Multiple data*). Este sistema posee la capacidad de soportar múltiples secuencias de instrucciones que operan además múltiples flujos de datos, todo esto de forma simultánea [3]. Por tanto, los sistemas SIMD consisten normalmente en una colección de unidades o núcleos de procesamiento que trabajan de forma totalmente independiente, cada uno de los cuales tiene su propia unidad de control.

Lo que diferencia a estos sistemas de los sistemas SIMD, es que los MIMD suelen ser asíncronos, lo que quiere decir que los procesadores de cada unidad pueden funcionar a su propio ritmo.

Características de los sistemas MIMD:

- Cada unidad de procesamiento ejecuta una instrucción distinta a las demás.
- Cada unidad de procesamiento procesa un flujo de dato distinto al de los demás.
- Todas las unidades de procesamiento operan de manera simultánea.

3.1.3.1.3. Redes de Interconexión

La interconexión juega un papel de suma importancia y decisivo en el rendimiento de los sistemas de memoria distribuida y compartida, esto es un hecho incluso si contamos con procesadores que tengan capacidad de procesamiento ilimitado. Esto debido a que una interrelación lenta degrada significativamente el rendimiento general de los programas que se están ejecutando de forma paralela [3].

3.1.3.2. Software Paralelo

3.1.3.2.1. Coordinación de Procesos/Hilos

Obtener un excelente rendimiento en sistemas paralelos es una tarea que en la mayoría de casos no es para nada trivial. Esto debido a que convertir un programa que trabaja de manera secuencial a la versión paralelizada del mismo requiere un profundo análisis y observación.

Cuando se requiere hacer uso de todas las unidades de procesamiento de un sistema, es necesario mediante la programación

realizar una adecuada coordinación entre las tareas a ejecutar simultáneamente. Además de tener en cuenta la relación existente entre las tareas a ejecutar o los datos resultantes del procesar esas tareas, de no planificar bien la coordinación entre los hilos de ejecución que lo requieran, puede derivar en la inexactitud de los resultados esperados.

3.1.3.2.2. Memoria Compartida

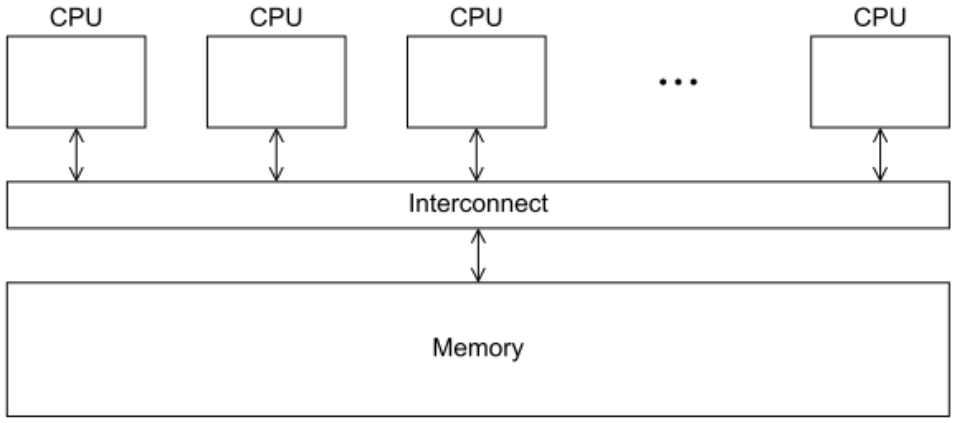

Figura (2) *Esquema de memoria compartida (Pacheco,* [3]*)*

La memoria compartida o shared memory recibe su nombre porque es una memoria común a todos los hilos o bloques de tareas que se estén procesando, siendo para todos estos el medio de comunicación entre ellos. Como principal ventaja de los sistemas que poseen memoria compartida destaca la posibilidad de acceso rápido a cualquier variable declarada en ella, puede ser accedida por cualquier bloque de código que se esté ejecutando simultáneamente, aunque es de aclarar que el alcance de cada variable compartida declarada

está limitado a los bloques de tareas de la misma instancia de la variable [2].

3.1.3.2.3. Memoria Distribuida.

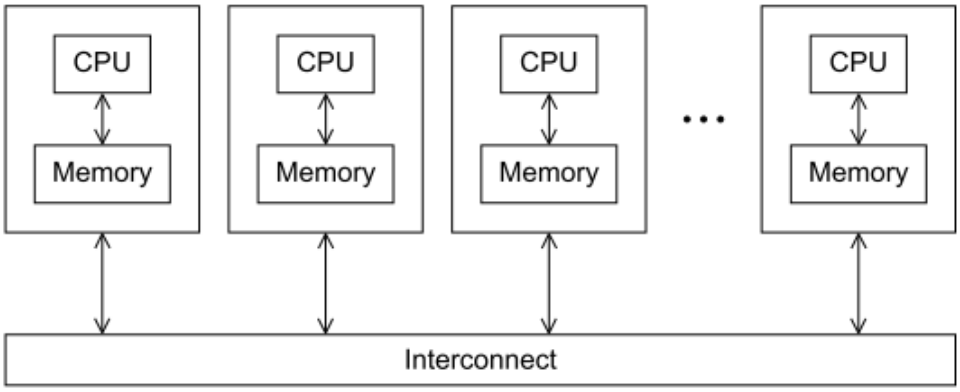

Figura (3) *Esquema de memoria distribuida (Pacheco,*[3] *)*

Los sistemas de memoria compartida aunque suelen ser eficientes no son escalables, una alternativa a estos sistemas son los computadores de memoria distribuida o simplemente sistemas de memoria distribuida. Este tipo de computadoras paralelas se forma conectando varias computadoras completas a través de una red de interconexión. Cada procesador paralelo es un procesador con memoria local, a esta memoria solo puede acceder el procesador local. En esta arquitectura la memoria paralela está distribuida entre los distintos procesadores. La red de interconexión permite a los procesadores comunicarse enviando o recibiendo mensajes a otros procesadores en la máquina, los mensajes enviados pueden incluir datos requeridos por otros procesadores. Esta comunicación debe ser explícitamente especificada [2].

3.1.3.2.4. Programación de Sistemas Híbridos

Existe la posibilidad de programar sistemas como clúster de procesadores multicore usando una combinación de una API para memoria compartida en cada uno de los nodos y una API de memoria distribuida para la comunicación entre nodos. Sin embargo esto generalmente solo se hace para programas que requieren los niveles de rendimiento más altos posibles, ya que la complejidad de esta arquitectura hibrida hace que el desarrollo del programa sea extremadamente difícil [3].

3.1.4. Clúster

Un clúster es un tipo de sistema informático en paralelo o distribuido, que consiste en una colección de sistemas autónomos interconectados que trabajan juntos en equipo como un solo recurso integrado [8]**.**

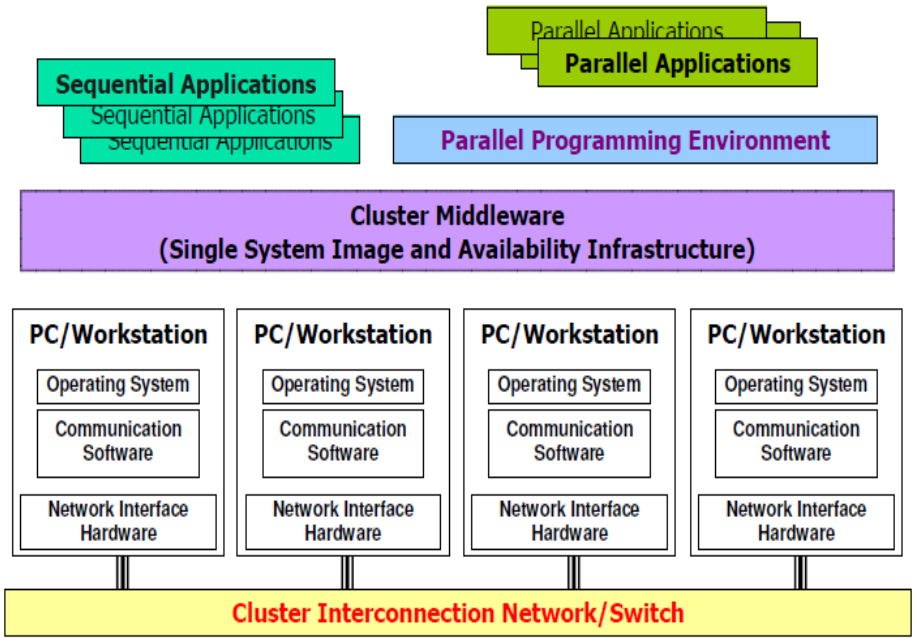

Figura (4). Esquema de la arquitectura de un Clúster (R. Buyya [9])**.**

3.1.4.1. Clúster de Alto Rendimiento

El objetivo de un clúster de alto rendimiento es compartir el recurso más valioso de un computador, es decir, la capacidad de procesamiento. Este tipo de clústeres son muy populares, y quizás también los más antiguos, ya que en sus inicios fueron desarrollados para centros de cómputo enfocados en investigaciones científicas donde era necesario tener un gran poder de procesamiento, capaz de soportar una inmensa cantidad de cálculos para resolver un problema específico[10]. El tipo de clúster de alto rendimiento que se implementará en la investigacion será el llamado clúster beowulf, el cual tiene como característica que usa computadores de uso personal y su interconexión

es a través de una red estándar, no hay uso de hardware desarrollado netamente para computación paralela

3.1.5. Paradigmas de Programación

La construcción de software paralelo se basa en la idea de tomar un problema grande y dividirlo en subproblemas, cada uno de los cuales se resuelve en forma concurrente al resto. Intuitivamente es de esperar un mejor desempeño de la versión paralelizada del problema respecto a la versión secuencial del mismo. Desear e implementar algoritmos que se ejecuten sobre maquinas paralelas no están fácil. No existe una única solución paralela. La naturaleza del problema, la relación entre los datos y la independencia o dependencia entre los resultados, son determinantes para obtener multiplicidad de algoritmos paralelos para resolver un mismo problema [2].

3.1.5.1. Modelos Paralelos Estándares

Un modelo que describe una estructura o patrón para resolver un problema de una u otra manera se le denomina paradigma. Un paradigma muy usado es el llamado “divide y vencerás”, este se caracteriza por dividir el problema en partes más pequeñas y abordar cada uno de ellos hasta darle solución, para luego obtener la solución al problema original [2]. Al momento de construir software paralelo hay que tener en cuenta:

- La naturaleza del problema
- La relación entre los datos
- La posibilidad de dividir los datos o el problema
- La independencia o dependencia de los resultados

3.1.5.1.1. Paralelismo de Datos

El paralelismo de datos se caracteriza por la ejecución paralela de la misma operación sobre distintos datos. En otras palabras, múltiples unidades funcionales aplican simultáneamente la misma operación a un subconjunto de elementos. Dicho subconjunto integra la partición del conjunto total de datos [2]**.**

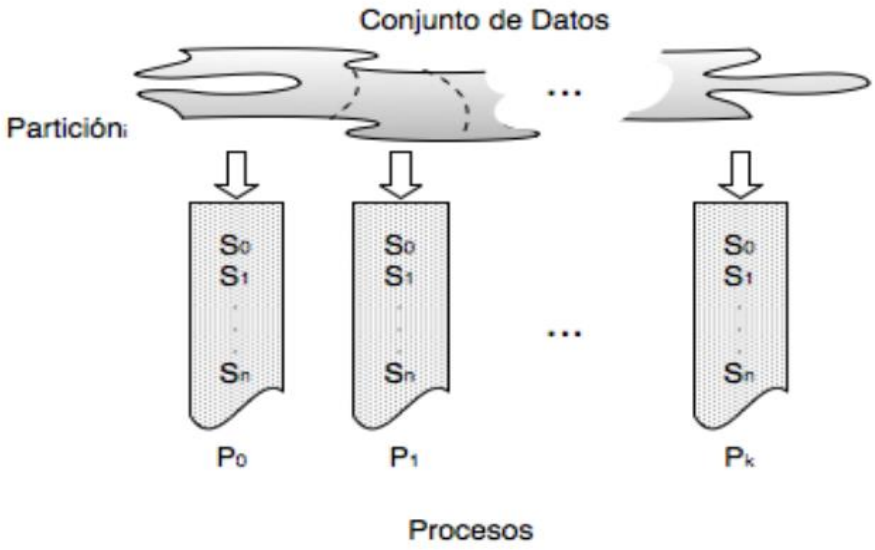

Figura (5). Paralelismo de datos (M. Piccoli [2])

Este modelo de programación paralela de datos está dirigido a arquitecturas con memoria distribuida físicamente, un clúster puede

ser quizá el ejemplo de arquitectura más idónea para la implementación de este paradigma.

3.1.5.1.2. Paralelismo de Tareas

A este paradigma también se le conoce como paralelismo de control, consiste en aplicar simultáneamente diferentes operaciones a distintos elementos de datos. El flujo de datos entre los distintos procesos que aplican el paralelismo de control puede llegar a ser demasiado complejo, dificultando su programación [2]**.**

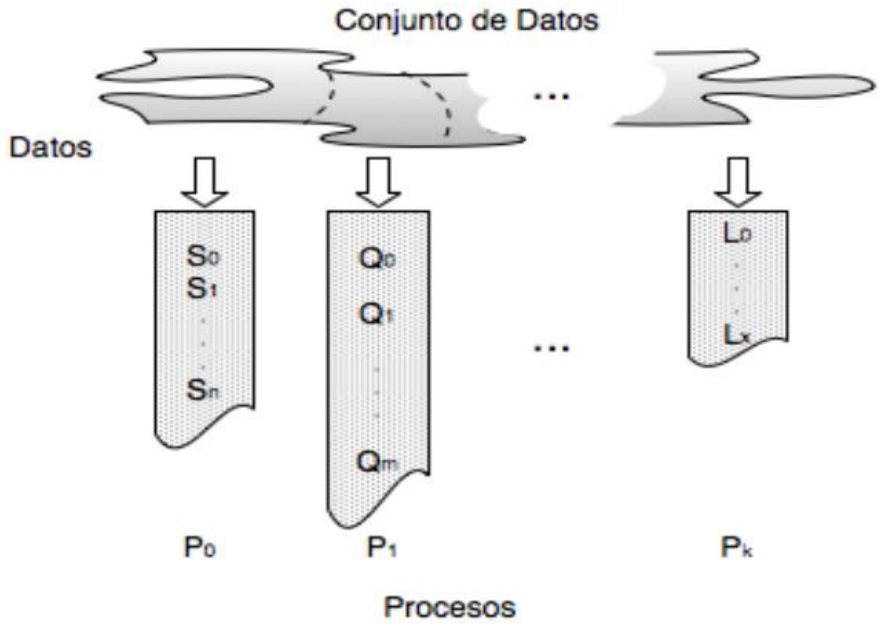

Figura (6). Paralelismo de control (M. Piccoli [2])

Una buena técnica para entender este paradigma es desarrollar un grafo en el cual se evidencien todas y cada una de las dependencias de las tareas involucradas en el sistema paralelo. En este se reflejaran las interrelaciones de las distintas tareas y sus dependencias [2].

3.1.5.1.3. Paralelismo Anidado de Datos

Cuando el estilo SIMD, propio del paralelismo de datos, deriva en un estilo SPMD, (del inglés: Simple Programa-Múltiples Datos), operaciones complejas sobre los dominios de los datos, tienen lugar. Si los dominios de datos son en sí mismos estructurados, y si las operaciones paralelas a aplicarles también lo son, se está en presencia de un nuevo modelo de paralelismo: paralelismo anidado de datos [2].

3.1.5.2. Paradigmas Orientados a la Arquitectura

Una computadora convencional consiste de un procesador, el cual ejecuta un programa almacenado en su memoria. El procesador puede acceder a cada dirección de memoria. Extender el modelo de único procesador, significa tener múltiples procesadores y múltiples módulos de memorias conectados de alguna manera. La topología de la conexión entre los procesadores y los módulos de memoria está determinada por cómo se considera a la memoria: única o propia. En el primer caso todos los módulos integran un único espacio de direcciones; para el segundo caso, la memoria se considera propia de cada procesador. A continuación se explican cada uno de los paradigmas que surgen [2].

3.1.5.2.1. Paradigma de Memoria Compartida

La programación de un multiprocesador implica tener el código ejecutable y los datos necesarios para su ejecución almacenados en la memoria compartida. De esta manera cada unidad de

procesamiento accede a la memoria, ya sea para ejecutar el código, obtener los datos para la ejecución o guardar los resultados de ella. Para un programador este paradigma es muy atractivo, porque la memoria compartida en conveniente para compartir los datos, una de las limitantes en este se encuentra en el hardware, los accesos a cada memoria compartida deben ser ágil y se deben instrumentar los mecanismos para proveer la consistencia de la misma [2].

3.1.5.2.2. Paradigma de Paso de Mensaje

Cuando se realiza programación multicomputadora o sobre un clúster, esto conlleva la división del problema en partes o tareas. Cada una de estas tareas puede ejecutarse en una computadora o procesador diferente del clúster. La comunicación entre las tareas se realiza través del pasaje de mensaje, por lo general las librerías son las encargadas de proporcionar las herramientas para al hecho [2].

Algunas características de este paradigma son:

- Un programa paralelo con pasaje de mensajes se puede obtener utilizando cualquier tipo de biblioteca que provea herramientas para la comunicación entre procesos a través del paso de mensajes.
- El pasaje de mensaje debe ser especificado en el código del programa que resuelve el problema.
- Los datos son propios de cada tarea, no se comparten entre ellas.

- No necesita tener mecanismos especiales para el control de acceso simultáneo a datos.

3.1.6. Herramientas de Programación

A causa de las progresivas mejoras tecnológicas desarrolladas en las últimas décadas, la capacidad de procesamiento de los computadores ha aumentado progresivamente. Sin embargo estas mejoras tecnológicas tienen un límite físico [11], por lo que se puede decir que la capacidad de procesamiento no es ilimitada.

Existen una serie de problemas que no pueden resolverse con las tecnologías que poseen los computadores en la actualidad, es por eso que desde hace más de dos décadas se vienen investigando y desarrollando el procesamiento en paralelo. Actualmente se usa esta tecnología y parece ser la tendencia para mejorar la potencia de procesamiento de los computadores

3.1.6.1. Lenguajes de Uso General

Por muchos años existió una carencia total de un lenguaje diseñado netamente para la programación en paralelo y además los que se fueron desarrollando no se ha masificado su uso. De este modo, para llenar ese vacío se tuvo que hacer uso de lenguajes de uso general entre los que destacaron C y Fortran.

3.1.6.1.1. C/C++

Por muchos años ha sido quizá el lenguaje de programación más difundido y usado, puede ser también el lenguaje de iniciación en el

mundo de la programación para la mayoría de programadores. Al ser un lenguaje compilado le asegura una mayor rapidez en la ejecución de programas, esta cualidad la conserva incluso al compararlo con otros lenguajes que poseen la característica de ser compilado.

En la computación paralela ha tenido un valor preponderante siempre, por la característica antes citada. Sin embargo C/C++ no es un lenguaje cuyo propósito sea el desarrollo de aplicaciones en paralelo, por ello debe valerse de extensiones creadas para cumplir de manera más eficiente este objetivo entre las que sobresalen algunas como OpenMP y MPI.

3.1.6.1.2. Fortran

Pese a ser un lenguaje de alto nivel de uso general, Fortran tiene una marcada especialización orientada al cálculo matemático, debido a esto es ampliamente adoptado en el área científica. Como en la informática no resulta fácil cambiar algo que ya funciona, esto genera cierta reticencia a migrar el código a otros lenguajes más especializados en la computación paralela, por tanto, solo se debe paralelizar aquello que de verdad merezca ser paralelizado.

3.1.6.1.3. JAVA

Es también un lenguaje de propósito general, multiparadigma, orientado a objetos. Es considerado uno de los lenguajes de programación más populares en uso, aun así no es un lenguaje que destaque en la programación paralela.

Java Parallel Processing Framework es un framework de java que provee una API para la computación en grids. El framework ofrece un conjunto de herramientas extensibles y modificables para facilitar la paralización de aplicaciones que requieran de esta y así mejorar el rendimiento [12].

3.1.6.2. Lenguajes de programación en Paralelo

Los lenguajes de programación paralela son aquellos creados especialmente para este ámbito, generalmente trabajan con memoria compartida. Estos lenguajes proveen estructuras de control, sentencias, entre otros que facilitan al programador trabajar con tareas independientes y variables de memoria compartida [13].

Aunque muchos lenguajes han evolucionado mediante el tiempo, ninguno ha sido ampliamente aceptado universalmente, en su mayoría se creaban extensiones para usar en lenguajes ya populares.

3.1.6.2.1. Ventajas

- Facilita la creación de múltiples tareas para que a su vez estas sean ejecutadas de manera independiente.
- No requiere paso de mensajes, puesto en su mayoría trabajan con memoria compartida.
- Muchas de las operaciones con procesos y subprocesos las maneja directamente el compilador.

3.1.6.2.2. Desventajas

- Requiere por parte del programador el aprendizaje de un nuevo paradigma, reglas adicionales para programar, lo cual en muchos casos genera miedo al cambio.
- No existe un consenso generalizado para popularizar estos lenguajes, esto genera la discontinuidad en el soporte de los mismos.
- Como aplican una capa extra para lo que significa operaciones con procesos, acceso de memoria compartida entre otras cosas hace que en ocasiones el tiempo de ejecución no sea el esperado.

