Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

ESTUDIO EXPERIMENTAL DE LA COMPLEJIDAD COMPUTACIONAL PARA LA PRODUCCIÓN DE UNA DISTRIBUCIÓN LINUX BASADA EN GENTOO CON CLUSTERS DE COMPILACIÓN ICECREAM Y DISTCC

Víctor Leonel Orozco López

Asesorado por el Ing. José Francisco Lobos

Guatemala, junio de 2011

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

ESTUDIO EXPERIMENTAL DE LA COMPLEJIDAD COMPUTACIONAL PARA LA PRODUCCIÓN DE UNA DISTRIBUCIÓN LINUX BASADA EN GENTOO CON CLUSTERS DE COMPILACIÓN ICECREAM Y DISTCC

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

VÍCTOR LEONEL OROZCO LÓPEZ
ASESORADO POR EL ING. JOSÉ FRANCISCO LOBOS

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, JUNIO DE 2011

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Murphy Olympo Paiz Recinos
VOCAL I	Ing. Alfredo Enrique Beber Aceituno
VOCAL II	Ing. Pedro Antonio Aguilar Polanco
VOCAL III	Ing. Miguel Ángel Dávila Calderón
VOCAL IV	Br. Juan Carlos Molina Jiménez
VOCAL V	Br. Mario Maldonado Muralles
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Murphy Olympo Paiz Recinos
EXAMINADOR	Ing. Oscar Alejandro Paz Campos.
EXAMINADOR	Ing. Cesar Augusto Fernández Caceres
EXAMINADOR	Ing. Pedro Pablo Hernandez
SECRETARIA	Inga. Marcia Ivónne Véliz Vargas

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

ESTUDIO EXPERIMENTAL DE LA COMPLEJIDAD COMPUTACIONAL PARA LA PRODUCCIÓN DE UNA DISTRIBUCIÓN LINUX BASADA EN GENTOO CON CLUSTERS DE COMPILACIÓN ICECREAM Y DISTCC

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas, con fecha julio de 2010.

Víctor Leonel Orozco López

ACTO QUE DEDICO A:

Dios	En medio de las particularidades de la naturaleza humana, nos proporciona la sabiduría necesaria para crear nuestras propias soluciones con base en el ingenio, la pasión y el trabajo
Leonel Orozco	Por sus constantes reprimendas respecto a la importancia de la disciplina, por sus anécdotas respecto a la vida universitaria y por transmitirme el amor hacia la tricentenaria Universidad de San Carlos.
Liliam López	El soporte sentimental durante todo este trayecto universitario, sus consejos en momentos de desesperación por fin dieron sus frutos.
José Francisco Lobos	Sin su ayuda y su guía este trabajo de graduación jamás hubiera podido ser finalizado.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

TABLAS

GLOSARIO

Backend Aplicación destinada a realizar tareas en segundo plano, para otra aplicación

Balanceador Aplicación destinada a distribuir la carga de trabajo entre todos los nodos que pertenecen al *cluster*

Cluster Reunión de computadoras para la creación de sistemas de alta disponibilidad, alta eficiencia y alto desempeño en procesamiento de información

Código fuente Instrucciones que describen el funcionamiento de una tarea determinada para que sea ejecutada por una computadora

Compilación Proceso mediante el cual se traduce de un archivo de código fuente con instrucciones entendibles por un programador hacia un archivo ejecutable con instrucciones entendibles por una computadora

Complejidad Nivel de dificultad que presenta una tarea para ser finalizada con éxito

Data Mining Proceso de búsqueda y reconocimiento de patrones y relaciones entre un conjunto de datos

Demonio Cron Aplicación que se ejecuta en segundo plano en sistemas tipo Unix, destinada a la ejecución de otras tareas en periodos fijos de tiempo

Distribución Linux Conjunto de aplicaciones, paquetes y bibliotecas de funciones configuradas para formar un sistema operativo basado en el *kernel* Linux

Ensamblador Lenguaje de bajo nivel utilizado para escribir programas informáticos

Ethernet Sistema estandarizado de conexión de computadoras para redes de área local

GNU Acrónimo recursivo de *GNU Not Unix.* Fue el nombre que Richard Stallman ideo para el proyecto que pretendía crear un sistema operativo totalmente libre inspirado en Unix.

GNU/Linux Nombre formal para los sistemas operativos basados en el *kernel* Linux y el conjunto de utilidades de sistema, compiladores, y bibliotecas de funciones del proyecto GNU

Grid *Grid* es un sistema conformado de un número indeterminado de computadoras separadas por grandes distancias interconectadas generalmente a través de Internet

Infiniband Sistema estandarizado de conexión de computadoras para redes de área local con alto rendimiento y velocidad

Kernel	Programa encargado para servir de puente entre las aplicaciones de usuario y el *hardware* propio de cada computadora, es la base fundamental de cualquier sistema operativo
Linux	Nombre del *kernel* creado por Linus Torvalds en 1991 como un reemplazo al *kernel* del sistema *Minix*
Memoria Ram	Módulos de memoria que sirven para almacenamiento temporal de información en una computadora, mientras viaja entre procesador, *hardware* y dispositivos de almacenamiento
Mirinet	Sistema estandarizado de conexión de computadoras para redes de área local con alto rendimiento y velocidad
Modelo TAM	Modelo de aceptación de tecnología que pretende modelar cómo un usuario acepta o rechaza una tecnología
MPI	Interfaz de Paso de Mensajes, es una especificación para la creación de interfaces de programación en sistemas distribuidos
Nodo	Un computador miembro de un sistema más grande
P2P	*Peer to peer*. Término genérico para describir redes cuyo funcionamiento es descentralizado y las interconexiones entre miembros de la red se realizan de punto a punto

Paquete Conjunto formado por una aplicación, sus especificaciones y sus métodos de instalación, muy comunes en sistemas Unix para distribución de *software*.