3.1.6.2.3. Algunos Lenguajes Paralélelos

- **Chapel**
- **X10**
- **High Performance Fortran**
- **Zilk**
- **ZPL**
- **Parlog**
- **NESL**
- **Linda**
- **C***

3.2. Estado del Arte

Con el paso del tiempo las necesidades de los seres humanos van cambiando con más frecuencia. Esto se hace más evidente cuando vemos la evolución en tecnologías, que lleva una relación muy estrecha con el tipo de problemas al que se plantea darle solución.

En computación, específicamente la de alto rendimiento se ha venido reflejando avances significativos que van desde el hardware, pasando por las diferentes arquitectura hasta llegar al software. Python se abre paso cada vez más entre la comunidad científica e ingeniería, en trabajos como el de *Introduction to Computing with Python* [14] y también *A Just-In-Time Parallel Accelerator for Python* [15] se pueden observar las características que permiten al lenguaje tener tal nivel de aceptación, entre las que destacan lo rápido que significa escribir código en Python y la disponibilidad de módulos y librerías que proporcionan gran número de funciones para abordar problemas de distinta índole.

Lo que respecta a la computación de alto rendimiento con Python, se puede decir que este lenguaje es relativamente nuevo en el sector, pero se ha destacado de manera notable logrando ser en este un lenguaje referente a la hora de querer escribir código paralelo. A medida que los sistemas de memoria distribuida se vuelven más comunes, también se hacen comunes implementaciones que permitan usar la comunicación de MPI, tal es el caso de PupyMPI que en *PupyMPI - MPI Implemented in Pure Python* [16] presenta las características de dicha implementación, además muestra puntos de

referencia contra implementaciones en C con MPI (Interfaz de Paso de Mensaje), destacando un rendimiento aceptable. Similar a esta se encuentran también implementaciones que se describen en artículos como *Evolution of the fast parallel molecular analysis library for C++ and python* [17]*, A Parallel Python library for nonlinear systems* [18] que muestran utilidad para el modelado molecular así como también para la resolución de sistemas no lineales por el método del gradiente de forma paralela respectivamente.

Las aplicaciones que tiene la computación paralela con el lenguaje de programación Python son variadas, va desde el campo de la meteorología y el estudio de los océanos como lo muestra *Investigating Read Performance of Python and NetCDF When Using HPC Parallel Filesystems* [19], dando un panorama bastante amplio de como procesar volúmenes grandes de datos meteorológicos mediante la computación paralela. Además de eso proporciona una idea de que patrones de lectura y tamaño son más eficaces cuando se utiliza la librería netCDF4-python.

En lo que respecta netamente con el rendimiento se puede observar en trabajos como *Scalable multimedia content analysis on parallel platforms using python* [20] donde se hace el uso de medidas para evaluación del desempeño de las plataformas paralelas que usan Python, teniendo en cuenta principalmente la escalabilidad de estas. Otros trabajos que evalúan el desempeño, desde el punto de vista del software son*: High-Performance Parallel Computations Using Python as High-Level Language* [21]*, High performance Python for direct numerical simulations of turbulent flows* [22]**,** *Parallel optimal choropleth map*

classification in PySAL [23] *y High-performance Python-C ++ bindings with PyPy and Cling* [24]*.* Donde además de abordar el tema del rendimiento se realiza una evaluación y comparación de múltiples librerías para la programación paralela con Python e incluso con librerías de otros lenguajes como C++.

3.3. El Lenguaje Python

Python es un lenguaje de programación sencillo y robusto que ofrece tanto la potencia como la complejidad de los lenguajes tradicionales compilados, junto con la facilidad de uso de lenguajes scripting e interpretados más sencillos[25]**.** Se trata de un lenguaje multiparadigma, ya que soporta la orientación a objetos, la programación imperativa, programación funcional, entre otros. Además destaca entre sus características el tipado dinámico del lenguaje, lo que implica que una variable puede tomar valores de distintos tipos.

3.3.1. Historia

La historia del lenguaje de programación Python se remonta a la década de 1980 [26]. Pese a que fue concebido a finales de 1980, su puesta en marcha se dio en diciembre de1989 por Guido van Rossum en el CWI de los países bajos como reemplazante del lenguaje de programación ABC capaz de manejar excepciones. Van Rossum es el autor principal de Python, su rol fue direccionar y mantener la filosofía con la que fue creado la que a día de hoy mantienen.

Python es un lenguaje de programación de uso general y de alto nivel ampliamente utilizado. Su filosofía de diseño enfatiza en la legibilidad del

código y su sintaxis permite a los programadores expresar conceptos en menos líneas de código, lo que sería imposible en otros lenguajes como c++ o Java [25]. Actualmente este se encuentra en su versión 3.5.2 y es administrado por Python Software Foundation, su popularidad ha ido en aumento debido a que se ha popularizado el Framework Django para desarrollo de aplicaciones web, este Framework está escrito en Python y está enfocado en el patrón de diseño modelo-vista- controlador (MVC).

3.3.2. Características

Aunque ya van más de tres décadas desde que el lenguaje Python vio la luz, la filosofía y principales características aún se mantienen, de estas podemos destacar las más relevantes.

3.3.2.1. Alto nivel

Con cada generación nueva de lenguajes que aparecen nos trasladamos a un nivel más alto, en su tiempo lenguajes como Assembly representaron un gran salto para quienes debían batallar con lenguaje máquina. Luego aparecieron lenguajes con FORTRAN, Pascal y C que llevaron la programación a otro plano y crearon la industria de desarrollo de software, estos lenguajes evolucionaron a los actuales lenguajes de sistemas compilados como C++ y Java [25].

Se pudo escalar aún más con los lenguajes de scripting interpretados como Perl y Python. Cada uno de estos lenguajes tiene estructuras de datos de un nivel superior que reduce el tiempo de desarrollo y las líneas de código necesarias.

3.3.2.2. Orientado a objetos

La orientación a objetos añade otra dimensión a los lenguajes estructurados y procedimentales, donde los datos y la lógica son elementos discretos de la programación [25]. En Python la orientación a objetos fue parte de su diseño desde el principio, diferenciándose de otros lenguajes de scripting que carecían de esta característica.

3.3.2.3. Escalable

Python promueve el diseño de código limpio, la estructura de alto nivel y el empaquetado de múltiples componentes. Todos los cuales ofrecen flexibilidad, consistencia y tiempos de desarrollo requerido más rápidos a medida que los proyectos se expanden en amplitud y alcance [25].

El término escalable se aplica al lenguaje de programación Python debido a que este proporciona bloques y estructuras básicas que permiten la construcción de aplicaciones, y a medida que esas aplicaciones crecen la arquitectura modular de Python permite un fácil mantenimiento y gestión de la misma.

3.3.2.4. Extensible

Un proyecto puede verse en la obligación de aumentar sus líneas de código, ya sea por la necesidad de incluir nuevas características o bien por corrección de errores y depuración. Esta condición hace que surja la necesidad de organizar mejor este código, es posible que pueda organizarlo lógicamente debido a sus entornos de programación estructurados u orientados a objetos, o mejor aún, crear “módulos”

agrupe código y atributos que sean fácilmente accesibles desde otros módulos.

Este tipo de extensibilidad proporciona a los desarrolladores en un proyecto la flexibilidad de agregar o personalizar sus herramientas para ser más productivos y que estas sean desarrolladas en un periodo de tiempo más corto [25]

3.3.2.5. Portable

Python se encuentra disponible para una variedad considerable de plataformas, el que este lenguaje sea multiplataforma lo hace altamente portable, dicho de otra manera aunque hay algunos módulos específicos para una plataforma, cualquier aplicación general de Python podría ejecutarse en cualquier plataforma con apenas algunas modificaciones [25]**.**

3.3.2.6. Fácil de Aprender

Python tiene relativamente pocas palabras reservadas, estructura simple y una sintaxis claramente definida. Estas características le permiten aprender el lenguaje en corto tiempo. No existe un esfuerzo extra desperdiciando en el aprendizaje de conceptos totalmente extraños o palabras claves desconocidas [25]**.**

3.3.2.7. Fácil de Leer

Se encuentran totalmente ausentes de la sintaxis de Python los símbolos habituales que se encuentran en otros lenguajes para acceder a

variables, definir bloques de código, etc. Estos incluyen el símbolo de dólar ($), punto y coma (;), entre otros.

Sin todas estas distracciones, el código de Python está mucho más claramente definido y mejor presentado a la vista de quien esté interesado en su lectura [25]**.**

3.3.2.8. Interpretado

Python se clasifica como un lenguaje interpretado, lo que significa que el tiempo de compilación ya no es un factor importante a tener en cuenta durante el desarrollo de aplicaciones. Tradicionalmente los lenguajes puramente interpretados son más lentos que los lenguajes compilados por que la ejecución no tiene lugar en el lenguaje binario nativo de un sistema. Sin embargo, como Java, Python es en realidad compilado por bytes, resultando en una forma intermedia más cercano al lenguaje máquina. Eso mejora el rendimiento de Python, pero le permite conservar todas las ventajas de los lenguajes interpretados [25]**.**

3.3.3. Aplicaciones

Una de las cualidades que destaca en el lenguaje de programación Python es la de ser un lenguaje de uso general, por tal razón, veremos aplicaciones de escritorio, aplicaciones web y muchos más aplicativos software que tienen en común el hecho de ser desarrollados a partir de este lenguaje. Otra razón por la cual Python se encuentra siendo usado en casi todas las áreas de la computación es la variedad de paquetes suministrados por terceros disponibles, que pueden ayudar a reducir el tiempo de codificación.

A continuación se detallan algunos de los campos en los que Python es aplicado:

3.3.3.1. Shell script

La automatización de procesos con pequeños scripts es quizá unos de los primeros usos que se le dieron a este lenguaje y hasta la fecha puede decirse que es el más recurrente. Vemos como plataformas como Linux y Mac OS incluyen el intérprete de este por defecto en su sistema, además framework enfocados en la auditoria web automatizan sus procesos con Python [26].

3.3.3.2. Web

La aplicación del lenguaje de programación Python en la web es la más reciente, en sus inicios fue usado tímidamente en ella, pero no de forma masiva. La aparición de los framework para el desarrollo de aplicativos Web y sus ventajas a la hora de crear aplicaciones en un tiempo relativamente más corto, hizo que este lenguaje cobrara relevancia en este sector, ya que Django es uno de los framework más usados actualmente, consta de un conjunto de componentes que facilitan el desarrollo y está completamente escrito en Python [27], [28]**.**

3.3.3.3. Científico

Por sí solo, con su biblioteca estándar Python no ofrece mucho en la computación de carácter científico, pero como se ha dicho una de sus características principales es que posee un gran número de módulos externos desarrollados por terceros, estos módulos potencian usos en los

que Python puro no es fuerte. Para el computo de ámbito científico destacan los módulos Numpy (modulo para manejo de vectores y matrices), Scipy (proporciona algoritmos matemáticos para Python) y matplotlib (biblioteca para la generación de gráficos). Python en conjunto de estas y otros módulos más puede significar una alternativa libre a software de licencia privativa como lo es Matlab [29].

3.4. Programación paralela en Python

La limitación física de los sistemas secuenciales, la necesidad de resolver problemas de alta complejidad computacional y los grandes volúmenes de datos que requieren procesamiento, motiva el uso de la programación paralela. Aunque Python por su naturaleza de ser interpretado puede decirse que será más lento en tiempos de ejecución comparado con lenguajes compilados, también cabe acotar que el hecho de que este lenguaje tenga una sintaxis sencilla y clara facilitara la escritura y ahorrara tiempo en la codificación. Puede sumarse a esto la gran cantidad de librerías que posee Python para la computación científica e ingeniería, lo que lo convierte en un candidato a tener en cuenta a la hora de elegir un lenguaje para resolver tareas que necesiten paralelizarse.

3.4.1. Wrappers

3.4.1.1. PyCUDA

PyCuda es un wrapper de Python para CUDA desarrollado por Andreas Klöckner [30], su principal ventaja consiste en abstraer al programador de la gestión de memoria, así como presentar una sintaxis clara y

potentes estructuras de datos. CUDA es una arquitectura de cálculo paralelo desarrollada por NVIDIA que busca aprovechar la potencia de la GPU (Unidad de Procesamiento Grafico) para proporcionar un incremento considerable en el rendimiento general del sistema. CUDA ha sido recibida con mucha expectativa por la comunidad científica y ya se encuentra siendo usada en simuladores y demás.

3.4.1.1.1. Modelo CUDA

CUDA basa su modelo de paralelismo en tres puntos clave:

- **Jerarquía de Hilos**
  - Los hilos están contenidos dentro de bloques y estos a su vez dentro de grids.
  - Identificador del hilo threadIdx.
  - Identificador del bloque blockIdx.
- **Memoria Compartida**
  - Cada hilo posee su memoria local privada.
  - Cada bloque de hilos posee su memora compartida.
  - Todos los hilos pueden acceder a la memoria global.
  - Adicionalmente existen dos tipos de memorias: Memoria de texturas y memoria constante.
- **Sincronización por barrera**
  - Hilos en CUDA se ejecutan en device.
  - El resto del programa en el host.

- Espacios de memoria propios.

3.4.1.1.2. Ventajas

- Incrementa el número de núcleos computacionales.
- Provee de granularidad fina en el paralelismo de los datos e hilos.
- Extiende el lenguaje con un conjunto reducido de instrucciones.

3.4.1.2. PyOpenCL

La popularidad del lenguaje Python se ha disparado en años recientes, es fácil de entender este fenómeno debido a la simplicidad que representa a la hora de escribir código en todos los sistemas operativos.

Andreas Klöckner del instituto Courant de ciencias matemáticas ha ampliado las capacidades de Python al liberar PyOpenCL [31], lo novedoso de este Wrappers es que permite llevar la computación paralela de OpenCL a Python mediante su API.

3.4.1.2.1. Características y Ventajas

- Cuenta con una limpieza de objetos ligada a la vida misma de cada objeto, esto hace que sea más fácil escribir código correctamente.
- Pone todo el poder de la API de OpenCL a disposición de PyOpenCL.
- Comprobación de errores automática, todos los errores OpenCL traducirán en excepciones de Python.

- Posee gran velocidad, ya que la capa de PyOpenCL está escrita en C++.
- Cuenta con una documentación muy amplia y completa, además su licencia en libre

3.4.2. Extensiones

Existe variedad de implementaciones del lenguaje de programación Python, estas extensiones al lenguaje buscan dotarlo de características que el lenguaje por sí solo no posee. Entre las extensiones del lenguaje Python destacan CPython, IronPython y Jython.

3.4.3. Librerías

Python cuanta dentro de su biblioteca estándar con una gran cantidad y variedad de librerías que cubren un amplio espectro de la programación, aun así, para casos específicos en que estas librerías no satisfacen la necesidad del programador existe la posibilidad de usar librerías desarrolladas por terceros.

Para programación en paralelo estas librerías escasean dentro de la biblioteca estándar de Python sobre todo para MPI, sin embargo podemos hacer uso de aquellas que son desarrolladas fuera de Python sin que esto nos signifique un esfuerzo extra.

Entre las librerías más populares en Python para la programación en paralelo se encuentran:

3.4.3.1. Multiprocessing

Multiprocessing es un paquete que apoya a Python en el desarrollo de programas paralelos mediante una API muy similar a la encargada de manejar hilos en Python[35]. Esta librería dota al lenguaje de un paquete para la concurrencia tanto local como remota de manera efectiva mediante el uso de subprocesos.

Esta librería también introduce API´s que no tienen análogos. Un ejemplo de esto es el objeto que ofrece un medio conveniente para la ejecución de una función a través de múltiples valores de entrada y la posterior distribución de estos datos de entrada entre cada uno de tareas que lo necesitan.

3.4.3.2. MPI4PY

Este paquete proporciona una interfaz de alto nivel para ejecutar de forma asíncrona llamadas a tareas que se encuentran en un pool de procesos trabajo usando MPI para la comunicación entre procesos [36].

Usando MPI en Python e posible crear nuevos grupos de trabajo mediante un método intercomunicador, alternativamente los grupos de procesos separados pueden establecer comunicación entre ellos usando un enfoque Cliente/servidor.

Las versiones recientes de la librería añaden nuevas especificaciones entre ellas la inclusión de un modelo de gestión de procesos que proporciona una interfaz básica entre una aplicación y los recursos, con

esto se solvento la falencia de versiones anteriores en la cual una aplicación paralela era estática, es decir, no se podían agregar ni borrar procesos.

3.4.3.3. OpenMP

Python proporciona una interfaz para realizar procesamiento paralelo de memoria compartida a través de OpenMP. Esto permite escribir código paralelo de manera eficiente directamente en Python sin necesidad de tener que escribir contenedores en otros lenguajes como C/C++, Fortran, Java, entre otros [37].

OpenMP es una especificación para escribir programas multiprocesos, que incluye una serie de directivas de procesadores de C para gestionar cada subproceso. Entre las directivas que provee OpenMP se encuentran algunas que generan patrones de comunicación entre procesos, balanceo de carga y características de sincronización. Además de Python, la API de OpenMP se encuentra disponible para otros lenguajes que la implementan ales como C/C++ y Fortran.

3.5. La Librería Parallel Python (PP)

Parallel Python (pp) es un módulo Python que proporciona el mecanismo para la ejecución en paralelo de código Python en SMP (Sistemas de Múltiples Procesadores) y Clúster (Sistema interconectados vía red). Es ligero, fácil para la instalación y brinda la posibilidad de integrarse con otros programas escritos en Python [37].

Actualmente el software escrito en Python se encuentra en una gama amplia de categorías incluyendo los negocios, el análisis de datos y el cálculo científico. Esto con la amplia disponibilidad de ordenadores SMP y la facilidad de implementar Clústeres en el mercado, crea la demanda para la ejecución en paralelo de código Python que permita optimizar tiempos y recursos.

3.5.1. Características de Parallel Python

- Ejecución para lela de código Python en SMP (Multiproceso Simétrico) y Clúster.
- Facilidad para entender y poner en práctica la técnica.
- Detecta automáticamente la configuración más óptima.
- Asignación de procesos dinámicos, el número de procesadores trabajando puede ser cambiado en tiempos de ejecución.
- Posee equilibrio óptimo de carga.
- Proporciona seguridad, ya q posee métodos para la autenticación basada en SHA para las conexiones en red.
- Fácil portabilidad para sistemas Windows, Unix y Linux.
- Es Open Source (código Abierto).

4. Desarrollo de aplicaciones en paralelo con PP

4.1. Identificación del Problema a Paralelizar

La identificación del problema a paralelizar es el primer paso para poder escribir código que funcione de manera paralela. Cabe agregar que no todo problema es susceptible de paralelizar para ejecutar sobre un clúster, dado que este funciona bajo el paradigma de paralelismo de datos, se debe poder realizar un fraccionamiento de estos datos para que sean procesados de forma independiente en cada nodo presente en el clúster de alto rendimiento.

Aunque en la práctica todo problema en el que sus datos se puedan dividir sin que esto influya en la integridad de los mismos se puede paralelizar, también debe tenerse en cuenta la comunicación entre los nodos del clúster. Una comunicación deficiente deriva una mala configuración y esto se ve reflejado en el rendimiento general del sistema.

4.2. Análisis del Problema

Una vez identificado el problema con altas posibilidades de poder ser paralelizado, se debe pasar a una segunda etapa. En esta etapa se debe realizar un análisis muy detallado del problema para identificar cuales sectores deben ser tratados en paralelo y cuáles de forma secuencial.

Un buen apoyo es contar con una versión serial o secuencial del problema que se está tratando, si no se cuenta con ella se debería poder realizar preferiblemente. De esta manera se puede observar con mayor facilidad cuales son los sectores que generan cuello de botella perjudicando el rendimiento. Estos sectores críticos en el programa son los llamados a paralelizar, para ellos debe detallarse cada variable y su influencia en dicho sector.

4.3. Escribir el Programa en Paralelo

Teniendo identificado el problema a paralelizar y habiendo hecho un análisis exhaustivo del mismo, la codificación de este debería ser una tarea relativamente más sencilla. En parallel python se debe tener en cuenta que los sectores críticos identificados al momento de llevarlos a la forma paralela se han de convertir en funciones de python y las variables influyentes en este sector pasaran a ser parámetros de esta función.

Dado que el paradigma sobre el que funciona un clúster con parallel python es el de paralelismo de datos, esto quiere decir que la función que acabamos de escribir será la tarea a ejecutar de manera independiente en cada nodo. Estos nodos reciben una parte del volumen original de datos a tratar y son procesados de manera independiente generando una respuesta según sea programada.

5. Aplicaciones

Como problema de aplicación se abordó el tratamiento de imágenes digitales, particularmente la segmentación de imágenes mediante técnicas de clustering. El clúster fue implementado en las instalaciones del laboratorio de ciencias computacionales de la universidad de pamplona (CICOM), su instalación y configuración se puede apreciar en la guía desarrollada para tal motivo **(Ver Anexo C)** Lo que se busca con la segmentación de una imagen es simplificar o cambiar la representación de la misma, de tal manera que sea más significativa y más fácil de analizar. Existen variedad de técnicas de clustering. En este caso se usó la técnica llamada K-Means o K-Medias, la cual es descrita a continuación.

5.1. K-Means

Es un algoritmo creado por MacQueen en 1967 [38], es el algoritmo de clustering más conocido y utilizado ya que es de muy simple aplicación pero muy eficaz. Se caracteriza por seguir un procedimiento simple de clasificación de un conjunto de objetos en un determinado número K de clústeres, K determinado a priori [39].