Programa Ejecutable Es un programa conformado por instrucciones en lenguaje máquina listas para su ejecución

PVM Biblioteca de funciones especialmente diseñada para creación de programas en sistemas distribuidos.

Script Programa desarrollado en un lenguaje de programación interpretado que generalmente se utiliza para realizar tareas repetitivas sin necesidad de ser compilado.

***Software* Libre** Paquete de *software* que cumple con cuatro principios o "libertades básicas". 1) Uso bajo cualquier propósito, 2) Posibilidad de estudio del funcionamiento de la aplicación, 3) Libertad de distribución de copias del programa sin limitante, 4) Libertad de aplicar mejoras a la aplicación y de la libre distribución de estas mejoras.

RESUMEN

En el ámbito nacional el campo de ciencias de la computación está a muchos años de distancia de otras sociedades más avanzadas, un campo particularmente activo en años recientes ha sido el desarrollo de aplicaciones bajo metodologías de código abierto y los sistemas GNU/Linux. Actualmente se dan las primeras etapas análogas a los años noventa en Estados Unidos y Europa, cuando Linus Torvalds presentaba la versión 0.1 del *kernel* Linux y en *Sillicon Valley* se formaban los primeros grupos de usuarios de Linux.

Los sistemas GNU/Linux son desarrollados en un ambiente colaborativo, donde el resultado final es un conjunto de *software* desarrollado por distintos proyectos y distintos programadores, en distintas partes del mundo. El siguiente paso para la comunidad de *Software* Libre en Guatemala es pasar de la promoción y difusión del uso *Software* Libre, hacia el desarrollo activo de más programas para aumentar la funcionalidad. Para lo cual, además de tiempo y capital humano, se necesita capacidad de cómputo a la hora de producir programas ejecutables.

Este estudio se ha elaborado comparando dos herramientas para elaboración de *clusters* de compilación distribuida, utilizando como instrumentos de comparación la complejidad computacional, basados en el modelo de complejidad computacional de la Universidad de York (Canadá).

La medición de datos significativos para la elaboración del estudio se realizó en un *cluster* experimental de tres nodos, ejecutando la distribución Gentoo GNU/Linux y fueron analizados con el *software* estadístico *SPSS*. La conclusión fue que sí existe diferencia de rendimiento significativo entre el *software* Distcc, *Icecream* y Distcc con optimizaciones.

OBJETIVOS

General

Elaborar un estudio para explicar la complejidad computacional a la hora de producir distribuciones Linux, específicamente, en la compilación de paquetes con *clusters*, HTC para compilación *Icecream* y Distcc.

Específicos

1. Elaborar un estudio de complejidad sobre la producción de una distribución basada en Gentoo GNU/Linux con ambas herramientas para la creación de *clusters.*

2. Identificar las fortalezas y debilidades respecto al rendimiento para un *cluster Icecream* y un *cluster* Distcc, respectivamente

3. Evaluar las distintas optimizaciones que son comunes para ambos tipos de *cluster*, como Ccache

4. Confrontar los resultados para determinar si un proyecto comunitario como Distcc es superior o inferior a un proyecto patrocinado como *Icecream* (patrocinado por Novell).

5. Establecer criterios de elección para elegir una herramienta u otra dependiendo de las características de los lenguajes a generar.

INTRODUCCIÓN

Si hacemos una breve reflexión sobre la humanidad y su desarrollo científico nos encontramos con el hecho de que el ser humano, desde tiempos históricos, ha tenido la necesidad de auxiliarse de máquinas y herramientas para realizar una tarea importante para el desarrollo “el cálculo y procesamiento de datos”. La computadora, como tal, fue concebida originalmente como una máquina de cálculo avanzada. Aunque, sus funciones y campos de acción se han diversificado, todas sus funciones se hacen con el tratamiento y transformación de información. Desde convertir los bits almacenados en archivos .mp3 a ondas sonoras, hasta análisis de datos y presentación de informes. El cálculo y procesamiento han sido las capacidades a mejorar siempre tanto a nivel de *software* como *hardware*. Con la particularidad de que el *software* es el que siempre obliga al *hardware* a mejorar. El mayor reto a nivel de *hardware* es lograr más capacidad de cálculo con la menor inversión posible.

En los primeros días de la computación, el escalado se podía hacer de una sola forma: agregando más *hardware* hasta llevar a la computadora al límite. Esto se conoce como “escalado vertical”. Sin embargo, con la creación de las primeras redes de computadoras, UNIX y más recientemente la arquitectura PVM nacen los verdaderos sistemas distribuidos. Abriendo la posibilidad de que los recursos computacionales utilizados para grandes procesamientos ya no dependan de una única computadora. Ahora pueden utilizarse varias estaciones de trabajo para realizar las mismas tareas.