A continuación se muestra la descripción del algoritmo básico:

- Seleccionar centroides, donde es el número de clúster deseado.
- Asignar cada punto al centroide más cercano y cada colección de puntos asignados a un centroide es un clúster.
- Actualizar los centroides de cada clúster, basados en los puntos asignados al clúster.

- Repetir el proceso de asignación y actualización hasta que ningún punto cambie de clúster, o lo que es lo mismo, hasta que los centroides permanezcan iguales.

5.2. Desarrollo de la Aplicación

Teniendo en cuenta que la segmentación de imágenes puede aplicarse a muchos campos, donde se busca la solución de problemas de la vida cotidiana, y teniendo en cuenta que el método K-Means, es un algoritmo que consume considerables recursos computacionales, se ve la necesidad de desarrollar una versión paralela del mismo que se ejecute sobre un clúster de alto rendimiento. En este caso se contó previamente con una versión secuencial del mismo, lo que facilito el análisis y se identifica el sector que genera altos consumos de procesamiento. La versión del código K-Means secuencial se muestra a continuación:

```
#Iportamos las librerías y módulos a utilizar
import numpy as np
import cv2
from time import time
tiempo_i = time()
#Leemos la imagen a procesar
img = cv2.imread('h22kyudai.jpg')
Z = img.reshape((-1,3))
# La convertimos a np.float32
Z = np.float32(Z)
# definimos criteria, numero  clusters(K) y applicamos
kmeans() de la librería opencv
criteria = (cv2.TERM_CRITERIA_EPS +
cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 5
ret,label,center=cv2.kmeans(Z,K,criteria,10,cv2.KMEANS_RAN
DOM_CENTERS)
# Ahora convertimos de nuevo a uint8, y construimos la
imagen original
```

```
center = np.uint8(center)
res = center[label.flatten()]
res2 = res.reshape((img.shape))
tiempo_f = time()
tiempo = tiempo_f - tiempo_i
tiempo = tiempo / 60
cv2.imwrite('Imagen_nueva.jpg',res2)
print ("Tiempo: ",tiempo, "Para K=",K)
```

El llamado a la función cv2.kmeans es el más crítico y por lo tanto el llamado a paralelizar, así que para la versión paralela se toma esa porción de código y se incluye en una nueva función que será ejecutada por separado en cada uno de los nodos y esta recibirá solo una parte de la imagen original. Así la versión paralelizada del anterior queda resumida de la siguiente forma:

```
#importamos las lbrerias a utilizar
import numpy
import cv2
import sys
import pp
#leemos la imagen
img = cv2.imread('h22kyudai.jpg')
nombrei1 = "parte-1.jpeg"
nombrei2 = "parte-2.jpeg"
#partimos la imagen a la mitad, en este caso por q se
usaran 2 nodos
parte_1 = img[0:8486, 0:6000]
parte_2 = img[0:8486, 6000:12000]
#guardamos los trozos de imagen creados
cv2.imwrite(nombrei1, parte_1)
cv2.imwrite(nombrei2, parte_2)
#Leemos cada trazo de imagen
img1 = cv2.imread(nombrei1)
img2 = cv2.imread(nombrei2)
#convertimos cada trozo a np.float32
Z1 = img1.reshape((-1,3))
Z1 = numpy.float32(Z1)
Z2 = img2.reshape((-1,3))
Z2 = numpy.float32(Z2)
#declaramos una lista con las matrices representativas de
la imagen para enviar como parametros
```

```
inputs = (Z1, Z2)
criteria = (cv2.TERM_CRITERIA_EPS +
cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
#creamos la funcion que contiene el llamado al sector de
mayor consumo de recursos y retorna una matriz tratada
def kmeans(Z, criteria):
    print "ENTRE"
    K = 5
   ret,label,center=cv2.kmeans(Z,K,criteria,10,cv2.KMEANS_R
ANDOM_CENTERS)
    # se convierte de nuevo al original
    center = numpy.uint8(center)
    res = center[label.flatten()]
    return res
#creamos el servidor con los nodos que van a procesar
#En este caso se usa la funcion que detecta los nodos
activos con ("*")
ppservers = ("*",)
job_server = pp.Server(0, ppservers=ppservers)

#se declara una lista donde se guarda cada respuesta de
los nodos
lista = []
#Se le envia a cada nodo la parte de la imagen que debe
procesar y se almacena la respuestaen una lista
jobs = [(input, job_server.submit(kmeans, (input,
criteria), modules=("cv2","numpy", ))) for input in
inputs]
for input, job in jobs:
    print ("Procesando...")
    result = job()
    print ("RESULT DENTRO: ", result)
    lista.append(result)

#Recuperamos cada parte de la imagen trada almacenada en
la lista
#para parte 1
res = lista[0]
res2 = res.reshape((img1.shape))
cv2.imwrite("1.png",res2)
#para parte 2
res = lista[1]
res2 = res.reshape((img2.shape))
cv2.imwrite("2.png",res2)
#leemos las imagenes q creamos y las concatenamos para
luego almacenarla
img_1 = cv2.imread("1.png")
```

```
img_2 = cv2.imread("2.png")
vis = numpy.concatenate((img_1, img_2), axis=1)
cv2.imwrite("IMAGEN_FINAL.jpeg", vis)
#imprimimos las estadísticas del cluster
job_server.print_stats()
```

Los resultados obtenidos en la ejecución del programa en el clúster y el análisis de los mismos, se presentan en la sección de evaluación del rendimiento y análisis de resultados.

6. Evaluación del Rendimiento

La principal motivación para escribir programas paralelos suele ser la obtención de un mejor rendimiento, pero surge la siguiente pregunta, ¿Cómo podemos evaluar nuestro sistema?

Existen muchas formas de evaluar el rendimiento de un sistema, una de ellas es considerar situaciones donde una carga de trabajo fija se puede repartir entre un diferente número de procesadores, en este caso particular la plataforma a evaluar es un clúster de alto rendimiento que se sometió a prueba mediante la ejecución de varios algoritmos. Los factores a tener en cuenta para evaluar el sistema son el de SpeedUp (Aceleración o Ganancia), la eficiencia y la escalabilidad.

6.1. ¿Cómo Determinar el Rendimiento?

6.1.1. Speed Up (Aceleración)

Para medir el aumento del rendimiento en el sistema que puede obtenerse con la inclusión de alguna mejora realizada, se utiliza la ley de Amdahl [40], nos dice que la ganancia global del sistema está dada por :

$$GananciaGlobal = \frac{Tiempodeejecucionsinmejora}{Tiempodeejecucionconmejora}$$

El SpeedUp nos permite saber cuánto se acelera el sistema al incluir nuevos nodos o unidades de procesamiento, para efectos de la programación paralela es necesario tener en cuenta algunas precisiones. El tiempo de ejecución sin mejora hace referencia al tiempo que se toma el sistema en ejecutar algún programa secuencialmente, y el tiempo con mejora es básicamente el tiempo demorado en ejecutar el mismo problema de manera paralela, así tendríamos lo siguiente:

$$S(n) = Ganancia(n) = \frac{Tiempo(1)}{Tiempo(n)}$$

Donde n es el número de nodos usados en la ejecución del programa en paralelo.

6.1.2. Eficiencia

La eficiencia mide la porción útil del trabajo realizado por cierta cantidad de unidades de procesamiento [3], en otras palabras la eficiencia se encarga de determinar cuál es el grado de aprovechamiento que está teniendo el sistema. *La eficiencia del sistema* para un sistema con *n* procesadores se define como:

$$E(n) = \frac{S(n)}{n} = \frac{Tiempo(1)}{n * Tiempo(n)} \leq 1$$

Donde *n* es el número de procesadores presentes en el sistema, en este caso particular el número de nodos presentes en el clúster. La eficiencia es

una comparación del grado de ganancia conseguido frente al valor máximo. La eficiencia más baja *E(n)=0* corresponde al caso en que todo el programa se ejecuta en un único procesador, la eficiencia máxima *E(n)=1* se obtiene cuando todos los procesadores del sistema están siendo completamente utilizados durante todo el periodo que tarde la ejecución.

6.1.3. Escalabilidad

Un sistema se dice que es escalable para un determinado rango de procesadores, si la eficiencia E(n) del sistema se mantiene constante y en todo momento por encima de un factor 0.5 [41]. Normalmente todos los sistemas tienen un número determinado de procesadores a partir del cual empiezan a disminuir su eficiencia de una forma más o menos abrupta. Se puede decir entonces que un sistemas es más escalable que otro si este número de procesadores, a partir del cual la eficiencia disminuye, es menor que el del otro sistema. Es muy frecuente confundir el término escalable con ampliable. Un sistema es ampliable si físicamente se le puede adicionar más hardware, que un sistema sea ampliable no significa que sea escalable, pues la eficiencia no tiene por qué mantenerse constante al ampliar el sistema y por tanto podría no ser escalable [41].

6.2. Resultados de la Evaluación

Para poder realizar un análisis más acertado sobre los datos, se hicieron series de 30 ejecuciones de cada programa **(Ver Tablas en Anexo D)**. Estas series se repitieron sucesivamente a medida que se le adicionan nodos hasta ocupar todos los disponibles en el clúster. Teniendo el tiempo resultado por cada

iteración se calculó el tiempo promedio que tarda en ejecutarse cada programa en función del número de nodos involucrados, para determinar el rendimiento de este en factores tales como: Aceleración, eficiencia y escalabilidad.

6.2.1. Diferenciación Automática

Descripción:

Este programa calcula las sumas parciales de $f(x) = X - \frac{X^2}{2} + \frac{X^3}{3} + \frac{X^4}{4}$ ...

Y sus primeras derivadas utilizando para ello el método de diferenciación automática en el límite $f(x) = ln(X+1) y f'(x) = \frac{1}{X+1}$.

La diferenciación automática es una herramienta informático-matemática muy potente para el cálculo de cualquier tipo de derivada de funciones. Entre sus ventajas respecto a otras formas de calcular derivadas están su precisión, eficiencia y sencillez de implementación **(Ver implementación en Anexo B)**.

Resultados:

Los resultados obtenidos al probar este programa se muestran a continuación.

En la Figura (4), se muestra un gráfico de los tiempos de ejecución por nodos para diferenciación automática. En este se observa un decaimiento exponencial del tiempo con el aumento del número de nodos involucrados.

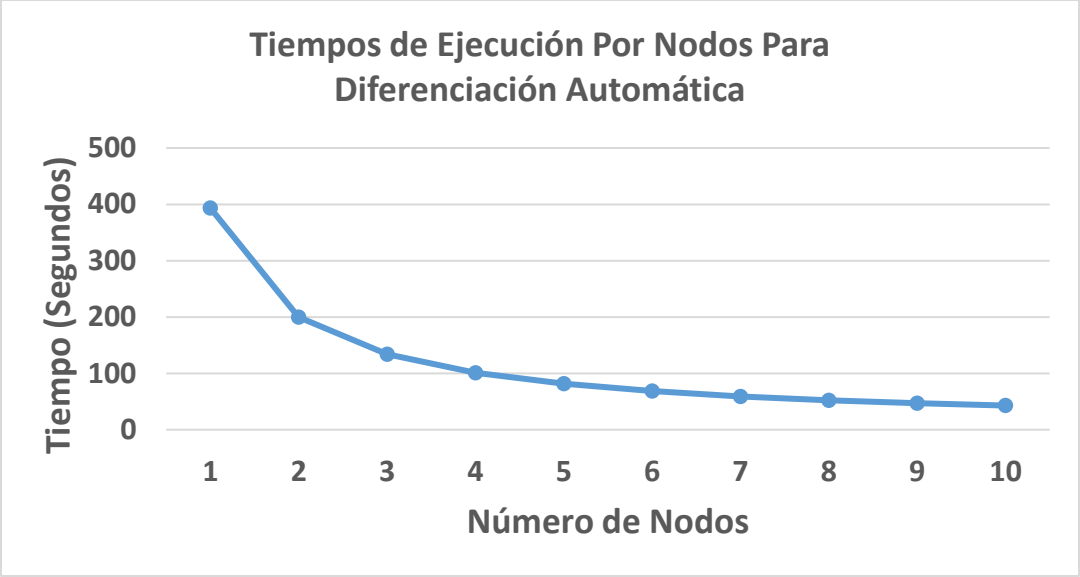

Figura (7). Variación del tiempo de ejecución para diferenciación automática

En la Figura (5), se muestran los resultados de la variación de la aceleración en función de los nodos para diferenciación automática. Se puede observar que esta aceleración es muy cercana a la ideal.

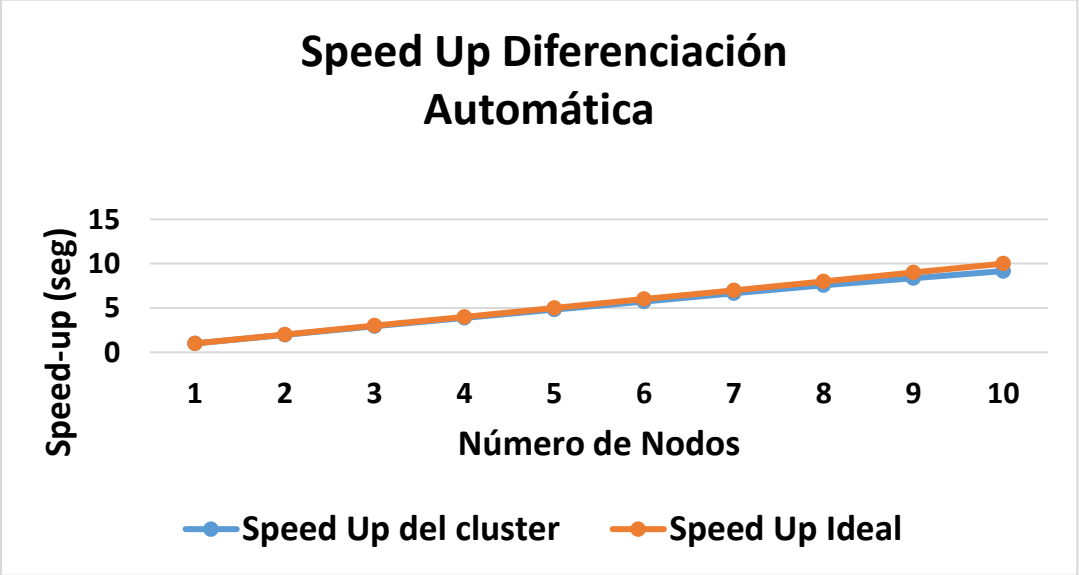

Figura (8). Variación de la aceleración para diferenciación automática

En la Figura (6), se muestra el resultado para diferenciación automática. Se puede observar que a medida que aumenta el número de nodos presentes, la eficiencia disminuye.

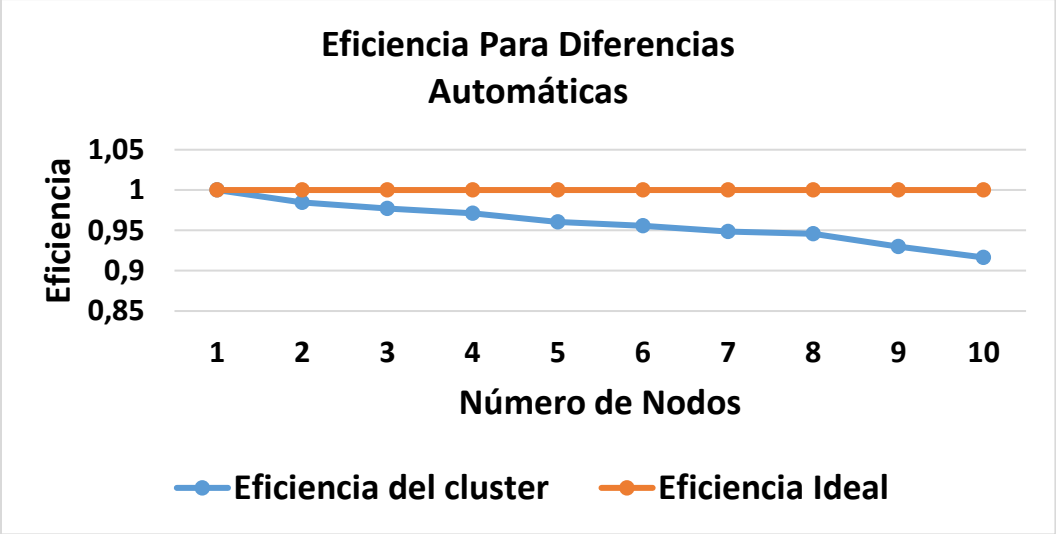

Figura (9). Variación de la eficiencia para diferenciación automática

6.2.2. Inverso de Hash MD5

Descripción:

En este programa lo que se pretende es invertir el orden de un hash generado en Md5, este es un proceso usado frecuentemente en los procesos de decodificación y descifrado de claves seguras. El MD5 es un algoritmo de reducción criptográfico de 128 bits ampliamente usado, este algoritmo recibe una cadena de texto y devuelve un hash codificado.**(Ver implementación en Anexo B)**

Resultados:

Los resultados obtenidos al probar este programa se muestran en las siguientes figuras.

En la Figura (7), se muestra un gráfico de los tiempos de ejecución por nodos para Hash Md5 invertido. Se observa una disminución del tiempo con aumento del número de nodos.

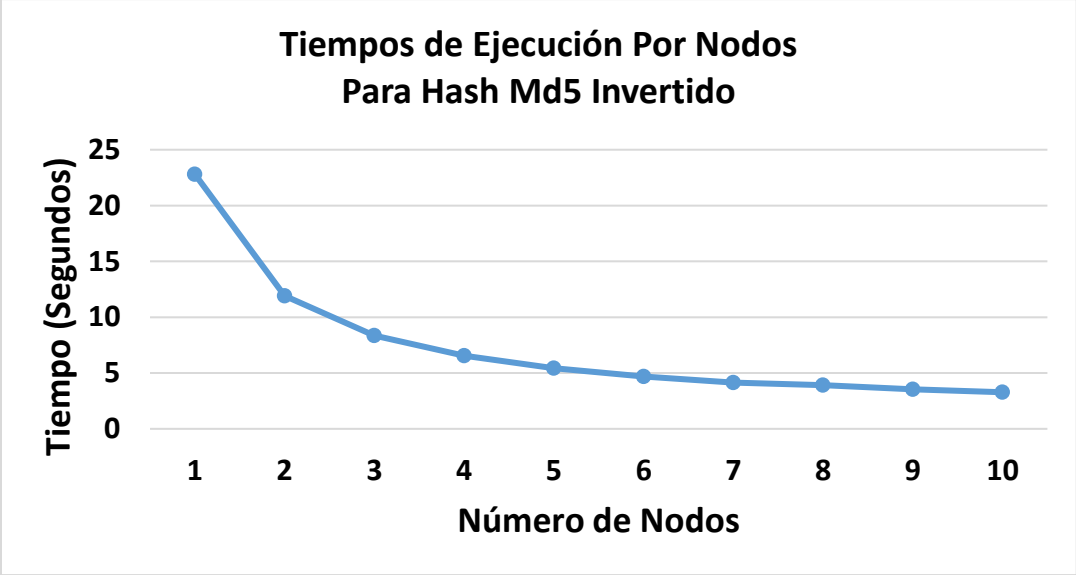

Figura (10). Variación del tiempo de ejecución para hash md5 invertido

En la Figura (8), se muestra el resultado de la variación de la aceleración en función del número de nodos para hash Md5 invertido. Se observa a partir del nodo 4 que la aceleración empieza a alejarse de la ideal.