Esta reunión de computadoras se conoce como *clusters.* Palabra anglosajona que no tiene traducción literal al Español pero puede interpretarse como reunión de recursos. La ciencia de la supercomputación no es nada nueva y actualmente las supercomputadoras se elaboran con base en computadoras más pequeñas que colaborarán en realizar una o varias tareas. Tareas tan diversas, desde simular el planeta tierra, hasta analizar imágenes para la búsqueda de vida extraterrestre.

Dependiendo de la tarea que se realiza, los *clusters* se pueden clasificar en tres funciones: para garantizar la continuidad de un servicio o *clusters* de alta disponibilidad (*HA* por sus siglas en inglés). Para procesar la mayor cantidad de datos dedicando todos los recursos a una sola tarea o *clusters* de alto rendimiento (*HPC* por sus siglas en inglés). Y, por último, pero no menos importante, *clusters* dedicados a realizar la mayor cantidad de tareas en el menor tiempo posible o *clusters* de alta eficiencia (*HTC* por sus siglas en inglés). Donde la plataforma por excelencia es Linux como el sucesor natural de UNIX. Y es que, aunque existan otras opciones ya probadas, basadas en otros UNIX comerciales como Solaris o Mac OS e incluso algunas soluciones basadas en Windows, Linux es un difícil competidor, por su rendimiento, sus herramientas de administración y sus costos.

Linux es un sistema en evolución permanente y altamente inestable respecto a sus lanzamientos. Todos los componentes que conforman un sistema Linux tienen ciclos de lanzamiento muy diferentes y actualizaciones muy constantes. Esto provoca que las grandes compañías y comunidades que dan soluciones Linux, congelen el estado de muchos de sus componentes y generen distribuciones con cierto grado de estabilidad. Sin embargo, el desarrollo nunca se detiene.

Esto lleva a que las comunidades o compañías generen proyectos comunitarios (como el caso Fedora-Red Hat) o generen proyectos inestables (como en las ramas de Debian *testing* y *stable*). Proyectos que reciben actualizaciones muy rápidas y necesitan procesar alta cantidad de información para generar los paquetes que integrarán la distribución.

Este estudio se elaboró con la finalidad de comparar dos herramientas para creación de *clusters* de alta eficiencia y que generen paquetes binarios a partir de código fuente. *Icecream*, originalmente fue creada por OpenSuse, proyecto patrocinado por Novell. Y Distcc fue original del proyecto Samba. El estudio y las métricas se analizan con base en las teorías de la complejidad computacional y minería de datos. Con el único objetivo de ser una guía para quien que esté interesado en utilizar alguna de estas dos herramientas y que pueda conocer sus fortalezas y debilidades. Y que pueda escoger cuál utilizar, a la hora de trabajar en su proyecto.

1 . COMPLEJIDAD COMPUTACIONAL

1.1 . ¿Qué es la complejidad computacional y cómo se mide?

El ser humano se visualiza como una de las máquinas más perfectas existentes, capaz de sentir, inventar y crear. Sin embargo, tiene entre los limitantes de su naturaleza algunas características que hacen necesario el auxiliarse de máquinas. El simple hecho de memorizar cantidades complejas o realizar cálculos con números de gran cantidad de dígitos, hace evidente que el humano y su flexibilidad son débiles a la hora de ejecutar procedimientos bien establecidos.

Este patrón se repite no sólo en el ser humano, cuando una aplicación se hace más compleja o más especializada. Incluso a nuestros recursos de cómputo se les hace difícil procesar la información.

1.2 . Complejidad de los lenguajes de programación

El caso de los lenguajes de programación es tal vez uno de los ejemplos más claros acerca del uso y consumo de recursos. Cada día hay lenguajes de programación, donde aumentan sus características, dejando menos responsabilidad al usuario y, por consiguiente, los compiladores son más especializados pero requieren más tiempo para traducir el código.

Basta con analizar la evolución desde el lenguaje ensamblador hacia una de las plataformas más extendidas a nivel mundial: C++. Los primeros compiladores de lenguajes de programación como ensamblador se limitaban a traducir instrucciones expresadas en estructura de tres direcciones a código máquina, haciendo una optimización de código para procesos tan simples como código inalcanzable o variables sin utilizar. Era responsabilidad del programador expresar de manera explícita lo que la computadora debía de hacer con sus dispositivos y con el procesador. Esto ha cambiado totalmente con los lenguajes de alto nivel. Donde el objetivo es que el programador se concentre en lo que su aplicación debe de hacer sin preocuparse de cómo la computadora llevará a cabo las tareas.

1.3 . Complejidad en la producción de archivos ejecutables

La responsabilidad del programador cada vez se reduce más con los lenguajes de nueva generación. Actualmente, el programador ya no debe de preocuparse de cosas como: la cantidad de memoria que utilizará la aplicación, bloques y segmentos donde ubicar el programa, programación de ciclos con base en sentencias *GOTO*. Esto se repite en cada innovación de los lenguajes de programación. Cada salto en los lenguajes de programación genera mejores herramientas, y ayuda a que el programador se preocupe menos por los detalles técnicos y más por la lógica que debe realizar el programa. Su efecto es el contrario, en lo que al compilador respecta, cargándolo con más tareas. Ahora el compilador también se preocupa por las cosas que deben de suceder en tiempo de ejecución, traducción de las ideas que el programador plasmó en un lenguaje de alto nivel.