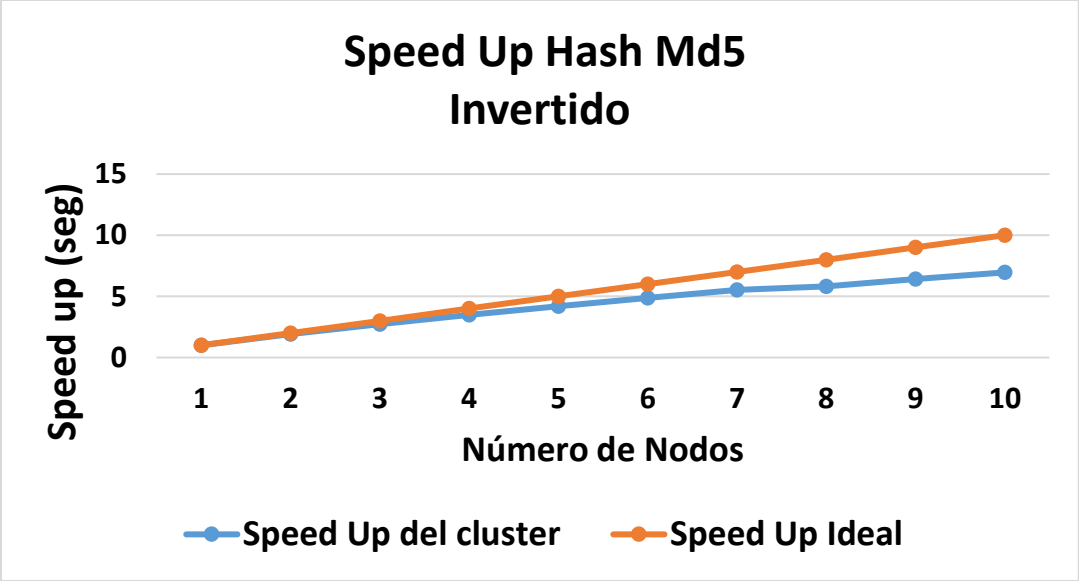

Figura (11). Variación de la aceleración para hash Md5 invertido

En la Figura (9), se muestra el resultado de la eficiencia en función del número de nodos para hash md5 invertido. Puede verse en la gráfica una ligera diferencia entre la eficiencia ideal y la alcanzada en la práctica

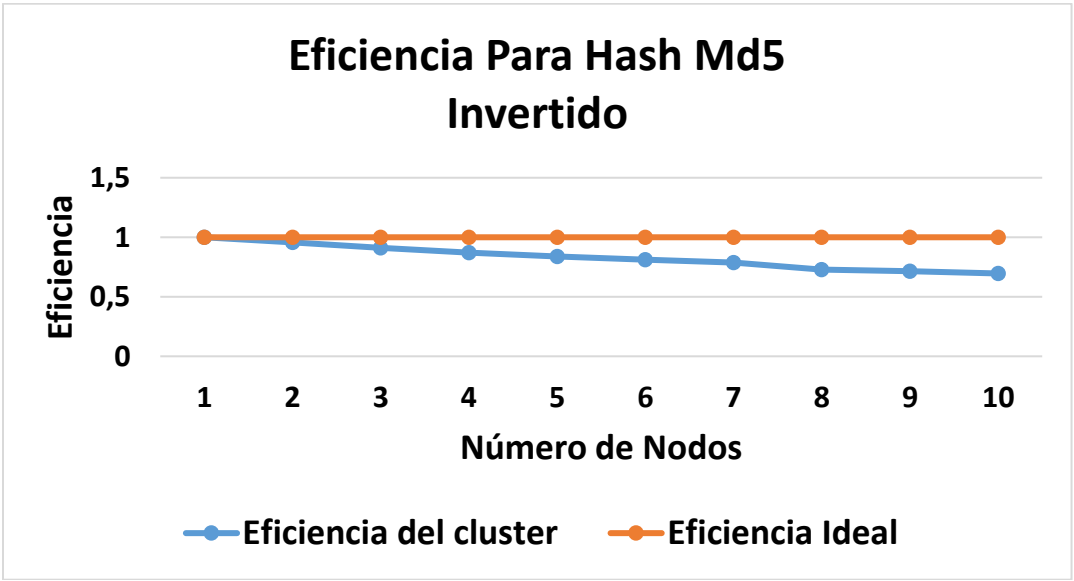

Figura (12). *Variación de la eficiencia para hash md5 invertido*

6.2.3. Suma de Números Primos

Descripción:

Este algoritmo realiza la suma de todos los números primos encontrados hasta un número determinado, el cual es definido previamente consta básicamente de dos funciones, una determina si un número es primo o no y la otra realiza la suma basada en la información provista por la función antes descrita. Para aportar mayor complejidad se pueden proporcionar grandes volúmenes de datos agrupándolos en una estructura de control propia de Python (Listas) **(Ver implementación en Anexo B)**.

Resultados:

Los resultados obtenidos al probar este programa se muestran en las siguientes figuras.

En la Figura (10), se muestra la variación del tiempo en función del número de nodos para la suma de números primos. A mayor número de nodos presentes el tiempo disminuye.

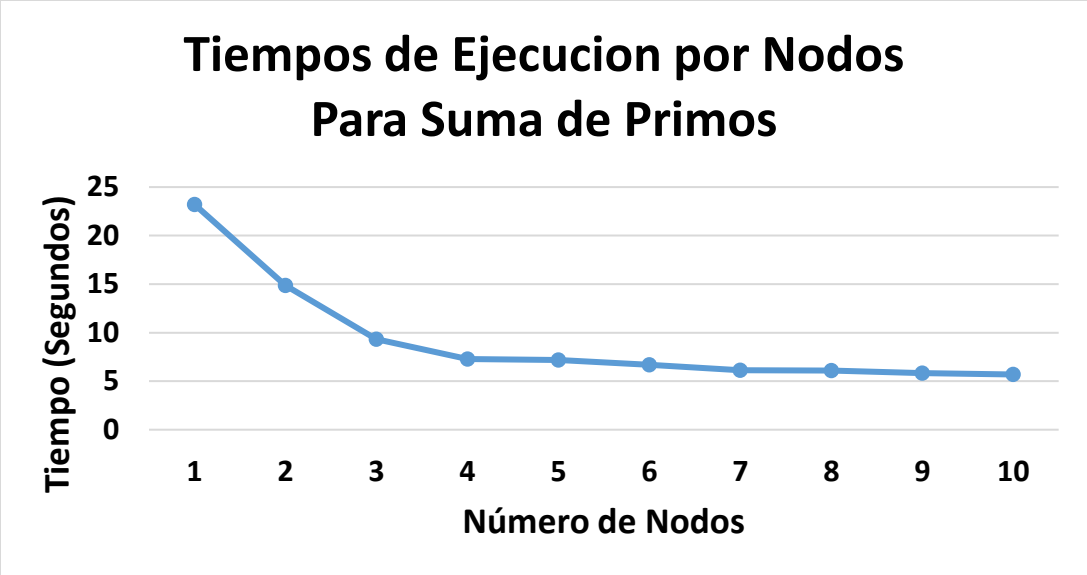

Figura (13). Variación del tiempo de ejecución para suma de números primos

En la Figura (11), se muestra de la aceleración en función del número de nodos para suma de números primos. Se puede observar que a partir del nodo 5 la aceleración empieza a disminuir considerablemente

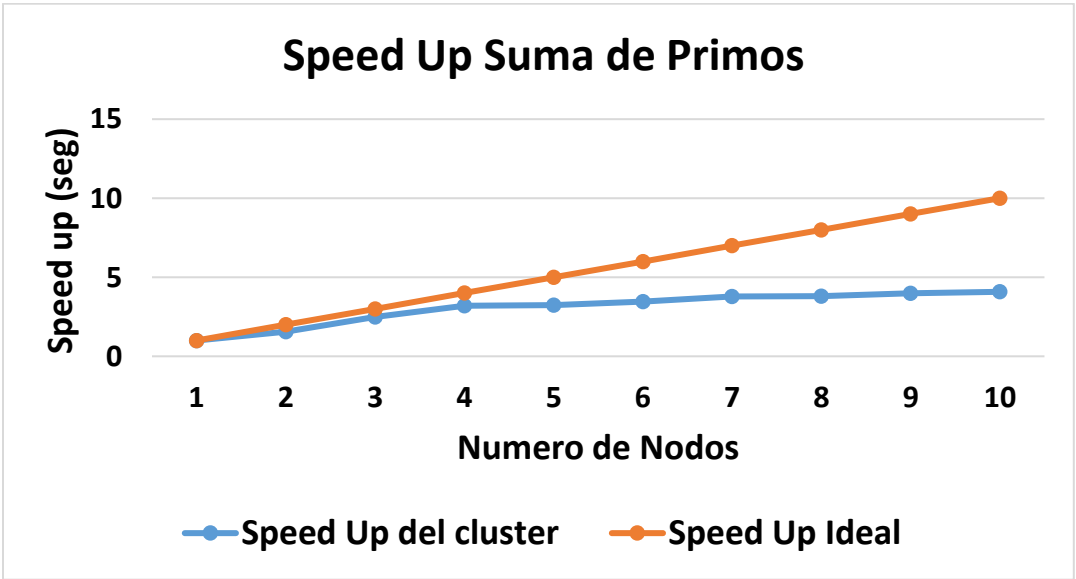

Figura (14). *Variación de la aceleración para suma de números primos*

En la Figura (12), se muestra la variación de la eficiencia en función del número de nodos, para suma de números primos. Se puede observar la tendencia a disminuir a medida que se agregan nodos.

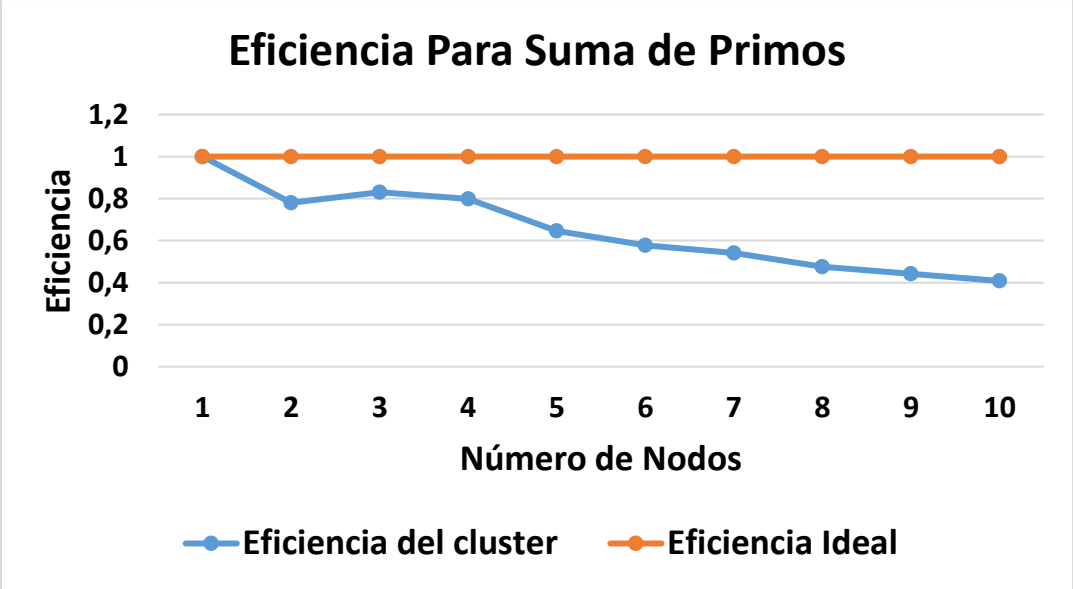

Figura (15). Variación de la eficiencia para suma de números primos.

6.2.4. Sumas Parciales

Descripción:

Este programa calcula la suma parcial de $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \cdots$ y el límite es Ln(2). Las sumas parciales son comúnmente también conocidas como series infinitas y son en sí mismas una sucesión de sumas sin terminar, por ello se hace necesario describir un límite **(Ver implementación en Anexo B)**.

Resultados:

Los resultados obtenidos al probar este programa se muestran en las siguientes figuras.

En la Figura (13), muestra la variación del tiempo en función del número de nodos, para sumas parciales. Se puede observar disminución en los tiempos de ejecución, aunque a mayor cantidad de nodos presenta una tendencia a presentar tiempos similares

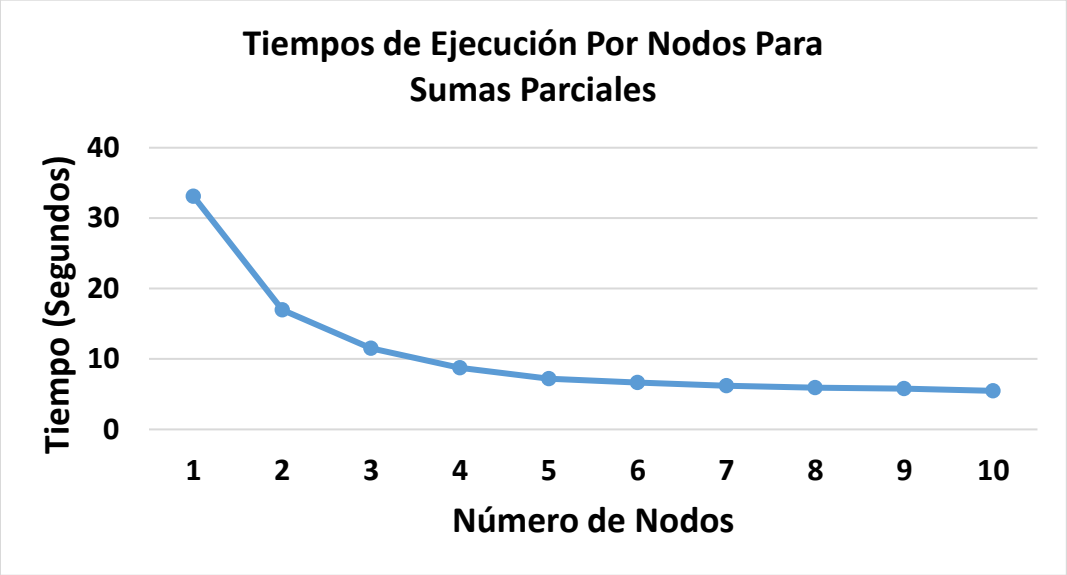

Figura (16). *Variación del tiempo de ejecución para sumas parciales.*

En la Figura (14), muestra la variación de la aceleración en función del número de nodos, para sumas parciales. A partir del nodo 5 se puede ver una tendencia al alejamiento de la aceleración ideal.

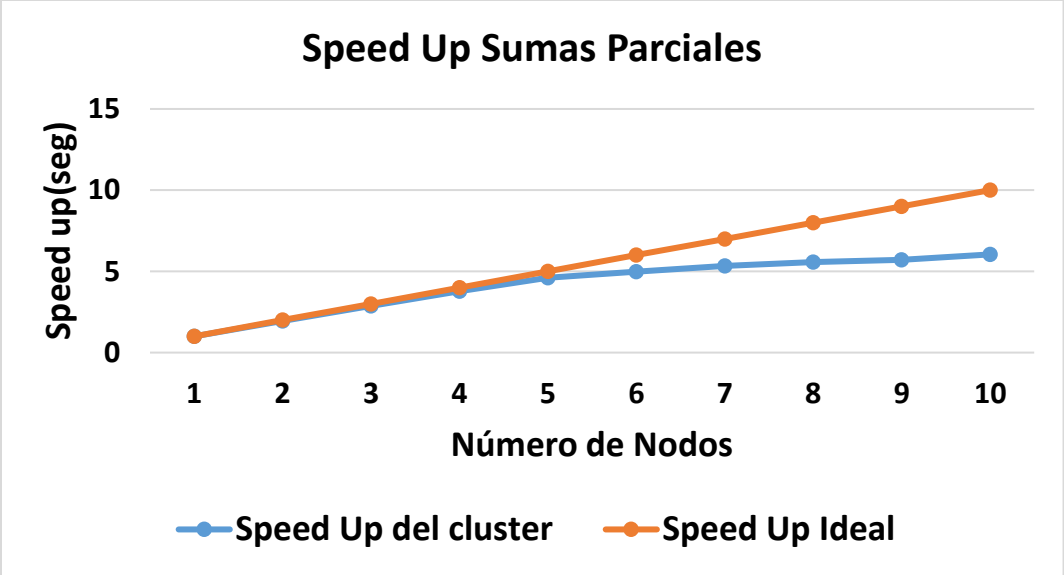

Figura (17). *Variación de la aceleración para sumas parciales*

En la Figura (15), muestra la variación de la eficiencia en función del número de nodos, para sumas parciales. En ella puede observarse como desde el nodo 5 va disminuyendo la aceleración.

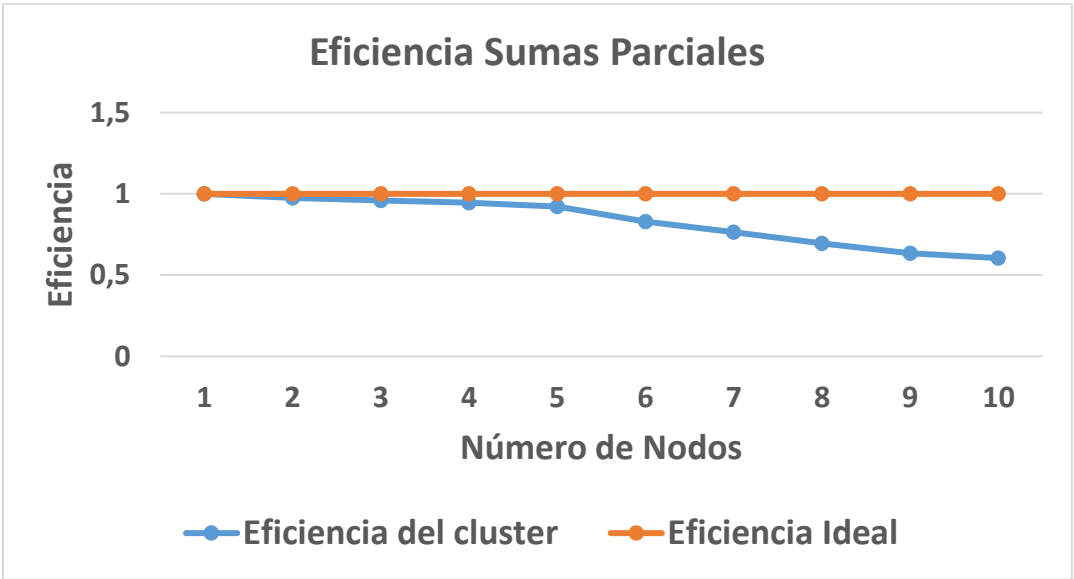

Figura (18). *Variación de la eficiencia para sumas parciales*

6.2.5. Ordenamiento de Vectores por Quicksort

Descripción:

Este es uno de los métodos de ordenamiento más comunes y está basado en la técnica de divide y vencerás. El Quicksort trabaja eligiendo un elemento de la lista el cual llama pivote, luego va reordenando el resto de elementos a cada lado del pivote de tal manera que de un lado queden los elementos menores a él y del otro los mayores.

En esta implementación se pretende que este método ordene una lista de una longitud considerablemente grande para añadirle complejidad y evaluar el comportamiento del clúster con datos muy grandes, este vector se divide en partes más pequeñas que serán enviadas a procesarse en cada uno de los nodos del clúster**(Ver implementación en Anexo B)**.

Resultados:

Los resultados obtenidos al probar este programa se muestran en las siguientes figuras.

En la Figura (16), muestra la variación del tiempo en función del número de nodos para Quicksort. Se observa como los tiempos presentan un comportamiento irregular a medida que se adicionan nodos.

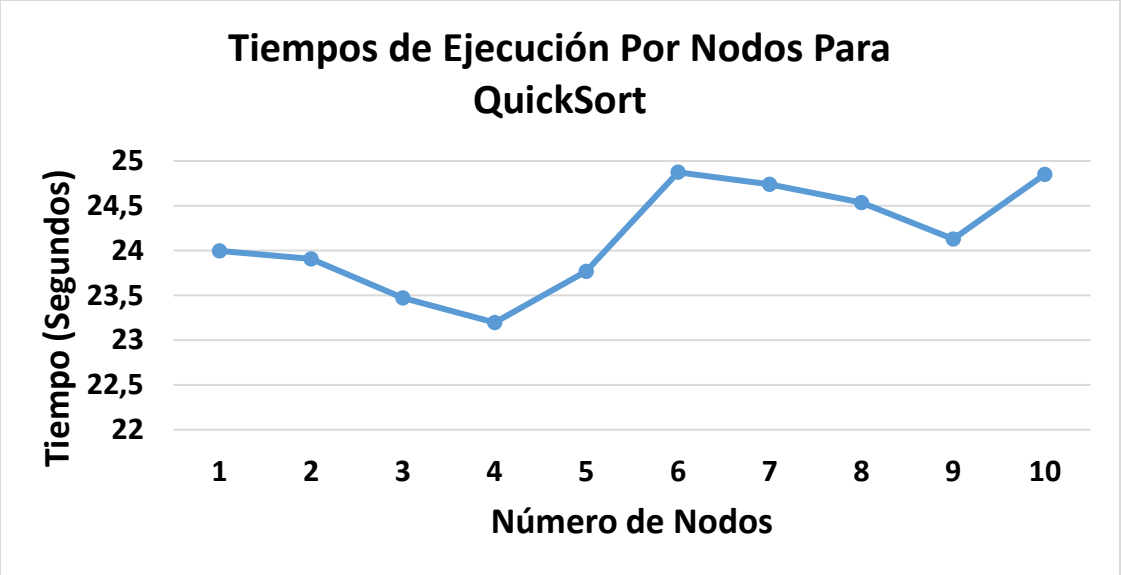

Figura (19). *Variación del tiempo de ejecución para Quicksort*

En la Figura (17), muestra la variación de la aceleración en función del número de nodos para Quicksort. Se puede observar en esta que la aceleración no sufre cambios notables.

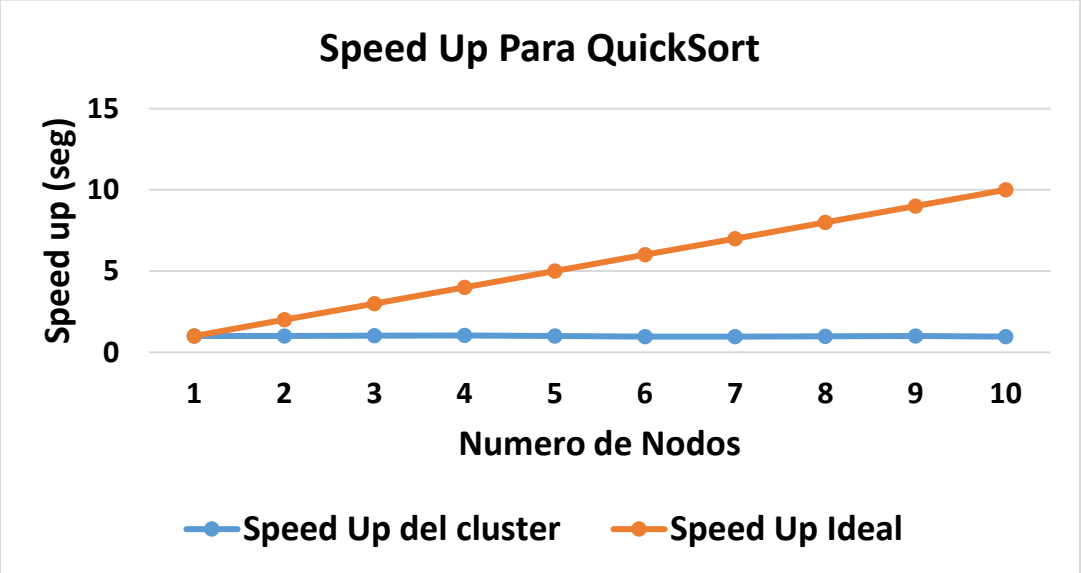

Figura (20). *Variación de la aceleración para Quicksort*

En la Figura (18), muestra la variación de la eficiencia en función del número de nodos para Quicksort. La grafica nos presenta una pérdida de eficiencia a medida que aumentamos el número de nodos.