La compilación en particular es una tarea bastante compleja. Además de traducir el código, el mismo se tiene que optimizar, enlazar a componentes externos, conformar un único ejecutable con base en muchos archivos y algunos otros procesos que hacen que su transformación sea cada vez más difícil. Prueba de ello es que desde los años ochenta, hasta la fecha sólo han habido cuatro versiones mayores del *GNU C compiler*[1] (renombrado posteriormente a *GNU compiler collection*), el compilador por excelencia para Linux y para los UNIX en general.

La complejidad computacional es una de las áreas más extensas de las ciencias computacionales, su campo de acción trata de cubrir los problemas relacionados con la dificultad de diseñar algoritmos para resolver problemas y también los recursos necesarios para ejecutar estos algoritmos. Aunque GCC es *software* libre y cualquiera puede ver el código y los algoritmos que lo componen. Seria muy complicado analizar el impacto que tienen las herramientas de compilación distribuida sobre los algoritmos de GCC y su estructura.

Si pensamos en GCC como uno de los primeros frutos del proyecto GNU, el cual busca un sistema totalmente libre basado en UNIX, estamos frente a un producto con bastante código heredado desde los años ochenta. Y uno de los proyectos más grandes del *software* libre en general. Por lo cual, los algoritmos que lo componen son complejos, bien desarrollados y diseñados por muchos expertos en el tema. Habilidades que no tiene el administrador de sistemas promedio.

1 GCC Timeline - http://gcc.gnu.org/releases.html

1.4 . La complejidad computacional vista desde afuera

Ésta no es ni la primera ni la última situación que es difícil estudiar desde el centro, de hecho, muchos fenómenos primero se estudian por sus resultados para entender la relación de sus componentes. Esta búsqueda y reconocimiento de relaciones se conoce como *Data Mining*[2], herramienta de análisis para este estudio.

La Universidad de York define como variables que influyen en los recursos de la complejidad computacional: tiempo, memoria utilizada, número de procesos que se ejecutan en paralelo y porcentaje de utilización del procesador[3]. Variables que existen y pueden ser medidas a la hora de producir un paquete para integrar una distribución Linux. Estas variables pueden ayudarnos a descubrir patrones con base en algunos otros factores independientes; por ejemplo: el tamaño de los datos a procesar, la herramienta con que se configuró el paquete para su complicación y el lenguaje en que está escrito. Permitiéndonos ver si estos factores independientes influyen en el desempeño de *Icecream* y Distcc; herramientas con las que se producen paquetes en un *cluster.*

2 Data Mining - http://www.businessdictionary.com/definition/data-mining.html

3 York University - http://www.fsc.yorku.ca/york/istheory/wiki/index.php/Complexity_theory

2 . CLUSTERS

2.1 . Origen de los *clusters*

Es interesante evaluar la creación de lo que conocemos como la computación moderna y su naturaleza evolutiva. La computadora en sus primeros días ni siquiera fue concebida para las funciones que hoy realiza. Su objetivo principal era realizar cálculos que para el ser humano eran complicados o procesar y almacenar información para su procesamiento. Ejemplos como la *bombe de Turing* para romper códigos en la Segunda Guerra Mundial confirman esta idea.

Lo que conocemos como computación moderna fue una era marcada con el reemplazo de los tubos de rayos catódicos por silicio. Lo cual hizo más factible fabricar computadoras a menos costo y con mayor capacidad. Provocado por la minimización y eficiencia de los componentes. A su vez se provocaron máquinas con mayor capacidad, naciendo con “Leyes” como la “Ley de Moore”[4] afirmación empírica, que nos dice que los microprocesadores duplicarán su capacidad cada 18 meses y que, hasta el momento, se ha cumplido de una manera aceptable. Este modelo; sin embargo, presenta deficiencias en el aspecto de la inversión.

4 Ley de Moore - http://www.intel.com/technology/mooreslaw/

Algunas de las limitantes más importantes de este modelo son:

- El agregar capacidades a una computadora siempre se ve limitado por su diseño original, no importa si es una computadora de bolsillo o de grandes prestaciones.

- La inversión en una computadora de grandes prestaciones no suele ser factible porque son millones de dólares invertidos en un equipo que se devaluará rápidamente.

- Aunque se tenga el dinero algunas tareas son difíciles de realizar al no existir computadoras tan potentes como sus necesidades.

Estas dificultades provocaron lo que parecía la idea más obvia, dividiendo el trabajo entre varias computadoras, las cuales acelerarían el procesamiento de datos o crearían sistemas con más capacidad. Este hecho no era aislado ni tampoco desconocido. En 1967 se presenta uno de los primeros estudios acerca de *clusters,* elaborado por Gene Amdahl[5] para IBM, artículo que se conocería como la Ley de Amdah. En este artículo, Amdahl elabora un estudio donde describe matemáticamente el proceso de aumentar la velocidad al paralelizar las operaciones. Aunque la era de la supercomputación moderna necesitaba de algunas otras creaciones para ser técnicamente posible, este artículo define algunas de las bases de la computación y los *clusters* de hoy en día. Complementándose por los medios que ayudarían a formar esta unión de computadoras.

5 Ley de Amdah http://www.redbarnhpc.com/v2/index.php/cluster-computing/the-history-of-cluster-computing

Entre estos medios se encuentran: el protocolo de red TCP/IP como parte del Centro de Investigación de Palo Alto (PARC por sus siglas en inglés), la invención de UNIX, por parte de Ken Thompson, Dennis Ritchie y Douglas McIlroy como uno de los primeros sistemas operativos orientados y diseñados para ambientes de red; y más recientemente la creación de la arquitectura de programación PVM, para escritura de programas paralelizables.

2.2 . Clasificación de los *clusters*

Si partimos del principio de que la computadora es una máquina de cálculo y procesamiento de información, podemos pensar que un *cluster* se dedica a lo mismo. En efecto, toda la computación, de una u otra forma, se reduce a estas dos operaciones. Se hace necesario clasificar una computadora con base en su función, por la manera en que ésta va a ser diseñada. Un ejemplo de esto son las computadoras para juegos, frente a las estaciones de trabajo de una entidad bancaria. Una computadora destinada al mercado de los juegos, al igual que las estaciones de trabajo de una entidad bancaria, captura información mediante dispositivos de entrada, la procesa y muestra un resultado en pantalla. Aunque no se necesita ser un genio para ver que el enfoque y el uso de los recursos es totalmente distinto. Mientras en una entidad bancaria lo que se procesa es la información de los cuentahabientes, en una computadora para juegos el proceso y los recursos de centran en la generación de *pixels* y geometría avanzada con base en los movimientos del usuario y la inteligencia artificial del juego.

Aunque existen más formas para clasificar una computadora. La clasificación respecto a tareas es una de las más aceptadas. En el área de los *clusters* es igual. Un *cluster* se clasifica respecto de la tarea que realiza y la forma en que la realiza. Esta clasificación se describe a continuación.

2.3 . *Clusters* de alta disponibilidad (HA)

El concepto de *clusters* es bastante grande y no sólo se trata de realizar tareas rápidas. Se trata de reunir varias computadoras para ayudar a completar una tarea sin importar si ésta es paralelizable o no. En este sentido, se pueden elaborar *clusters* para completar una tarea y garantizar hasta cierto punto que ésta pueda ser completada. Estos *clusters* se conocen como *clusters* de alta disponibilidad. Un buen ejemplo de esta tecnología son los *clusters* conformados por servidores apache para garantizar el funcionamiento de sitios *web* con alto número de peticiones a sus servidores[6].

2.4 . *Clusters* de alto desempeño (HPC)

Un *cluster* de alto desempeño es bastante acoplado, donde se utilizan algoritmos para resolver problemas o simulaciones. Cada nodo procesa información y ejecuta programas. A diferencia del anterior caso, un *cluster* HPC actúa con API's de PVM o MPI, que son básicamente API's de comunicación, para que los algoritmos que corren en paralelo puedan compartir información entre si.

6 Modproxy Balancer - http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html

Este es el mayor uso de *clusters* en el nivel científico, como en el caso del El Earth Simulator en Japón que se encarga de generar una simulación a futuro del estado del clima en el planeta[7].

2.5 . *Clusters* de alto desempeño (HTC)

A diferencia de los *clusters* HPC, los *clusters* HTC no son pensados para grandes y complejos procesamientos. Son pensados para cumplir una tarea en el menor tiempo posible, distribuyendo el trabajo entre los nodos sin que éstos, necesariamente tengan comunicación. Esta clasificación es la más importante, ya que el trabajo y el *cluster* con el cual se va a experimentar se ubican acá. La producción de paquetes en *software*, como Distcc o *Icecream* se hace procesando la información de manera distribuida, donde un paquete escrito en C/C++ se compone de varios archivos .cpp los cuales se traducen en los nodos del *cluster* y se enlazan en el nodo maestro distribuyendo la carga y aprovechando al máximo los recursos de las computadoras, sin que esto involucre una alta comunicación entre nodos.

7 Earth Simulator Center - http://www.jamstec.go.jp/esc/index.en.html

2.6 . Partes que conforman un *cluster*

Un *cluster* es más que un grupo de computadoras conectadas y que trabajan juntas de alguna forma. Aunque un *cluster* se fundamenta en interconexión de computadoras al igual que una red LAN lo que hace a un *cluster* diferenciarse de una red de computadoras normal es la función que desempeña y como está diseñado. Los componentes de los *clusters* se definen por los mismos campos que componen las redes de computadoras típicas: *Hardware*, Medios de Comunicación y *Software.*

2.7 . *Hardware* de *cluster*

El *hardware* de *cluster* no es tan distinto a otro tipo de *hardware* en realidad, los nodos que son utilizados en un *cluster* son simples computadoras y el diseñador del *cluster* debe elegir sus prestaciones. Desde estaciones de trabajo, hasta sistemas de altas prestaciones, con múltiples procesadores; depende cuánto necesitemos y de cuánto dinero dispongamos. Aunque los *clusters* pueden tener nodos heterogéneos, se recomienda que éstos tengan las mismas características. Esto porque si los nodos son muy dispares se producen cuellos de botella al asignar y procesar las tareas. Por el simple hecho de que los nodos más rápidos tengan que esperar a que los nodos más lentos terminen sus tareas y esto resulta contraproducente.