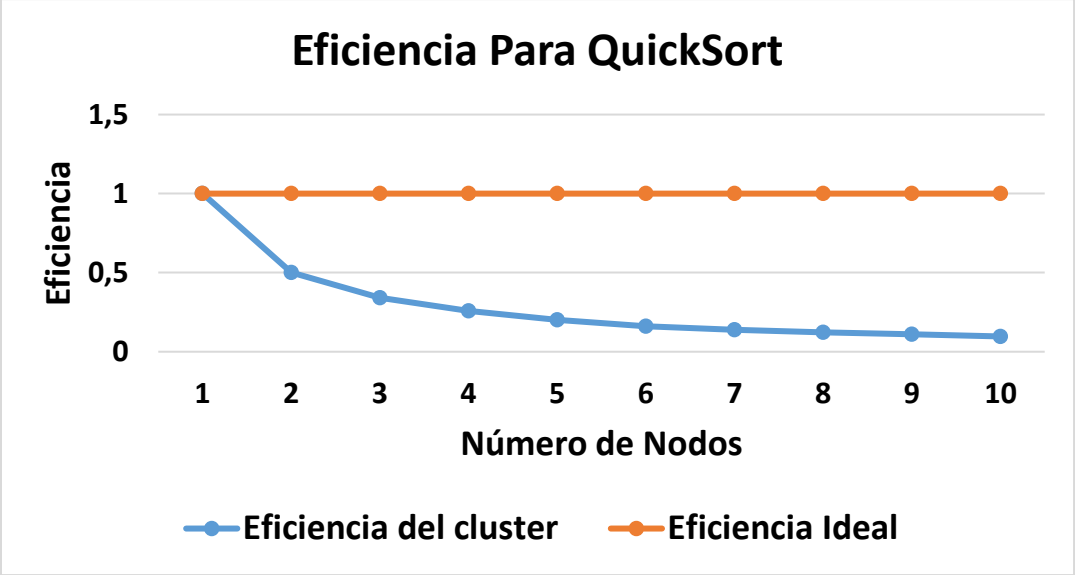

Figura (21). *Variación de la eficiencia para Quicksort*

6.2.6. Segmentación de imágenes con K-Means

Resultados:

Para el caso de la segmentación de imágenes con K-Means, se probaron dos versiones paralelizadas del algoritmo, a continuación se muestran los resultados de cada versión.

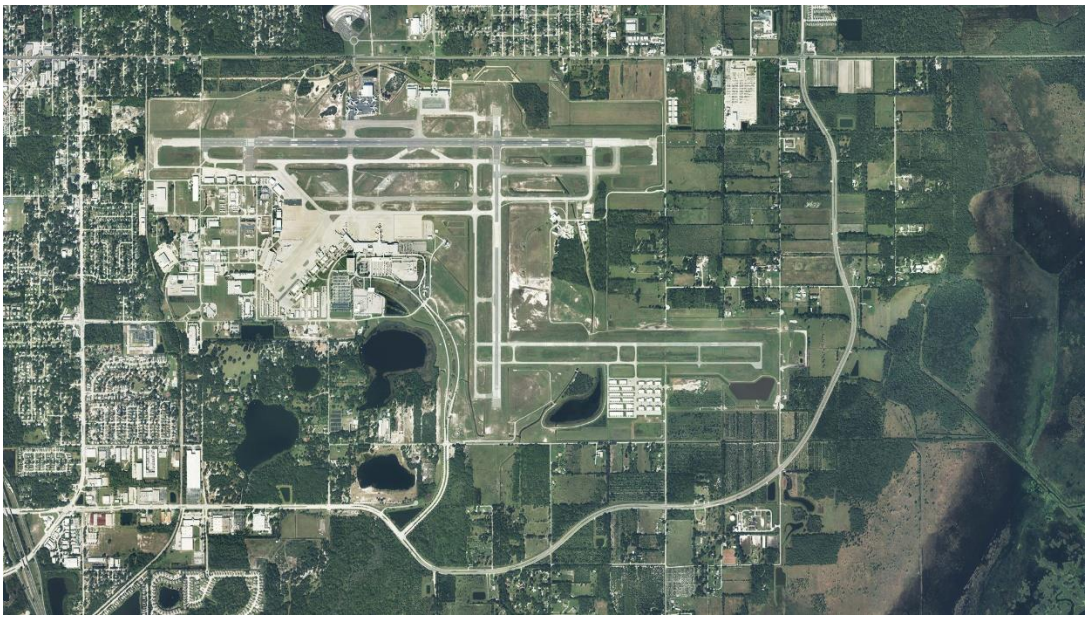

Figura 22 *Imagen satelital original*

Se trataron imágenes satelitales, a las cuales se les aplico el algoritmo de k-means. En la Figura (22), se puede ver la imagen original y posteriormente en la Figura () se observa el resultado obtenido luego de la clasificación de la imagen.

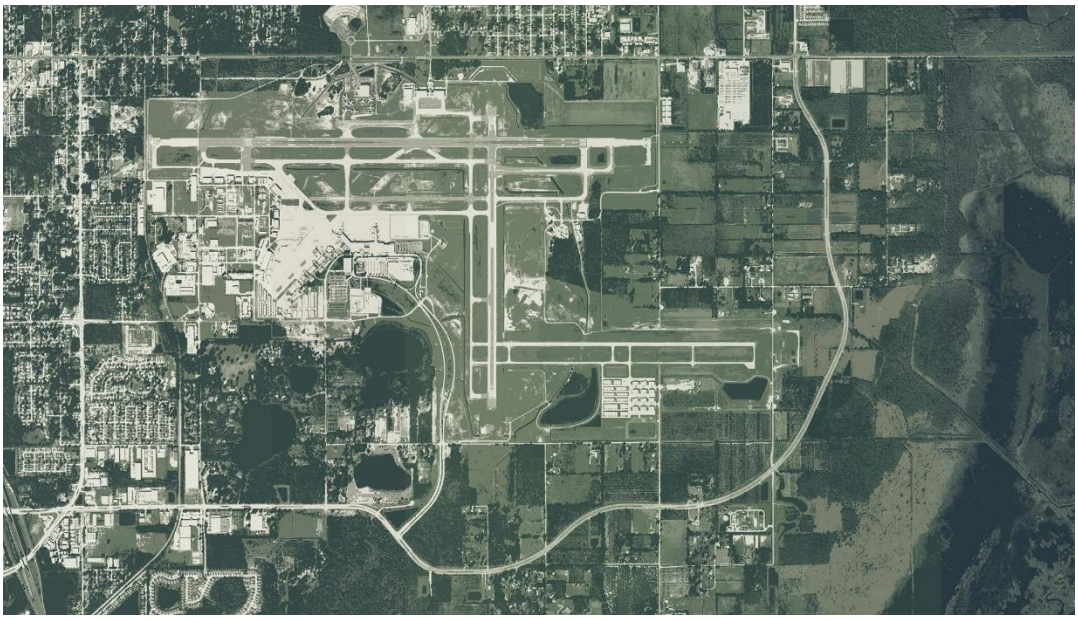

Figura 23 *Imagen satelital procesada con k-means*

Parte (I)

En la Figura (19), se muestra la variación del tiempo en función del número de nodos para K-Means. Puede observarse que al adicionar un segundo nodo, el tiempo aumenta considerablemente.

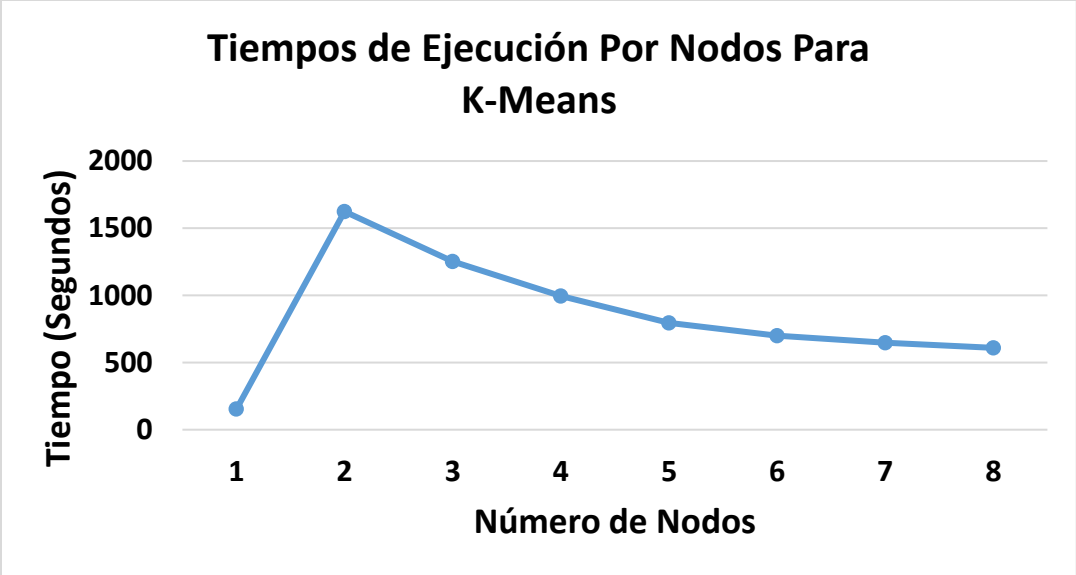

Figura (24). *Variación del tiempo de ejecución para K-Means*

En la Figura (20), se muestra la variación de la aceleración en función del número de nodos para K-Means. Se puede ver como al adicionar el segundo nodo la aceleración decae para permanecer estable posteriormente.

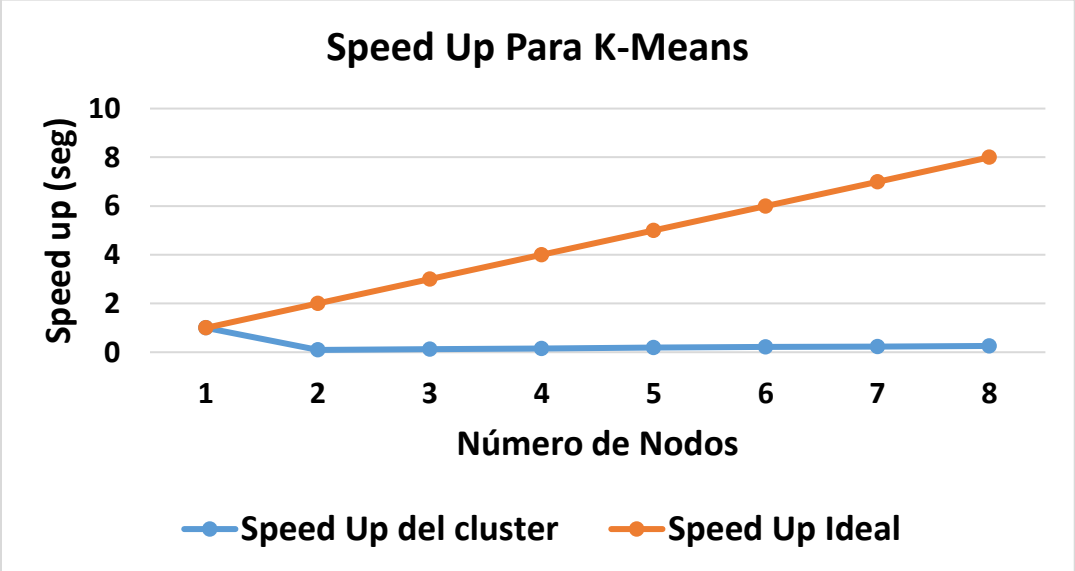

Figura (25). Variación de la aceleración para K-Means

En la Figura (21), se muestra la variación de la eficiencia en función del número de nodos para K-Means. Se puede ver como al agregar nodos la eficiencia decae hasta casi llegar a cero.

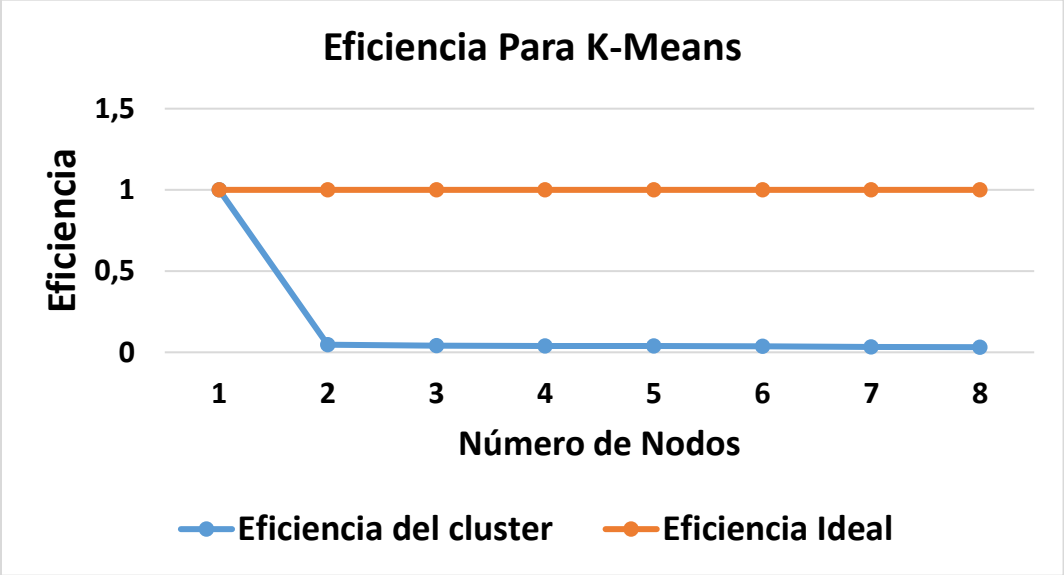

Figura (26). *Variación de la eficiencia para K-Means*

Parte (II)

Los resultados que se muestran a continuación derivan de la modificación realizada al algoritmo de K-Means, el cambio consistió en eliminar por completo el envío de fracciones de la imagen. La ausencia de envío se suple teniendo la imagen original en cada nodo y a estos se les asigna con anterioridad que sección de la misma deben procesar.

En la Figura (22), se muestra la variación del tiempo en función del número de nodos para K-Means (II). Puede observarse que a diferencia de los datos para la versión anterior este presenta una tendencia a disminuir tiempos.

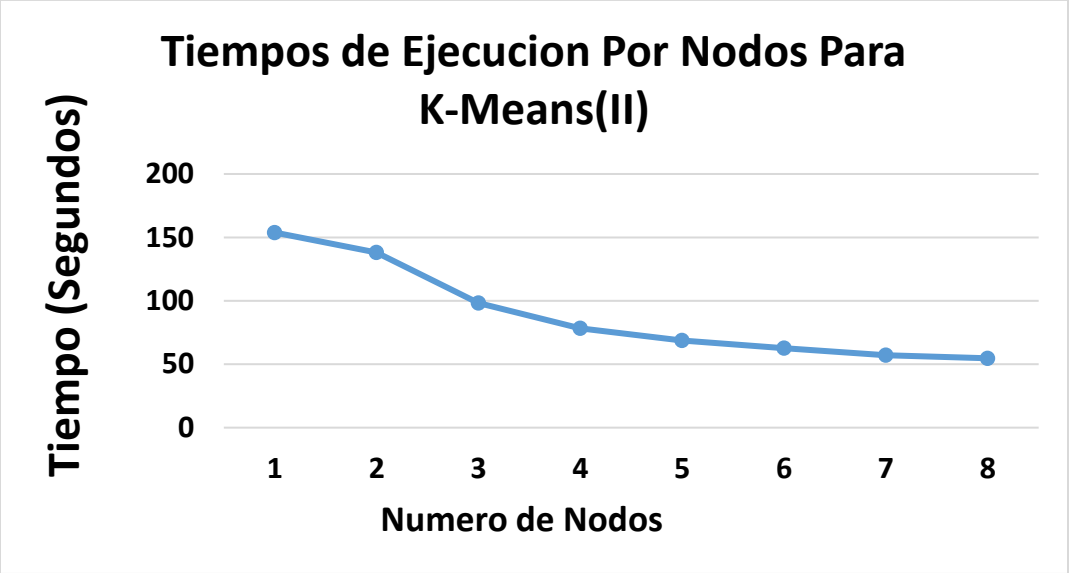

Figura (27). *Variación del tiempo de ejecución para K-Means (II)*

En la Figura (23), se muestra la variación de la aceleración en función del número de nodos para K-Means (II). Se presencia una mejoría con respecto a la versión anterior

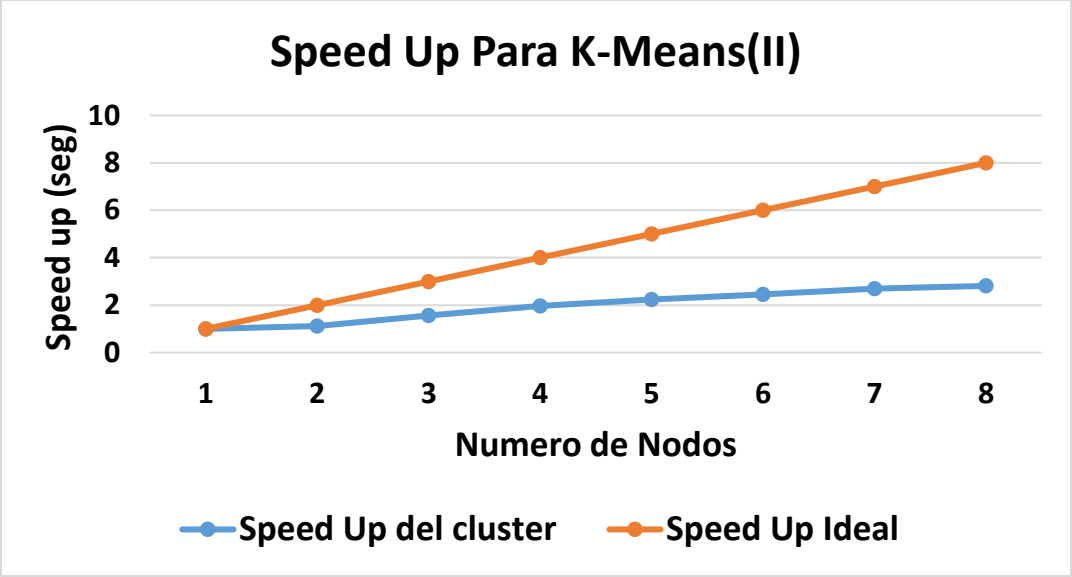

Figura (28). *Variación de la aceleración para K-Means (II)*

En la Figura (24), se muestra la variación de la eficiencia en función del número de nodos para K-Means (II). Se observa que la eficiencia disminuye abruptamente al adicionar el segundo nodo, a partir del tercero en adelante se estabiliza, pero con una tendencia a disminuir.

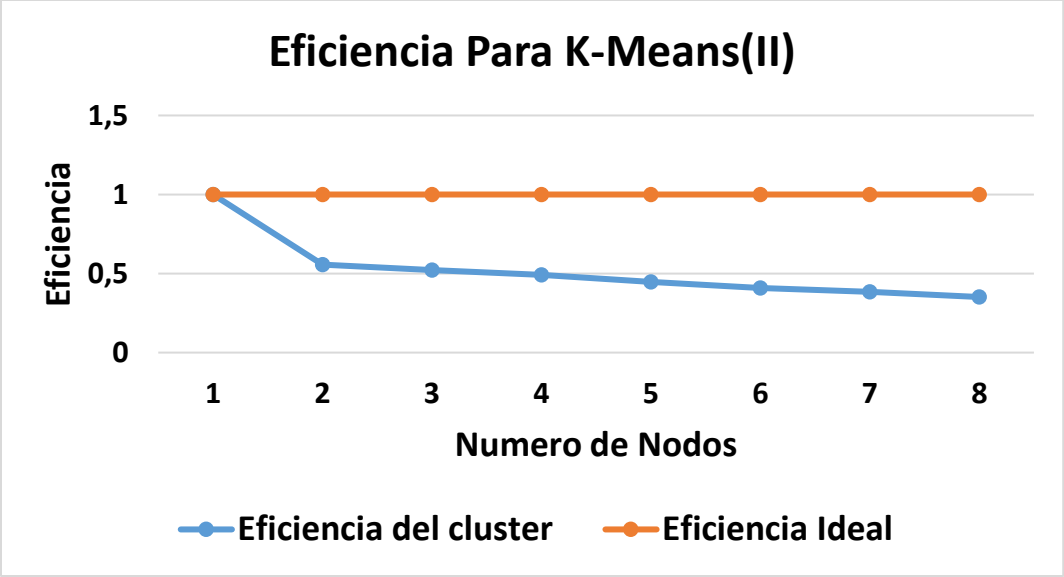

Figura (29). *Variación de la eficiencia para K-Means (II)*

7. Análisis de Resultados

Esta investigación tuvo como finalidad evaluar el desempeño de un clúster de alto rendimiento usando computación paralela con Python, usando para ello la librería “Parallel Python”. Sobre todo se pretendía probar problemas de diferente tipo para observar el comportamiento del sistema con cada uno de ellos y posterior a ello realizar una evaluación del desempeño teniendo en cuenta varias medidas de rendimiento como Speed up, eficiencia, entre otros.