2.8 . Medios de comunicación

El medio de comunicación sólo es la interfaz de conexión. Dependiendo del tipo de *cluster* que se vaya a elaborar, se debe de elegir el medio, dependiendo de la interacción que exista entre los nodos. En *clusters* HA, basta con una conexión 10/100 como básica y una conexión 100/1000 entre los nodos como ideal. Parámetro que también se aplica a un *cluster* HTC. Sin embargo, una conexión 100/1000 mbps sería lo mínimo para un *cluster* HPC, donde el estándar son conexiones de fibra óptica en *mirinet* o *infiniband*, para no provocar retrasos y lograr que el procesamiento entre los nodos del *cluster* sea lo más transparente posible.

2.9 . Computación de *Grid* y Computación P2P

Con la creación de Internet la evolución natural de muchas tecnologías ha sido la interconexión de recursos y la masificación hacia todo el globo. La supercomputación no ha sido la excepción.

Así como la evolución natural de los programas de escritorio han sido las aplicaciones *web* colaborativas. Los *grids* de computación y las redes P2P son la evolución natural de los *clusters*. Cuyo principio de funcionamiento es el mismo: “Enlazar computadoras dispersas en distintas zonas geográficas” para utilizar sus recursos.

Para crear redes computacionales alrededor del planeta se han usado distintos esquemas, los cuales tienden a converger en dos “ramas” de la supercomputación distribuida. Estas dos ramas no tienen una clara división y muchas veces se les nombra simplemente como *Grids.*

Rajkumar Buyya define a los *grids* como "un tipo de sistema paralelo y distribuido que permite compartir, seleccionar y agregar recursos de distintos orígenes"[8]. O, en otras palabras, un *Grid* no es más que una interconexión de supercomputadoras dedicadas al procesamiento.

A su vez, el escritor americano Clay Shirky, describe a las tecnologías P2P con características similares a las de un *grid* "Peer to Peer son aplicaciones que toman ventaja de los recursos, almacenamiento, ciclos de procesamiento y el contenido disponible en la red".

Si ambas, definiciones apuntan a compartir recursos, vale la pena preguntarse ¿qué diferencia la computación de *grid* de la computación P2P?. La respuesta a esto reside en la forma que estos recursos se asignan y gestionan.

Mientras los centros de supercomputación del mundo toman ventaja de las redes de alta velocidad para compartir la información y distribuirse el procesamiento disponible en distintas zonas geográficas, por ejemplo la red de supercomputación española, existe otra tendencia a crear redes colaborativas conformadas por redes conformadas por computadoras de menor potencia, pero con mayores nodos conformando la red.

8 Buyya Grid Computing Info Center - http://www.gridcomputing.com/

Las redes P2P están elaboradas por computadoras de nivel doméstico, donde los nodos los conforman nodos de usuarios dispuestos a colaborar o centros de computación menos sofisticados, dedicando recursos sin utilizar en las computadoras que forman parte de la red P2P. Entre las diferencias importantes se encuentran:

- Recursos: en un *grid* se pueden utilizar los recursos disponibles, mientras que en una red P2P de computación, las aplicaciones se orientan más a utilizar los ciclos de procesamiento sin utilizar.

- Anonimato: en una red P2P el descubrimiento de nuevos nodos es descentralizado y se puede garantizar el anonimato.

- Control: mientras en un *Grid* se conocen todos los miembros que estarán disponibles, en una red P2P de computación esto no es posible y se tiene menos control de quienes se unen a la red, creando así importantes retos para la seguridad.

Para aclarar estas diferencias, Buyya elaboró la siguiente tabla en el marco de la "P2P *conference*" en Linkoping, Suecia.[9] La tabla describe las similitudes y diferencias de un *cluster*, contra un entorno de *Grid* y un entorno de P2P.

9 Gridbus Technologies for Service-Oriented Cluster and Grid Computing, 2nd IEEE International Conference on Peer-to-Peer Computing (P2P 2002), Linkopings, Sweden, September 5-7, 2002.

Tabla I. **Similitudes y diferencias con *clusters*, entornos P2P y *grid***

Características	*Cluster*	*Grid*	P2P
Computadoras miembros	Computadoras dedicadas	Computadoras dedicadas conectadas	Computadoras domésticas o de escritorio
Administración	Única	Múltiple	Múltiple
Formas de descubrimiento de nodos	Servicios y configuraciones	Índice centralizado e información descentralizada	Totalmente descentralizada
Administración de usuarios	Centralizada	Descentralizada	Descentralizada
Administración de recursos	Centralizada	Distribuida	Distribuida
Colocación y planificación de recursos	Centralizada	Descentralizada	Descentralizada
Escalabilidad	En el orden de los cientos de ordenadores	En el orden de los miles de ordenadores	Probablemente millones
Disponibilidad	Garantizada	Varia pero regularmente alta	Depende de los nodos y los recursos disponibles
Carga de trabajo	Media	Alta	Muy alta
Latencia/Ancho de banda	Baja/Alto	Alta/Baja	Alta/Baja

Fuente: Rajkumar Buyya, P2P conference, Linkoping, Suecia

2.10 . La compilación de paquetes como una aplicación HTC

Aunque el principio básico del funcionamiento de una computadora es el procesamiento de datos (que son información al adquirir valor semántico para nosotros); la naturaleza misma de los datos dictaminará de qué forma se van a procesar. Por ejemplo, las investigaciones asistidas por computadora generalmente se utilizan en campos que nada tienen que ver con la informática y un investigador sin capacitación y/o experiencia previa en herramientas de procesamiento de datos llegaría a la simple conclusión de que sí un *cluster* procesa más información automáticamente cualquier tarea será terminada más rápido, lo cual es cierto sólo en parte.

La diferencia entre una computadora de última generación con precio elevado y un *cluster* formado por computadoras de hace 3 o cuatro años no radica en la capacidad de RAM, disco o de capacidad de procesamiento. Factores que en teoría pueden ser exactamente los mismos, radica en la forma en que pueden procesar la información.

Si partimos de que no todas las aplicaciones pueden ser utilizadas en ambientes de *cluster*, vale la pena preguntar ¿Por qué la compilación de paquetes sí puede ser utilizada en *clusters*?. Y algo más importante ¿qué determina que sea un *cluster* HTC?.

El criterio genérico para decir si nuestra aplicación puede ser utilizada en un *cluster* o no es, sí nuestra aplicación tiene que procesar datos de manera secuencial. Este pareciera ser el criterio obvio pero a su vez el más importante. Aunque casi cualquier tarea se puede reprogramar para que utilice paralelismo en lugar de un hilo simple. No todas las tareas deben procesarse paralelamente. Tal es el caso de una película de alta definición, ya que aunque está formada por cuadros de imágenes independientes, estos cuadros deben procesarse de manera secuencial y a gran velocidad para que la película tenga sentido como tal, es decir, simule el movimiento real.

En el caso de los *clusters* de compilación de paquetes, este requisito es cumplido a cabalidad ya que un programa binario en Unix no es más que la reunión de códigos objeto producido a partir de archivos de código fuente. Así que estos archivos no tienen que ser traducidos a código objeto de una manera totalmente secuencial.

Ahora bien, lo que ubica a un *cluster* de compilación en el terreno de los *cluster* HTC es la independencia de las tareas que va a realizar cada nodo. Esta independencia viene dada por dos factores:

- Los datos que deben o no compartir

- Si el producto final puede ser producido fácilmente con base en muchos resultados de distintas tareas

La mayoría de compiladores sólo procesan un archivo por instancia, y si un programa es compuesto por varios archivos, las instancias de compiladores que se ejecuten en los distintos nodos del *cluster* no deben de compartir mayor información entre sí y sólo debe informarse al nodo maestro de cuando una tarea inicia o termina.

Además, cuando todos los archivos han sido traducidos, se pasa a la fase de enlace, ya que en la práctica, un archivo ejecutable binario requiere funciones de todos las unidades lógicas de programación que componen la aplicación e incluso de funciones externas de otros programas que se enlazarán de forma dinámica.

Así pues, un *cluster* de compilación de paquetes es definitivamente un *cluster* HTC y que en la vida práctica incluso podría realizarse en un entorno de *grid computing* distribuido.

3 . SISTEMAS DE PAQUETES EN DISTRIBUCIONES LINUX

3.1 . Definición y funcionamiento de los paquetes

Los sistemas basados en Unix se distinguen en la forma cómo se gestionan los programas. Desde sus inicios, Linux se ha caracterizado por ser un sistema parecido pero no exactamente Unix. Por tal motivo cumple con lo que Doug McIlroy llama “Filosofía Unix”[10]. Ésta se resume en tres pautas simples:

- Escribe programas que hagan una sola cosa y la hagan bien
- Escribe programas que trabajen juntos
- Escribe programas que manejen cadenas de texto, pues ésta es la interface universal

Distribuciones comunitarias como Debian llegan a tener a disposición de sus usuarios más de 17,000 programas listos para ser instalados y utilizados. Por este motivo, se hace necesario que éstos se distribuyan en lo que se conocen como paquetes.

10 Filosofia Unix - http://www.linuxlots.com/~dunne/unix-philosophy.html

Un paquete en Linux no es más que un programa preparado para su instalación. Generalmente, el paquete está comprimido en formatos libres. Son archivos que incluyen el ejecutable binario del programa, reglas de instalación, archivos de configuración o algunas otras características como problemas conocidos con otros paquetes en el mismo sistema.

A diferencia de otros sistemas operativos, el sistema de paquetes de Linux constituye una fortaleza en la organización del sistema operativo, a su vez, una debilidad para los administradores de sistemas.

La filosofía de Unix provoca que sus aplicaciones se diseñen pensando en compartir instrucciones entre sí. Por ejemplo, para que un simple reproductor de audio funcione, se necesitan programas que accedan a las funciones del sistema de sonido ALSA, bibliotecas que entiendan los formatos a reproducir, y para dibujar mediante los controles de la interfaz gráfica.

En Unix todos los programas están pensados para trabajar juntos y que hagan una sola cosa bien. Un paquete casi siempre depende de otro para realizar sus tareas, lo que se conoce como dependencias del paquete. Esto sumado a la característica de desarrollo comunitario y ágil de Linux hace que las dependencias aumenten.