De los resultados obtenidos en las pruebas realizadas, es posible determinar cuál ha sido el rendimiento, además de caracterizar el tipo de problemas más

susceptibles a ejecutar sobre esta plataforma para poder obtener los mejores resultados y proponer sugerencias para aquellos que no proporcionan buen rendimiento.

Tabla 1 Resultados comparativos de celeridad

	Número de Nodos									
Programa	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
Diferenciación Automática	**1**	**1,96**	**2,93**	**3,88**	**4,8**	**5,73**	**6,64**	**7,56**	**8,36**	**9,16**
Hash md5	**1**	**1,91**	**2,73**	**3,48**	**4,19**	**4,87**	**5,51**	**5,81**	**6,42**	**6,96**
Suma de primos	**1**	**1,56**	**2,48**	**3,19**	**3,22**	**3,46**	**3,78**	**3,8**	**3,97**	**4,07**
Sumas parciales	**1**	**1,94**	**2,87**	**3,78**	**4,6**	**4,98**	**5,34**	**5,56**	**5,71**	**6,04**
Quicksort	**1**	**1**	**1,02**	**1,03**	**1**	**0,96**	**0,96**	**0,97**	**0,99**	**0,96**

En la Tabla (1), se puede observar el comportamiento de la celeridad del sistema desde que lo ejecutamos con un solo nodo hasta 10 nodos. Puede notarse en ellas que los resultados más cercanos a la celeridad óptima se obtienen al adicionar pocos nodos y a medida que se aumentan, la celeridad baja la proporción en la que esta crece. Por otro lado, se observa un comportamiento contrario en la ejecución del programa Quicksort. Para este programa en particular puede verse que existen momentos en que se desacelera.

Tabla 2 Resultados comparativos de eficiencia

	Numero de Nodos									
Programa	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
Diferenciación Automática	**1**	**0,98**	**1**	**0,97**	**0,96**	**0,96**	**0,95**	**0,95**	**0,93**	**0,92**
Hash md5	**1**	**0,95**	**0,9**	**0,87**	**0,83**	**0,81**	**0,78**	**0,72**	**0,71**	**0,69**
Suma de primos	**1**	**0,78**	**0,8**	**0,79**	**0,64**	**0,57**	**0,54**	**0,47**	**0,44**	**0,4**
Sumas parciales	**1**	**0,97**	**1**	**0,94**	**0,92**	**0,83**	**0,76**	**0,69**	**0,63**	**0,6**
Quicksort	**1**	**0,5**	**0,3**	**0,25**	**0,2**	**0,16**	**0,13**	**0,12**	**0,11**	**0,09**

En lo que respecta a la eficiencia del sistema, se observa un caso bastante similar al anterior. El mismo grupo de problemas presentan un comportamiento bastante aceptable en cuanto a eficiencia se refiere, como puede verse en la Tabla (2). Caso opuesto al problema de ordenamiento por Quicksort, donde se marca una tendencia a cero. Teniendo en cuenta que la eficiencia mide el grado en que se están aprovechando los recursos computacionales, puede decirse que los problemas del primer grupo antes descrito están aprovechando de manera aceptable los recursos del clúster, mientras que por otra parte se observa que sistema no funciona de manera eficiente con algún tipo de problemas.

Como particularidad de los algoritmos en los cuales el sistema presentó los resultados más deficientes de aceleración y eficiencia, estos son problemas que requerían para su procesamiento grandes volúmenes de datos. Él envió de estos datos por medio de la red es un proceso que regularmente consumía el mayor número de tiempo del total necesitado para darle solución al problema. En el caso del algoritmo de segmentación de imágenes con la técnica K-Means se realizó una modificación con el objetivo de eliminar el proceso del envió de imágenes por la red, la modificación consistió en que cada uno de los nodos tenga en su directorio una copia de la imagen. Con este sutil cambio, cada uno de los nodos debe centrarse solo en leer y procesar la parte de la imagen correspondiente, como resultado se obtuvo mejores registros en los tiempos de ejecución en función del número de nodos, así como tambien en aceleración y eficiencia como lo muestran las Figuras (22, 23, 24).

Al ir aumentando sistemáticamente el número de nodos que se involucran en el clúster, se puede ver la tendencia de disminuir la eficiencia del mismo. Esto nos muestra que hasta cierto punto puede considerarse escalable, considerando que la escalabilidad de es un factor que depende de la eficiencia del sistema el cual debe mantenerse por encima de 0,5, algo que en su mayoría se cumple salvo los problemas nombrados anteriormente que requieren envió de volúmenes grandes de datos para procesar.

8. Conclusiones

Durante el desarrollo del presente trabajo de hizo una revisión de los principales tópicos involucrados en la computación de alto rendimiento o HPC (por sus siglas en inglés, Hight Performance Computing), con el objeto de obtener una visión clara del área, posteriormente se ha enfocado este trabajo en revisar los antecedentes de la utilización del lenguaje Python, tanto en programación paralela, como en computación de alto rendimiento. Una revisión del estado del arte mostró que son varios los investigadores que han utilizado este lenguaje y sus herramientas para la computación paralela y el HPC, obteniendo resultados prometedores en varios campos de aplicación, con una evaluación del desempeño comparable al obtenido utilizando lenguajes tradicionales en el área como C/C++.

Para el contexto específico de este trabajo, se ha diseñado e implementado un clúster de alto rendimiento **(Ver implementación en Anexo C)**, tipo beowulf, utilizando el hardware disponible en el laboratorio de Ciencias Computacionales, el cual fue configurado para utilizar la librería Parallel Python. Se puede destacar al respecto, que la instalación y configuración del software es bastante sencilla, en comparación con otras tecnologías relacionadas tal como OpenMPI, lo cual puede ser una ventaja atractiva para investigadores interesados, con conocimientos apenas elementales en sistemas operativos. Aunque el clúster fue instalado sobre nodos con sistema operativo Linux, cabe destacar que es posible hacerlo, con diferentes sistemas operativos en cada nodo.

En el momento de estudiar los casos de prueba, se ha podido apreciar que el paso de una versión secuencial a un algoritmo paralelo es bastante sencillo, y detalles de sincronización y concurrencia son trasparente para el programador, lo cual puede ser también un atractivo para personas interesadas en realizar pruebas sobre sus algoritmos, sin invertir demasiado tiempo en capacitación.

Al evaluar el desempeño de las versiones secuenciales y en paralelo se pudo apreciar que debido a las prestaciones del hardware y posiblemente al tipo de conexión de red, los algoritmos con poco nivel de partición y/o distribución de datos mejoran el rendimiento al aumentar el número de nodos involucrados. En contraposición al ejecutar algoritmos, como el del caso de prueba, con una imagen de alta dimensión, en los cuales se requiere distribuir entre los nodos grandes cantidades de datos, el paso por la red de los mismos puede constituirse en un cuello de botella. Se pudo observar que el tiempo de procesamientos de los nodos

es eficiente y la carga computaciones distribuida en los mismos, disminuye el tiempo final, sin embargo, el proceso de transmisión de datos, termina restando eficiencia.

Para el caso particular desarrollado, se encontró una posible solución. Asignar previamente la imagen a procesar en cada nodo, y asignar desde el nodo maestro las regiones de la imagen a procesar por cada nodo esclavo, retornando al nodo maestro los resultados de la segmentación. De esta forma se ha logrado reducir el tiempo de transmisión de datos, y las medidas de rendimiento mejoraron notablemente.

Como conclusión final podemos afirmar, que aplicando las medidas de rendimiento seleccionadas en este trabajo tal como la Aceleración, Eficiencia y Escalabilidad, se ha determinado que la utilización de un clúster de alto rendimiento basado en la librería Parallel Python, mejora las prestaciones del hardware sobre el uso de algoritmos secuenciales en máquinas de las características de cada uno de sus nodos, con lo cual se tiene una alternativa, económica, fácil de configurar y aplicar para muchos problemas que requieren computación de alto rendimiento. En particular, para el caso de análisis de imágenes, se puede suplir el problema de los tiempos de distribución de datos por la red, agregando en cada nodo la imagen a analizar, para distribuir en cada nodo solo la carga del procesamiento de la misa, de tal forma que los resultados finales si puedan ser enviados de forma automática al nodo maestro.

Bibliografía

[1] R. Hernández, C. Fernández, and C. Baptista, *Metodología de la investigación*, vol. 53, no. 9. 2014.

[2] M. F. Piccoli, *Computación de alto desempeño en GPU*. 2011.

[3] P. S. Pacheco, *Introduction to Parallel Programming*. 2011.

[4] I. T. Stan Openshaw, *High Performance Computing and the Art of Parallel Programming: An Introduction for Geographers, Social Scientists and Engineers*, Routledge,., vol. 59, no. 2. 2013.

[5] V. Eijkhout, "Introduction to High Performance Scientific Computing," *Hpc*, p. 446, 2011.

[6] I. T. Foster, "Designing and Building Parallel Programs," *Interface*, no. Noviembre, pp. 1–209, 1995.

[7] M. J. Flynn, "Very High-Speed Computing Systems," *Proc. IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[8] C. S. Yeo, R. Buyya, H. Pourreza, R. Eskicioglu, P. Graham, and F. Sommers, "Cluster Computing: High-Performance, High-Availability, and High-Throughput Processing on a Network of Computers," *Handb. Nature-Inspired Innov. Comput. Integr. Class. Model. with Emerg. Technol.*, pp. 521–551, 2006.

[9] R. Buyya and others, "High performance cluster computing: Architectures and systems (volume 1)," *Prentice Hall, Up. SaddleRiver, NJ, USA*, vol. 1, p. 999, 1999.

[10] G. Cáceres, "Estrategia de Implementación de un Clúster de Alta Disponibilidad de N nodos sobre Linux usando Software Libre," p. 172, 2012.

[11] J. Manuel and G. Carrasco, "Herramientas Para Programacion Paralela."

[12] L. F. Hern and N. Anillo, "Desarrollo de aplicaciones con técnicas de programación paralela para el análisis del procesamiento 3D de imágenes de microscopía ."

[13] N. A. R. E, "Lenguajes Paralelos Contenido I na historia Ventajas y Desventajas Ventajas," 2013.

[14] R. Johansson, "Introduction to Computing with Python," in *Numerical Python*, Berkeley, CA: Apress, 2015, pp. 1–24.

[15] A. Rubinsteyn and others, "Parakeet: A just-in-time parallel accelerator for {Python}," *USENIX Conf. Hot Top. Parallelism*, p. 14, 2012.

[16] R. Bromer, F. Hantho, and B. Vinter, "pupyMPI - MPI Implemented in Pure Python," Springer Berlin Heidelberg, 2011, pp. 130–139.

[17] S. O. Yesylevskyy, "Pteros 2.0: Evolution of the fast parallel molecular analysis

library for C++ and python," *J. Comput. Chem.*, vol. 36, no. 19, pp. 1480–1488, Jul. 2015.

[18] H. Migallón, V. Migallón, and J. Penadés, "A Parallel Python library for nonlinear systems," *J. Supercomput.*, vol. 58, no. 3, pp. 438–448, Dec. 2011.

[19] M. Jones, J. Blower, B. Lawrence, and A. Osprey, "Investigating Read Performance of Python and NetCDF When Using HPC Parallel Filesystems," Springer International Publishing, 2016, pp. 153–168.

[20] E. Gonina, G. Friedland, E. Battenberg, P. Koanantakool, M. Driscoll, E. Georganas, and K. Keutzer, "Scalable multimedia content analysis on parallel platforms using python," *ACM Trans. Multimed. Comput. Commun. Appl.*, vol. 10, no. 2, pp. 1–22, Feb. 2014.

[21] S. Masini and P. Bientinesi, "High-Performance Parallel Computations Using Python as High-Level Language," Springer Berlin Heidelberg, 2011, pp. 541–548.

[22] M. Mortensen and H. P. Langtangen, "High performance Python for direct numerical simulations of turbulent flows," *Comput. Phys. Commun.*, vol. 203, pp. 53–65, 2016.

[23] S. J. Rey, L. Anselin, R. Pahle, X. Kang, and P. Stephens, "Parallel optimal choropleth map classification in PySAL," *Int. J. Geogr. Inf. Sci.*, vol. 27, no. 5, pp. 1023–1039, May 2013.

[24] W. T. L. P. Lavrijsen and A. Dutta, "High-performance Python-C ++ bindings with PyPy and Cling."

[25] W. Chun, *Core Python Programming Python核心编程*. 1989.

[26] B. Klein, "History of Python," *Python Course*, pp. 1–3, 2015.

[27] "The Web framework for perfectionists with deadlines | Django." [Online]. Available: https://www.djangoproject.com/.

[28] "Welcome to Python.org." [Online]. Available: https://www.python.org/.

[29] G. Borell Nogueras, "Python como entorno de desarrollo científico.," pp. 1–18, 2008.

[30] A. Klöckner, N. Pinto, B. Catanzaro, Y. Lee, P. Ivanov, and A. Fasih, "GPU Scripting and Code Generation with PyCUDA," *GPU Comput. Gems Jade Ed.*, pp. 373–385, 2012.

[31] M. Arbenz, *In Action*, no. November 2010. 2011.

[32] R. González Duque, "Python para todos," *Web B.*, no. 6, p. 108, 2000.

[33] M. J. Foord and C. Muirhead, *in a Ction*. .

[34] J. Juneau, J. Baker, V. Ng, L. Soto, and F. Wierzbicki, *Jython*. .

[35] “Multiprocesamiento - paralelismo basado en procesos - Python documentación.” [Online]. Available: https://docs.python.org/3/library/multiprocessing.html?highlight=multiprocessing#module-multiprocessing.

[36] “MPI for Python Lisandro Dalcin,” 2016.

[37] G. Lanaro, *Python High Performance Programming*. 2013.

[38] C. G. Cambronero and I. G. Moreno, “ALGORITMOS DE APRENDIZAJE: KNN & KMEANS [Inteligencia en Redes de Telecomuncicación].”

[39] G. Lorca, J. Arzola, and O. Pereira, “Segmentación de Imágenes Médicas Digitales mediante Técnicas de Clustering Digital Medical Image Segmentation with Clustering Techniques.”

[40] G. M. Amdahl, “1 Introduction,” pp. 1–4, 1967.

[41] “Capítulo 4. Introducción al paralelismo y al rendimiento. 4.1.,” pp. 1–23, 2012.

ANEXOS.

Anexo A. Tablas de los resultados de rendimiento

Tabla 3 Resultados obtenidos para diferenciación automática

Número de Nodos	Promedio (seg)	Speed Up (seg)	Eficiencia (seg)
1	393,33	1	1
2	199,77	1,968	0,984
3	134,15	2,93	0,977
4	101,25	3,88	0,971
5	81,9	4,8	0,96
6	68,6	5,73	0,955
7	59,22	6,64	0,948
8	51,99	7,56	0,945
9	47	8,36	0,929
10	42,91	9,16	0,916

Tabla 4 Resultados obtenidos para reverso de hash md5

Número de Nodos	Promedio (seg)	Speed Up (seg)	Eficiencia (seg)
1	22,8	1	1
2	11,91	1,91	0,95
3	8,35	2,73	0,91
4	6,54	3,48	0,87
5	5,43	4,19	0,83
6	4,68	4,87	0,81
7	4,13	5,51	0,78
8	3,91	5,81	0,72
9	3,54	6,42	0,71
10	3,27	6,96	0,69

Tabla 5 Resultados obtenidos para suma de números primos

Número de Nodos	Promedio (seg)	Speed Up (seg)	Eficiencia (seg)
1	23,2	1	1
2	14,87	1,56	0,78
3	9,32	2,48	0,82
4	7,26	3,19	0,79
5	7,19	3,22	0,64
6	6,7	3,46	0,57
7	6,13	3,78	0,54
8	6,09	3,8	0,47
9	5,83	3,97	0,44
10	5,69	4,07	0,4

Tabla 6 Resultados obtenidos para sumas parciales

Número de Nodos	Promedio (seg)	Speed Up (seg)	Eficiencia (seg)
1	33,1	1	1
2	16,98	1,94	0,97
3	11,5	2,87	0,95
4	8,75	3,78	0,94
5	7,18	4,6	0,92
6	6,45	4,98	0,83
7	6,19	5,34	0,76
8	5,94	5,56	0,69
9	5,79	5,71	0,63
10	5,47	6,04	0,6

Tabla 7 Resultados obtenidos para Quicksort

Numero de Nodos	Promedio (seg)	Speed Up (seg)	Eficiencia (seg)
1	23,99	1	1
2	23,9	1	0,5
3	23,47	1,02	0,34
4	23,19	1,03	0,25
5	23,76	1	0,2
6	24,87	0,96	0,16
7	24,73	0,96	0,13
8	24,53	0,97	0,12
9	24,12	0,99	0,11
10	24,89	0,96	0,09

Tabla 8 Resultados obtenidos para K-Means Parte I

Número de Nodos	Promedio (seg)	Speed Up (seg)	Eficiencia (seg)
1	153,83	1	1
2	1624,06	0,09	0,04
3	1253,18	0,12	0,04
4	996,32	0,15	0,03
5	795,73	0,19	0,03
6	701,04	0,21	0,03
7	648,29	0,23	0,03
8	609,47	0,25	0,03

Número de Nodos	Promedio (seg)	Speed Up (seg)	Eficiencia (seg)
1	153,83	1	1
2	138,04	1,11	0,55
3	98,27	1,56	0,52
4	78,32	1,96	0,49
5	68,68	2,23	0,44
6	62,61	2,45	0,4
7	57,1	2,69	0,38
8	54,68	2,82	0,35

Anexo B Códigos fuente de las implementaciones realizadas

1. Código fuente diferenciación automática

```
import math
import sys
import pp

class AD(object):

    def __init__(self, x, dx=0.0):
        self.x = float(x)
        self.dx = float(dx)

    def __pow__(self, val):
        if isinstance(val, int):
            p = self.x**val
            return AD(self.x**val, val*self.x**(val-
1)*self.dx)
        else:
            raise TypeError("Second argumnet must be an
integer")

    def __add__(self, val):
        if isinstance(val, AD):
            return AD(self.x+val.x, self.dx+val.dx)
        else:
            return AD(self.x+val, self.dx)

    def __radd__(self, val):
        return self+val

    def __mul__(self, val):
        if isinstance(val, AD):
            return AD(self.x*val.x,
self.x*val.dx+val.x*self.dx)
        else:
            return AD(self.x*val, val*self.dx)
```

```
    def __rmul__(self, val):
        return self*val

    def __div__(self, val):
        if isinstance(val, AD):
            return self*AD(1/val.x, -val.dx/val.x**2)
        else:
            return self*(1/float(val))
    __truediv__ = __div__ # PYTHON 3

    def __rdiv__(self, val):
        return AD(val)/self
    __rtruediv__ = __rdiv__ # PYTHON 3

    def __sub__(self, val):
        if isinstance(val, AD):
            return AD(self.x-val.x, self.dx-val.dx)
        else:
            return AD(self.x-val, self.dx)

    def __repr__(self):
        return str((self.x, self.dx))


class PartialSum(object):

    def __init__(self, n):

        self.n = n

    def t_log(self, x):

        return self.partial_sum(x-1)

    def partial_sum(self, x):
        return sum([float(i%2 and 1 or -1)*x**i/i for i in
range(1, self.n)])


print("""Usage: python auto_diff.py [ncpus]
    [ncpus] - the number of workers to run in parallel,
    if omitted it will be set to the number of processors in
the system
""")

ppservers = ()
```

```
if len(sys.argv) > 1:
    ncpus = int(sys.argv[1])
    # Creates jobserver with ncpus workers
    job_server = pp.Server(ncpus, ppservers=ppservers)
else:
    # Creates jobserver with automatically detected number of
workers
    job_server = pp.Server(ppservers=ppservers)

print("Starting pp with %s workers" % job_server.get_ncpus())

proc = PartialSum(20000)

results = []
for i in range(32):
    # Creates an object with x = float(i)/32+1 and dx = 1.0
    ad_x = AD(float(i)/32+1, 1.0)
    # Submits a job of calulating proc.t_log(x).
    f = job_server.submit(proc.t_log, (ad_x, ))
    results.append((ad_x.x, f))

for x, f in results:
    # Retrieves the result of the calculation
    val = f()
    print("t_log(%lf) = %lf, t_log'(%lf) = %lf" % (x, val.x,
x, val.dx))

# Print execution statistics
job_server.print_stats()
```

2. Código fuente inverso de hash md5

```
import math
import sys
import hashlib
import pp
if sys.version_info[0] >= 3:
    def b_(x):
        return x.encode('UTF-8')
else:
    def b_(x):
        return x


def md5test(hash, start, end):
```

```
    """Calculates md5 of the integers between 'start' and
'end' and
       compares it with 'hash'"""
    for x in range(start, end):
        if hashlib.md5(b_(str(x))).hexdigest() == hash:
            return x


print("""Usage: python reverse_md5.py [ncpus]
    [ncpus] - the number of workers to run in parallel,
    if omitted it will be set to the number of processors in
the system
""")

ppservers = ()

if len(sys.argv) > 1:
    ncpus = int(sys.argv[1])
    # Creates jobserver with ncpus workers
    job_server = pp.Server(ncpus, ppservers=ppservers)
else:
    # Creates jobserver with automatically detected number of
workers
    job_server = pp.Server(ppservers=ppservers)

print("Starting pp with %s workers" % job_server.get_ncpus())

#Calculates md5 hash from the given number
hash = hashlib.md5(b_("1829182")).hexdigest()
print("hash = %s" % hash)
#Now we will try to find the number with this hash value

start = 1
end = 2000000

parts = 128

step = int((end - start) / parts + 1)
jobs = []

for index in range(parts):
    starti = start+index*step
    endi = min(start+(index+1)*step, end)
    jobs.append(job_server.submit(md5test, (hash, starti,
endi),
            (), ("hashlib", "_hashlib")))
```

```
# Retrieve results of all submited jobs
for job in jobs:
    result = job()
    if result:
        break

# Print the results
if result:
    print("Reverse md5 for %s is %s" % (hash, result))
else:
    print("Reverse md5 for %s has not been found" % hash)

job_server.print_stats()
```

3. Código fuente suma de primos

```
import math
import sys
import pp


def isprime(n):
    """Returns True if n is prime and False otherwise"""
    if not isinstance(n, int):
        raise TypeError("argument passed to is_prime is not
of 'int' type")
    if n < 2:
        return False
    if n == 2:
        return True
    max = int(math.ceil(math.sqrt(n)))
    i = 2
    while i <= max:
        if n % i == 0:
            return False
        i += 1
    return True


def sum_primes(n):
    """Calculates sum of all primes below given integer n"""
    return sum([x for x in range(2, n) if isprime(x)])
```

```
print("""Usage: python sum_primes.py [ncpus]
    [ncpus] - the number of workers to run in parallel,
    if omitted it will be set to the number of processors in
the system""")

# tuple of all parallel python servers to connect with
ppservers = ()
#ppservers = ("127.0.0.1:60000", )

if len(sys.argv) > 1:
    ncpus = int(sys.argv[1])
    # Creates jobserver with ncpus workers
    job_server = pp.Server(ncpus, ppservers=ppservers)
else:
    # Creates jobserver with automatically detected number of
workers
    job_server = pp.Server(ppservers=ppservers)

print("Starting pp with %s workers" % job_server.get_ncpus())
job1 = job_server.submit(sum_primes, (100, ), (isprime, ),
("math", ))
result = job1()

print("Sum of primes below 100 is %s" % result)


# The following submits 8 jobs and then retrieves the results
inputs = (100000, 100100, 100200, 100300, 100400, 100500,
100600, 100700)
jobs = [(input, job_server.submit(sum_primes, (input, ),
(isprime, ),
        ("math", ))) for input in inputs]

for input, job in jobs:
    print("Suma de primos para %s es %s" % (input, job()))

job_server.print_stats()
```

4. Código fuente sumas parciales

```
import math
import time
try:
    import _thread as thread
```

```
except ImportError:
    import thread
import sys
import pp


class Sum(object):

    def __init__(self):
        self.value = 0.0
        self.lock = thread.allocate_lock()

    def add(self, value):
        """
        the callback function
        """
        # we must use lock here because += is not atomic
        self.lock.acquire()
        self.value += value
        self.lock.release()


def part_sum(start, end):
    """Calculates partial sum"""
    sum = 0
    for x in range(start, end):
        if x % 2 == 0:
            sum -= 1.0 / x
        else:
            sum += 1.0 / x
    return sum


print("""Usage: python callback.py [ncpus]
    [ncpus] - the number of workers to run in parallel,
    if omitted it will be set to the number of processors in
the system
""")

start = 1
end = 20000000

# Divide the task into 128 subtasks
parts = 128
step = int((end - start) / parts + 1)

ppservers = ()
```

```
if len(sys.argv) > 1:
    ncpus = int(sys.argv[1])
    # Creates jobserver with ncpus workers
    job_server = pp.Server(ncpus, ppservers=ppservers)
else:
    # Creates jobserver with automatically detected number of
workers
    job_server = pp.Server(ppservers=ppservers)

print("Starting pp with %s workers" % job_server.get_ncpus())
sum = Sum()

start_time = time.time()
for index in range(parts):
    starti = start+index*step
    endi = min(start+(index+1)*step, end)
    job_server.submit(part_sum, (starti, endi),
callback=sum.add)

job_server.wait()

# Print the partial sum
print("Partial sum is %s | diff = %s" % (sum.value,
math.log(2) - sum.value))

job_server.print_stats()
```

5. Código fuente ordenamiento por Quicksort

```
import sys, random
import pp
try:
    callable(min)
except NameError:
    def callable(x):
        import collections
        return isinstance(x, collections.Callable)

def quicksort(a, n=-1, srv=None):
    if len(a) <= 1:
        return a
    if n:
        return quicksort([x for x in a if x < a[0]], n-1,
srv) \
```

```
                + [a[0]] + quicksort([x for x in a[1:] if x
>= a[0]], n-1, srv)
    else:
        return [srv.submit(quicksort, (a,))]


print("""Usage: python quicksort.py [ncpus]
    [ncpus] - the number of workers to run in parallel,
    if omitted it will be set to the number of processors in
the system
""")

ppservers = ()

if len(sys.argv) > 1:
    ncpus = int(sys.argv[1])
    # Creates jobserver with ncpus workers
    job_server = pp.Server(ncpus, ppservers=ppservers)
else:
    # Creates jobserver with automatically detected number of
workers
    job_server = pp.Server(ppservers=ppservers)

print("Starting pp with %s workers" % job_server.get_ncpus())

n = 1000000
input = []
for i in range(n):
    input.append(random.randint(0,100000))

n = 5


outputraw = quicksort(input, n, job_server)

output = []
for x in outputraw:
    if callable(x):
        output.extend(x())
    else:
        output.append(x)

print("first 30 numbers in increasing order: %s" %
output[:30])

job_server.print_stats()
```

Anexo C. Descripción de la implementación del clúster

Guía De Instalación, Configuración y Uso de Un Clúster de Alto Rendimiento Usando la Librería de Python: “Parallel Python”.

Universidad de Pamplona

Año 2016

Introducción

Propósito del Documento

El presente documento está dirigido a entregar las pautas para la instalación, configuración y puesta en marcha de un clúster de alto rendimiento para computación paralela, usando para ello el lenguaje de programación python y el módulo Parallel Python.

Conceptos Importantes

¿Qué es un Clúster?

Un clúster es un tipo de sistema informático en paralelo o distribuido, que consiste en una colección de sistemas autónomos interconectados que trabajan juntos en equipo como un solo recurso integrado.

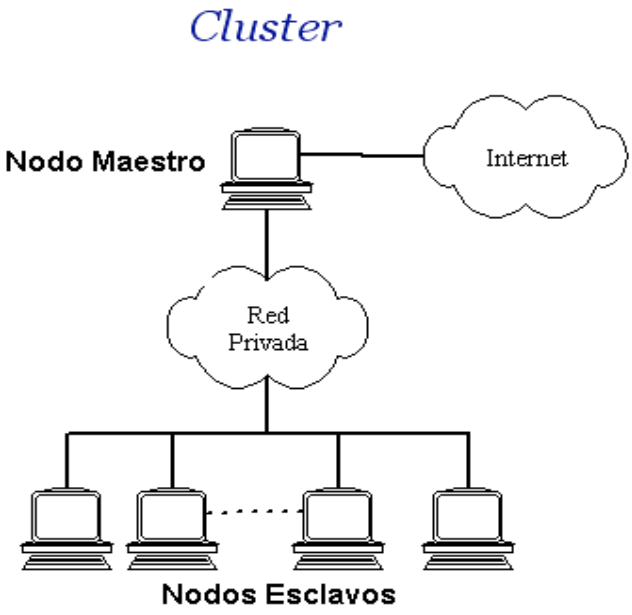

Figura 30 Ilustración de un Clúster

Clúster de Alto Rendimiento

El objetivo de un clúster de alto rendimiento es compartir el recurso más valioso de un computador, es decir, la capacidad de procesamiento. Este tipo de clústeres son muy populares, y quizás también los más antiguos, ya que en sus inicios fueron desarrollados para centros de cómputo enfocados en investigaciones científicas donde era necesario tener un gran poder de procesamiento, capaz de soportar una inmensa cantidad de cálculos para resolver un problema específico.

Python

Python es un lenguaje de programación sencillo y robusto que ofrece tanto la potencia como la complejidad de los lenguajes tradicionales compilados, junto con la facilidad de uso de lenguajes scripting e interpretados más sencillos. Se trata de un lenguaje multiparadigma, ya que soporta la orientación a objetos, la programación imperativa, programación funcional, entre otros. Además destaca entre sus características el tipado dinámico del lenguaje, lo que implica que una variable puede tomar valores de distintos tipos.

Parallel Python

Parallel Python (pp) es un módulo Python que proporciona el mecanismo para la ejecución en paralelo de código Python en SMP (Sistemas de Múltiples Procesadores) y Clúster (Sistema interconectados vía red). Es ligero, fácil para la instalación y brinda la posibilidad de integrarse con otros programas escritos en Python.

Actualmente el software escrito en Python se encuentra en una gama amplia de categorías incluyendo los negocios, el análisis de datos y el cálculo científico. Esto con la amplia disponibilidad de ordenadores SMP y la facilidad de implementar Clústeres en el mercado, crea la demanda para la ejecución en paralelo de código Python que permita optimizar tiempos y recursos.

Características:

- Ejecución para lela de código Python en SMP (Multiproceso Simétrico) y Clúster.
- Facilidad para entender y poner en práctica la técnica.
- Detecta automáticamente la configuración más óptima.
- Asignación de procesos dinámicos, el número de procesadores trabajando puede ser cambiado en tiempos de ejecución.
- Posee equilibrio óptimo de carga.
- Proporciona seguridad, ya q posee métodos para la autenticación basada en SHA para las conexiones en red.
- Fácil portabilidad para sistemas Windows, Unix y Linux.
- Es Open Source (código Abierto).

Guía de Instalación y Configuración

Requisitos Previos

Parallel Python entre sus características destaca el ser una librería multiplataforma, derivada del lenguaje con el que está escrita. Debido a esto

puede ser usado en equipos que funcionen con cualquier sistema operativo, siempre y cuando este tenga instalado el intérprete de Python en su versión 2+.

Descarga e Instalación de Parallel Python

Para la realización del proceso de instalación es necesario descargar desde la página oficial de Parallel Python [http://www.parallelpython.com/] un archivo que provee las funcionalidades de la librería. Este archivo viene en formato comprimido, una vez hagamos el proceso de descompresión tendremos un directorio con los archivos que se muestran en la Figura 31

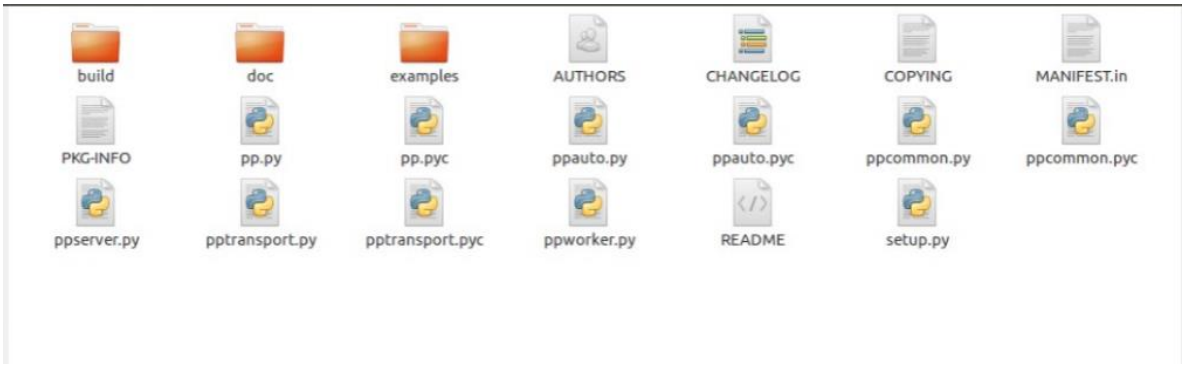

Figura 31 Archivos incluidos en directorio de pp

Ahora, para instalarlo debemos ejecutar el archivo python setup.py como se presenta en la Figura 32, este paso debe realizarse en todos los nodos que se pretenden involucrar en el clúster, inclusive en el que hará las veces de nodo maestro.

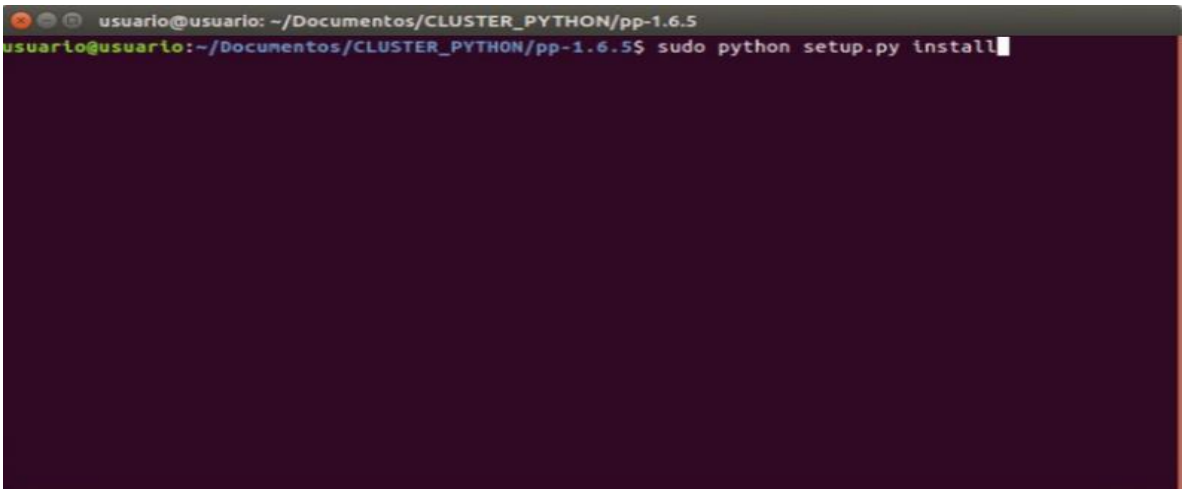

Figura 32 Comando de instalación de parallel python

Si la ejecución de este comando es satisfactoria, debería arrojarnos un resultado como el de la Figura 33

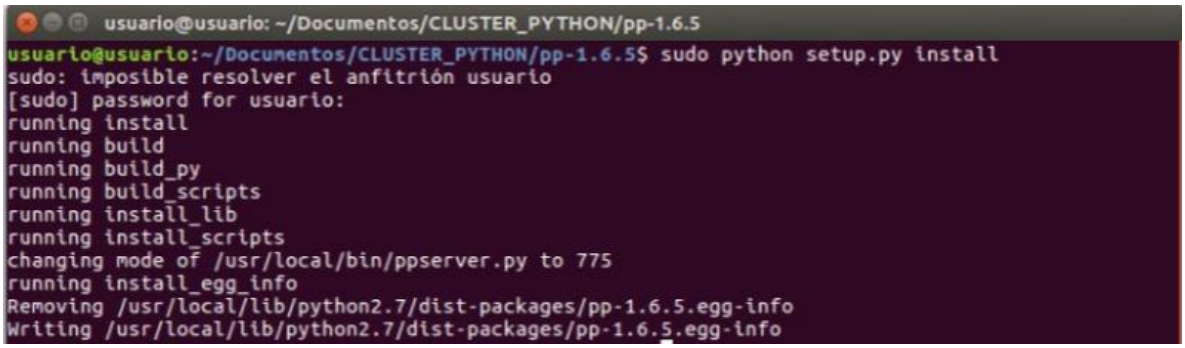

Figura 33 Resultado de ejecución de comando de instalación

Con la realización de estos pasos ya contamos con parallel python instalado en el nodo.

NOTA: Este comando se aplica con permiso de súper usuario por que escribe archivos de configuración en directorios protegidos por el sistema.

Preparación para el uso

Una vez que hemos culminado el proceso de descarga e instalación sin ningún inconveniente, ya dispondremos del módulo para realizar computación paralela, solo hay que tener en cuenta algunas consideraciones extra.

En los Nodos Esclavos

Cuando se han identificado los nodos que harán las veces de esclavos, en los cuales recaerá la carga del procesamiento, es necesario correr el servidor paralelo en los nodos que se van a usar

Nodo-1 >> python ppserver

Nodo-2 >> python ppserver

Nodo-3 >> python ppserver

Nodo-n >> python ppserver

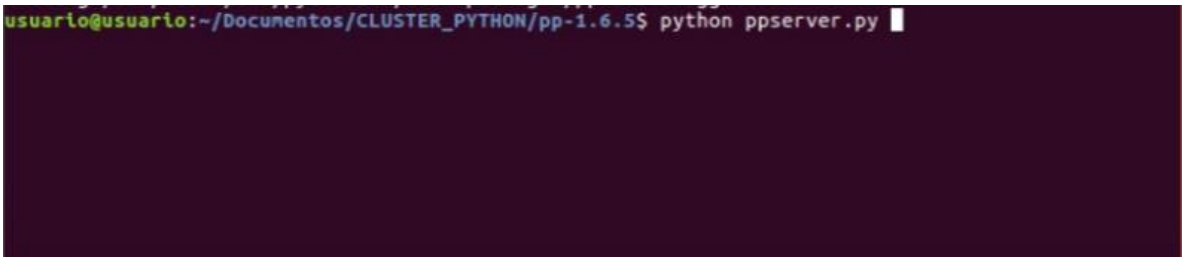

Figura 34 Ejecutar servidor de parallel python en un nodo

Parallel python proporciona la capacidad de auto detectar los nodos que están disponibles en el clúster, para activar esta opción se agregar el parámetro “–a” a la ejecución tal como se muestra en la Figura 35.

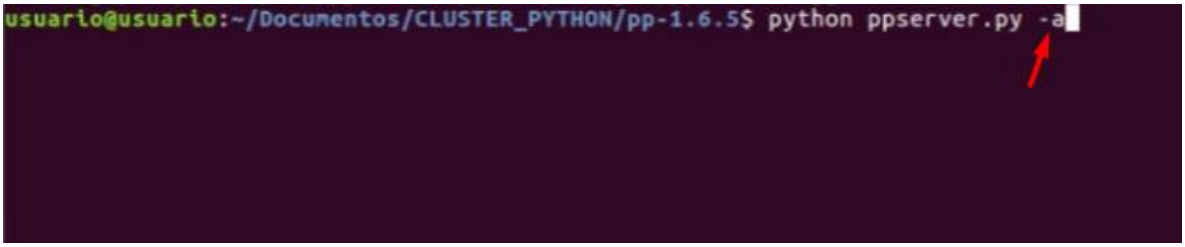

Figura 35 Ejecución de servidor con parámetro de autodescubrimiento

NOTA: Aunque el servidor se ejecute con el parámetro de autodescubrimiento, este no tendrá efecto si en el código a paralelizar no se especifica que se quiere hacer uso de dicha cualidad

En el Master

Teniendo todos los nodos a la escucha y espera de ser requeridos para tareas de procesamiento, solo falta un paso más para poder sacarle provecho al clúster.

En el nodo master donde se hará el lanzamiento de la aplicación paralela se debe modificar el código de dicha aplicación, se deben incluir las líneas de código que a continuación se describen:

1) **Importar el módulo de parallel python pp**

```
import pp
```

2) **Crear la lista con los nodos del clúster**

```
ppservers = ("ip-nodo1","ip-nodo2", "ip-nodox" )
```

Si se quiere hacer uso de la característica de autodescubrimiento la anterior linea debe modificarse quedando de la siguiente manera

```
ppservers = ("*",)
```

3) **Inicial la ejecución del servidos**

```
job_server = pp.Server(ppservers=ppservers)
```

4) **Presentar las tareas a ejecutar en paralelo**

```
f1 = job_server.submit(func1, args1, modules1)
```

Donde funnc1 es la función o tarea que se desea paralelizar, args1 son los argumentos que recibe como parámetro dicha función y modules1 son los módulos extra que se usan en la función paralelizada

5) **Recuperar los resultados si es necesario**

```
R1 = f1()
```

Solo es necesario recuperar en el caso de que la función que decidamos paralelizar retorne algún valor, de lo contrario se puede omitir este paso.

Ejemplo de Uso

En el directorio raíz de la librería se puede encontrar una carpeta llamada “examples”, en ella podemos encontrar varios programas que podemos usar para familiarizarnos con el uso del clúster y entender la forma en que este funciona. Teniendo instalada la librería en cada uno de los nodos, a continuación se muestra como ejemplo la ejecución del archivo auto_diff.py, el cual muestra la ejecución en paralelo de un programa que calcula la suma parcial de una función y su primera derivada mediante el método de diferenciación automática.

Para observar la diferencia de los tiempos de ejecución, primero o ejecutamos en un único nodo y observamos el resultado.

NOTA: La ejecución se realiza como cualquier otro programa de python

>>python auto_diff.py

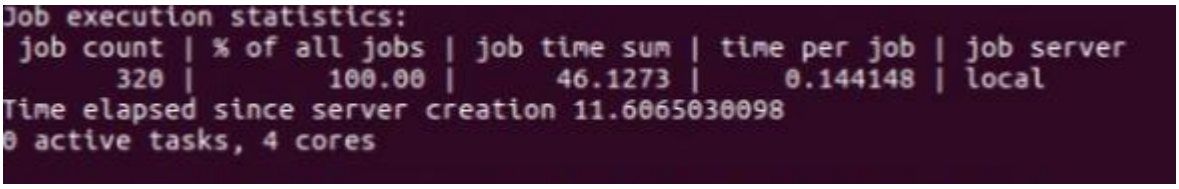

Figura 36 Estadísticas de ejecución en único nodo

En la figura 36 es posible observar que el tiempo que tardo en ejecutarse el programa fue de *11,60 segundos* aproximadamente.

Ahora volveremos a ejecutar el mismo programa, esta vez corriendo el servidor en dos nodos con el comando que se muestra en la Figura 34 y especificando en el programa cuales nodos se usaran por medio de su ip.

Figura 37 Estadísticas de ejecución para 2 nodos

En la Figura 37 puede notarse una disminución en el tiempo requerido para ejecutar el mismo programa, paso de *11,60* a *6,04 segundos* aproximadamente. Además de la notable mejora en los tiempos de ejecución en la imagen se destaca los nodos presentes en la ejecución, acompañando al nodo local también se encuentra e nodo identificado con la ip: 172.25.3.137

Anexo D. Tablas de tiempos por iteración.

Tabla 9 Tiempos por iteración para dif. Automática

Diferenciación Automática	Numero de Nodos									
	1	2	3	4	5	6	7	8	9	10
Tiempo (seg)	393,27	200,7	136	102	82	68,3	59,25	52,02	46,9	42,9
	393,51	200,4	133	102	82,38	68,4	59,2	52,25	47,1	43
	398,26	200,3	135	101	82,09	68,4	59,32	52,11	46,9	42,9
	391,92	201,1	134	101	81,94	69,3	59,08	51,88	47,2	42,9
	391,91	200,1	135	101	81,87	68,8	59,44	52,42	47	42,9
	395,92	200,9	134	102	81,57	69,1	59,48	52,21	47,1	43
	390,91	200,7	133	102	81,4	68,5	59,37	51,76	47,1	43
	392,56	199,3	133	101	81,26	68,6	59,21	51,92	47,2	43
	393,23	199,2	134	101	81,94	68,6	59,57	51,84	46,9	43,1
	392,34	199,7	133	101	81,93	68,5	59,07	51,7	47	42,7
	393,84	199,6	135	102	82,08	68,8	59,43	51,89	46,8	42,7
	390,35	199,1	133	102	81,69	68,4	59,33	51,7	47,2	43
	393,21	201,2	136	101	81,9	68,7	59,21	51,9	46,8	43
	396,76	198,9	135	101	82,05	68,4	59,5	51,79	47,4	42,8
	397,41	198,1	134	102	81,88	68,5	59,09	51,79	46,8	42,8
	393,57	201,1	134	101	82,1	68,4	59,15	52,12	46,6	43
	394,11	201,3	134	101	82,05	68,5	59,27	52,27	46,9	43,1
	389,76	199,2	134	101	82,23	68,2	59,12	51,9	47,3	43,1
	391,75	198,7	134	101	82,17	68,6	59,3	51,95	47,3	42,8
	395,58	200,2	134	101	81,78	68,5	59,22	51,89	46,7	43
	390,99	198,5	135	101	81,63	68,6	59,01	52,21	47,1	41,8
	392,16	199,3	135	100	81,88	68,6	58,98	52,18	47,1	42,9
	391,05	198,6	133	101	81,9	68,3	59,1	52,05	47	42,8
	395,77	199,3	135	102	82,05	68,9	59,15	52,17	47,3	43,1
	393,1	199	134	101	81,89	68,3	58,98	51,79	47	43
	396,85	199,9	133	102	81,51	68,9	59,12	51,74	46,8	43,1
	390,95	199,9	135	101	81,83	68,9	59,23	52,09	46,8	43
	393,99	199,6	133	102	81,86	68,7	59,28	52,21	47	43,1
	393,59	199,1	134	101	81,95	69	59,14	52,04	47,1	43,1
	391,49	200,3	133	101	82,23	68,7	59,23	51,97	47	42,8

Tabla 10 Tiempos por iteración para hash md5 invertido

Hash MD5	Numero de Nodos									
	1	2	3	4	5	6	7	8	9	10
Tiempo (seg)	22,02	12,28	8,31	6,54	5,44	4,73	4,1	4	3,64	3,26
	22,52	11,94	8,45	6,32	5,45	4,58	4,19	4	3,59	3,3
	23,15	11,65	8,27	6,5	5,42	4,58	4,09	3,97	3,64	3,24
	22,72	11,72	8,3	6,56	5,43	4,74	4,1	3,95	3,68	3,23
	22,77	11,69	8,28	6,6	5,47	4,65	4,17	3,89	3,42	3,29
	23,08	11,84	8,48	6,49	5,44	4,71	4,15	3,89	3,7	3,24
	23,1	12,21	8,27	6,55	5,49	4,59	4,15	3,85	3,44	3,32
	23,21	11,86	8,38	6,55	5,42	4,63	4,11	3,91	3,47	3,24
	22,58	12,21	8,3	6,58	5,44	4,7	4,1	3,9	3,7	3,25
	23,67	12,17	8,3	6,53	5,42	4,66	4,11	3,9	3,66	3,34
	22,64	11,76	8,37	6,58	5,4	4,72	4,15	3,89	3,45	3,23
	22,85	12	8,29	6,37	5,41	4,71	4,09	3,88	3,44	3,28
	22,88	11,91	8,27	6,59	5,55	4,7	4,11	3,9	3,63	3,24
	23,08	11,73	8,29	6,59	5,42	4,7	4,21	3,97	3,6	3,29
	23,28	11,72	8,23	6,59	5,41	4,64	4,14	3,97	3,47	3,31
	22,54	12,19	8,31	6,55	5,44	4,66	4,14	3,9	3,62	3,25
	22,81	11,8	8,37	6,55	5,48	4,68	4,07	3,89	3,59	3,23
	23,52	11,75	8,64	6,42	5,42	4,68	4,11	3,96	3,47	3,31
	22,02	11,91	8,29	6,43	5,44	4,71	4,19	4,01	3,43	3,29
	22,31	12,21	8,46	6,65	5,44	4,7	4,12	4,01	3,43	3,25
	22,41	12,2	8,27	6,76	5,44	4,66	4,11	3,91	3,49	3,3
	23,58	11,71	8,44	6,4	5,46	4,63	4,13	3,9	3,57	3,31
	22,77	11,75	8,52	6,59	5,39	4,65	4,15	3,89	3,75	3,31
	22,18	12,03	8,28	6,61	5,45	4,66	4,12	3,97	3,6	3,3
	23,01	11,76	8,46	6,56	5,43	4,83	4,13	3,83	3,49	3,3
	22,65	12,03	8,28	6,6	5,41	4,65	4,15	3,97	3,48	3,37
	22,9	11,8	8,47	6,45	5,48	4,72	4,11	3,86	3,46	3,22
	21,98	11,65	8,39	6,7	5,4	4,7	4,15	3,87	3,45	3,3
	23,07	12,15	8,35	6,57	5,42	4,78	4,23	3,88	3,44	3,24
	22,76	11,87	8,25	6,58	5,44	4,7	4,14	3,86	3,66	3,24

Tabla 11 Tiempo por iteración para suma de números primos

Suma de Primos	Numero Nodos									
	1	2	3	4	5	6	7	8	9	10
Tiempo (seg)	23,01	14,48	8,76	7,16	7,07	6,48	6,12	6,08	5,68	5,63
	23,27	14,44	11,86	7,27	7,36	6,5	6,05	6,03	5,59	5,67
	23,18	14,96	11,94	7,2	7,21	6,47	6,15	6,26	5,73	5,58
	23,21	15,59	8,79	7,35	7,17	7,06	6,31	6	6,06	5,59
	23,28	14,58	8,79	7,31	7,12	6,39	6,29	6,07	5,96	5,63
	23,12	14,59	8,74	7,17	7,18	6,39	6,18	5,68	5,76	5,92
	23,33	14,57	8,98	7,21	7,16	7,08	5,92	6,17	6,01	5,77
	23,39	15,01	9,42	7,21	7,19	6,85	6,12	6,09	5,64	5,61
	23,34	14,48	8,74	7,29	7,18	6,44	5,99	6,63	5,69	6,62
	23,16	14,64	8,74	7,3	7,17	6,87	6,14	6,04	5,75	5,57
	23,21	14,97	8,91	7,32	7,07	7,22	5,94	6,07	6,14	5,63
	23,05	15,92	8,917	7,2	7,41	6,9	6,16	6,08	5,73	5,71
	23,19	15,3	8,75	7,31	7,21	6,36	6,02	6,06	6,08	5,56
	23,19	14,66	8,78	7,26	7,14	6,43	6,11	5,99	5,77	5,53
	23,02	14,57	9,24	7,27	7,1	6,44	6,19	6,08	5,65	5,52
	23,08	14,59	8,78	7,25	7,33	6,98	6,15	6,06	5,65	6,04
	23,2	14,97	9,93	7,43	7,14	7,22	6,28	6,08	6	5,61
	23,06	14,9	8,9	7,31	7,2	6,97	6,13	6,05	5,6	5,59
	23,37	15,02	8,75	7,37	7,4	6,38	5,93	6,03	5,93	5,52
	23,34	14,92	8,77	7,04	7,07	6,93	6,18	6,01	5,66	5,61
	23,43	14,95	11,88	7,28	7,26	6,85	6,19	6,32	5,92	5,69
	23,2	15,05	8,73	7,38	7,18	6,57	6,2	6,06	5,98	5,54
	23,4	14,98	8,86	7,14	7,26	6,43	6,18	6,04	5,78	5,99
	23,18	14,5	8,74	7,21	7,21	6,43	6,1	5,97	5,76	5,6
	23,19	14,99	8,79	7,22	7,23	6,89	6,27	5,94	5,97	5,99
	23,17	14,96	9,86	7,19	7,15	6,5	6,17	6,04	6,09	5,56
	23,09	15,07	11,92	7,27	7,12	6,38	6,06	6,67	5,96	5,65
	23,17	14,96	8,77	7,44	7,25	7,23	6,15	6,14	6,03	5,54
	23,25	15,01	8,93	7,42	7,15	6,9	6,25	6,06	5,76	5,63
	23,18	14,51	8,73	7,21	7,03	6,48	6,08	6,04	5,65	5,6

Tabla 12 Tiempos por iteración para sumas parciales

Sumas Parciales	Numero de Nodos									
	1	2	3	4	5	6	7	8	9	10
Tiempo (seg)	33,06	17,05	11,48	8,76	7,16	6,82	5,67	5,52	5,26	5,56
	33,11	17,01	11,52	8,79	7,23	6,9	6,06	5,72	5,88	5,48
	33,1	16,98	11,53	8,73	7,26	6,79	6,66	6,02	5,47	4,74
	33,06	16,93	11,5	8,76	7,19	6,42	5,86	5,83	5,18	6,13
	33,12	16,89	11,53	8,77	7,12	6,62	5,99	5,77	5,51	4,98
	33,08	16,99	11,51	8,8	7,07	6,94	5,89	6,02	6,01	5
	33,09	16,99	11,44	8,75	7,21	6,73	6,64	5,56	5,63	5,36
	33,04	16,97	11,48	8,73	7,15	6,52	5,87	5,19	5,82	5,11
	33,15	16,99	11,53	8,78	7,13	6,39	5,93	6,23	6,18	5,07
	33,01	17	11,49	8,7	7,18	6,52	6,56	6,63	5,56	5,83
	33,09	16,97	11,5	8,75	7,15	6,8	6,16	6,12	6,68	5,52
	33,15	17	11,49	8,81	7,18	6,39	6,16	5,97	5,09	5,19
	33,1	17,04	11,56	8,73	7,17	6,69	5,55	6,38	6,01	4,86
	33,04	17,03	11,5	8,78	7,15	6,7	5,92	5,38	5,7	5,17
	33,11	17,01	11,6	8,78	7,17	6,82	5,92	6,14	6,05	5,74
	33,01	16,95	11,51	8,77	7,22	6,79	6,54	5,78	6	5,21
	33,05	17	11,46	8,73	7,23	6,71	6,36	5,81	5,83	5,32
	33,1	16,99	11,49	8,74	7,15	6,62	6,56	5,61	5,83	5,04
	33,12	16,99	11,5	8,76	7,2	6,66	6,26	6	5,89	6,43
	33,15	16,98	11,48	8,76	7,16	6,64	6,42	5,97	5,8	5,45
	33,18	17	11,56	8,82	7,17	6,5	5,97	6,14	6,27	5,81
	33,16	16,95	11,56	8,75	7,2	6,42	6,49	6,48	6,44	5,45
	33,16	16,99	11,47	8,72	7,18	6,36	6,44	5,71	5,74	5,29
	33,07	16,93	11,5	8,73	7,15	6,56	6,38	6,22	5,71	5,75
	33,04	16,96	11,46	8,76	7,13	6,49	7,01	5,75	5,26	5,75
	33,17	16,94	11,48	8,76	7,24	6,54	5,61	5,82	5,42	5,5
	33,08	17	11,45	8,75	7,15	6,6	6,16	6,31	6,49	6,33
	33,19	16,96	11,53	8,76	7,22	6,65	6,54	6,2	5,49	6,18
	33,16	17,06	11,5	8,71	7,19	6,89	6,03	6,17	6,01	6,12
	33,11	16,93	11,49	8,74	7,33	6,87	6,23	6,03	5,59	4,94

Tabla 13 Tiempos por iteración para ordenamiento por quicksort

Quicksort	Numero de Nodos									
	1	2	3	4	5	6	7	8	9	10
Tiempo (seg)	21,42	21,29	21,31	23,34	23,1	26,08	23,9	25,6	20,7	25,27
	20,81	20,18	20,43	19,16	28,3	20,44	30,3	27,84	24,3	26,27
	30,99	23,58	30,33	27,76	19,2	19,12	21,3	25,37	22,3	22,53
	33,28	35,48	19,87	19,57	19	22,2	22,5	24,36	24,6	29,39
	21,71	22,29	21,61	22,35	28,9	22,23	25,4	27,29	25,8	21,06
	27,79	23,77	24,11	22,14	28,7	22,24	20,5	21,65	21,4	23,54
	29,14	23,59	22,1	29,03	30	22,36	22,7	27,53	20,6	26,91
	25,42	20,63	24,89	21,6	22,9	22,78	22,9	21,59	22,8	22,5
	27,58	26,41	21,24	20,95	23,1	20,11	31,7	21,95	24,6	21,07
	21,46	25,98	24,9	25,27	20,8	34,38	22,8	25,65	23,9	29,96
	21,38	21,73	25,47	24,39	29,2	29,05	24,3	21,02	27,6	20,53
	22,11	26,45	29,44	21,04	19	28,38	21,9	26,52	24,6	25,5
	24,6	23,24	22,24	29,52	22,8	36,47	27,1	24,39	21,8	23,44
	22,57	21,8	26,2	22,74	23,8	23,39	22,6	23,25	29,6	23,01
	23,19	22,63	22,56	28,42	19,1	21,33	23,7	24,04	24,7	35,85
	23,88	21,67	29,75	20,78	23,6	21,28	21,7	21,76	25,5	26,22
	22,89	25,03	23,36	24,79	30,2	29,62	22,8	21,07	25,9	21,83
	21,61	19,98	29,24	19,68	29,5	25,89	24,2	28,99	21,5	21,58
	23,77	24,68	21,94	20,83	20,8	31,34	21,3	25,11	22,4	22,32
	20,92	20,18	22,39	22,96	22,1	22,43	23,8	18,54	26	31,27
	22,8	27,52	22,27	22,14	22,8	27,89	19,5	23,55	32,6	24,17
	21,27	21,45	20,06	29,73	21,6	21,61	23,9	19,89	22,9	24,52
	28,01	30,25	20,94	20,38	20,8	22,67	22,7	32,81	24,1	24,95
	26,98	25,46	20,29	20,67	30,1	24,48	23,9	24,99	30,9	24,58
	25,42	19,37	21,31	21,13	19,7	22,78	22,3	27,19	20,5	28,76
	21,59	22,22	20,88	19,93	20,3	27,45	46,6	23,92	20,1	28,68
	22,15	27,26	22,24	23,49	31,1	32,89	35,7	25,6	22,9	20,61
	21,52	22,32	20,49	27,84	19,9	23,9	22,6	27,84	18,5	21,24
	23,01	25,19	21,41	24,31	18,7	21,77	25,7	21,13	21,7	21,54
	20,63	25,55	30,87	19,96	24	19,66	21,9	25,62	29,1	26,37

Tabla 14 Tiempos por iteración para K-Means (I)

Kmeans CICOM	Numero de Nodos							
	1	2	3	4	5	6	7	8
Tiempo (seg)	153,22	1444,04	1263,45	996,57	788,57	704,23	651,1	611,09
	154,98	1533,65	1203,32	991,83	788,02	702,16	636,21	617,28
	150,76	1814,89	1322,05	1004,34	794,26	699,31	653,12	602,3
	154,45	1543,79	1350,65	1017,42	790,14	701,43	651,71	608,72
	156,13	1728,14	1166,49	985,91	793,75	703,2	648,43	613,66
	153,21	1752	1239,73	993,5	812,24	700,99	642,87	613,89
	155	1537,5	1279,58	998,9	805,87	698,16	661,01	613,17
	153,73	1723,54	1311,71	1007,45	816,42	707,15	653,69	611,38
	153,63	1478,37	1269,14	1001,01	795,1	697,36	654,93	606,7
	153,62	1548,55	1235,81	1000,99	789,12	702,15	638,74	612,96
	153,33	1720,31	1194,59	992,63	790,23	695,73	666,28	609,05
	154,56	1843,54	1246,76	991,87	794,98	698,1	651,95	611,36
	153,6	1526,73	1314,01	995,7	800,08	706,87	643,22	607,27
	153,93	1733,1	1225,28	994,94	798,59	699,76	650,22	604,08
	153,72	1720,13	1189,43	997,6	792,43	699,91	648,74	606,74
	153,65	1640,19	1267,05	988,19	791,51	698,19	652,31	610,07
	153,61	1567,37	1233,1	991,28	789,28	693,26	652,79	607,51
	153,39	1587,91	1194,34	997,31	788,98	695,38	648,38	613,36
	153,81	1608,56	1287,78	992,65	786,74	698,22	653,26	614,93
	153,94	1528,72	1318,2	992,81	796,48	705,77	647,41	617,1
	153,39	1680,1	1205,68	991,21	793,32	729,12	654,59	609,8
	153,61	1765,21	1222,51	993,75	790,89	701,13	636,29	613,65
	153,97	1489,72	1282,32	990,09	791,87	704,21	653,35	613,72
	153,7	1557,76	1242,9	992,5	799,89	705,32	645,78	602,15
	153,88	1538,05	1264,78	998,52	813,23	697,89	643,4	600,99
	153,57	1527,51	1330,15	992,76	804,79	699,42	632,6	601,59
	153,77	1498,97	1219,4	1021,78	796,21	695,34	625,89	611,23
	153,77	1767,68	1254,58	997,65	798,7	696,8	642,4	598,15
	153,81	1770,65	1232,3	992,89	791,11	702,13	657,1	610,9
	155,21	1545,2	1228,57	985,7	789,23	692,76	651,2	609,32

Tabla 15 Tiempos por iteración para K-means II

Kmeans II	Numero de Nodos							
	1	2	3	4	5	6	7	8
Tiempo (seg)	153,22	137,05	97,03	83,2	68,05	64,82	56,39	56,38
	154,98	137,78	100,29	76,86	70,01	63,2	57,52	54,76
	150,76	135,4	96,19	79,28	69,39	62,03	56,34	54,46
	154,45	141,33	98,81	77,09	67,96	61,38	57,82	54,58
	156,13	136,15	99,05	80,67	68,22	62,67	56,37	54,43
	153,21	138,94	101,65	79,54	68,77	62,49	56,73	55,09
	155	139,54	98,33	78.09	67,99	63,15	56,9	54,28
	153,73	137,5	103,24	80,48	69,41	61,24	57,91	53,8
	153,63	137,91	100,13	76,36	69,88	62,7	57,06	54,69
	153,62	140,37	99,72	77,13	68,42	63,01	57,47	57,87
	153,33	135,24	97,47	76,31	69,91	62,52	58,03	54,37
	154,56	134,06	96,3	75,96	67,33	63,21	57,56	53,85
	153,6	136,38	93,67	78,72	70,14	63,73	56,84	54,94
	153,93	137,71	96,46	78,34	67,7	62,02	57,7	54
	153,72	142,84	94,19	79,47	68,39	62,13	58,01	54,32
	153,65	135,6	94,53	76,23	68,81	62,48	56,73	54,7
	153,61	140,55	97,01	78,76	68,9	62,21	56,47	54,61
	153,39	138,72	94,17	76,77	68,03	63,07	56,98	54,78
	153,81	137,9	97,59	77,04	68,45	62,8	56,43	56,04
	153,94	137,37	97,23	77,42	68,33	62,79	56,1	53,94
	153,39	139,17	94,62	79,43	68,18	63,1	58,14	54,62
	153,61	136,87	93,74	79,17	68,6	62,87	57,72	54,25
	153,97	137,22	96,23	77,19	69,19	62,56	56,27	54,61
	153,7	136,82	94,1	78,63	68,59	62,54	56,07	54,68
	153,88	140,55	96,45	79,62	68,26	62,15	57,35	54,7
	153,57	139,75	97,67	78,66	68,69	61,67	59,92	54,5
	153,77	137,47	94,15	79,82	68,63	61,66	57,14	53,89
	153,77	140,25	99,2	77,26	68,07	63,18	56,19	54,99
	153,81	136,78	115,61	79,17	68,98	62,34	56,79	53,82
	155,21	138,23	113,34	76,77	69,4	62,83	56,27	54,62