3.2 . Administradores de paquetes

Para tener un entorno funcional, sin ensamblar el sistema operativo a mano, las distintas formas de distribución de sistemas operativos basados en Linux, implementan programas que automatizan tareas como: instalación, actualización, configuración y resolución de dependencias para un programa; los cuales se conocen como administradores de paquetes.

En casi todas las distribuciones Linux, el administrador de paquetes se incluye como una herramienta básica de administración, aunque no es necesaria para su funcionamiento. Proveen un sistema unificado de instalación, almacenando la información acerca de los paquetes disponibles para su instalación.

Aunque el desarrollo de cada distribución depende de una o más personas que se dan a la tarea de reunir los paquetes en un único sistema operativo, acorde a un público o necesidad específica; casi todas las distribuciones Linux utilizan tres formas para gestionar estos paquetes.

Las tres formas más comunes para la gestión de paquetes en sistemas GNU/Linux son:

- Gestionar los paquetes que proveen los proyectos de distribuciones convenientemente en código binario

- Gestionar la creación de estos paquetes mediante el código fuente de los mismos y la construcción local en código binario

- Sistemas híbridos que no son más que la reunión de los anteriores

Para que el administrador de paquetes realice estas funciones, por lo regular se auxilian de *backends*, que en versiones anteriores de estos sistemas fueron las únicas utilerias que existían para instalar los paquetes, siguiendo la filosofía Unix la cual establece que cada programa debe de hacer una tarea y hacerla bien.

La siguiente tabla da un ejemplo de cómo distintos administradores de paquetes utilizan la misma utileria de *backend*.

Tabla II. **Administradores de paquetes y backends**

Sistema	Administrador de paquetes	*Backend* (utileria de instalación)
Debian Linux	aptitude	dpkg
Mac Os	Fink	dpkg
Mandriva Linux	Urpmi	rpm
Red Hat Linux	Yum	rpm

Fuente: elaboración propia

3.3 . Administradores basados en binarios

Desde la creación de sistemas Unix por los Bell Labs[11], el sistema y la forma de instalación de paquetes se consideraba como tarea de administradores capacitados o usuarios especializados. En sus primeras versiones, la tarea consistía en configurar el paquete con las bibliotecas de funciones instaladas en el sistema y compilarlo para generar un binario que luego sería instalado en el sistema. Sin embargo, no cualquier persona puede dedicar tanto tiempo a estas tareas, así que en 1984 se puso a disposición de la mayoría de los Unix la utileria pkgadd. Pkgadd fue una de las primeras herramientas para instalación de paquetes binarios que básicamente consistía en congelar el estado y los archivos de un paquete, para luego desempacarlos sobre otro Unix y ejecutar algunas configuraciones extras.

11 The creation of Unix, Bell Labs - http://www.bell-labs.com/history/unix/

Hasta entonces, las utilerias de instalación de programas han evolucionado a lo que hoy conocemos como administradores de paquetes y en el caso de los paquetes binarios, introducen los conceptos de integración continua y repositorios.

Para entender el funcionamiento de los administradores de paquetes binarios debemos de entender cómo funcionan la mayoría de proyectos de *software*. Cuando un desarrollador o colaborador de una distribución Linux está interesado en generar un paquete nuevo o actualizar alguno ya existente, se da a la tarea de crear la estructura básica del mismo. Luego de que éste es revisado por algún equipo, si son paquetes críticos, el paquete se integra en un entorno de integración continua. Dado que la mayoría de programas en Linux son de libre distribución, cualquiera puede tener acceso al código, que por lo regular está en lenguajes interpretados uniformes entre arquitecturas de procesadores como *python, perl o java*. O si no están escritos en lenguajes de bajo nivel como C++ o C, los cuales pueden ser utilizados por el compilador por defecto en las distribuciones Linux: GCC. Al ser introducidos al sistema de integración continua, GCC se encarga de generar paquetes para todas las arquitecturas de procesador que el proyecto soporte. Desde unas cuantas como en el caso de Red Hat/Fedora, hasta la mayoría de arquitecturas existentes, como en el caso de Debian.

Si los paquetes no presentan problemas en su construcción, se integran en servidores que se conocen como repositorios. Un repositorio no es más que un servidor *web* o servidor FTP, cuya tarea es distribuir los paquetes generados con la integración continua a los usuarios de determinada distribución Linux.

Una vez los paquetes están disponibles, es cuestión de los administrador de paquetes conectarse a ellos para descargarlos vía internet (o verificar si no existen paquetes en su caché de descargas). Y, de esta manera se crea un sistema centralizado para la gestión de programas en distribuciones Linux.

3.4 . Administradores basados en código

Una de las particularidades que ha distinguido a UNIX es su versatilidad que, a diferencia de otros sistemas operativos, ha sido realizado gracias a la colaboración de varias compañías o usuarios. Esto, independientemente de el Unix en cuestión es libre o no, ha contribuido a que hoy por hoy se considere a los sistemas basados en Unix o con principios de Unix, sistemas bastante versátiles.

Es un hecho que compilar un paquete siempre es una tarea complicada frente a instalar un paquete binario, pero el acto de compilar un paquete puede ser deseable en alguno de estos casos:

- Aplicar parches de corrección de errores al código fuente, sin depender de la distribución

- Activar optimizaciones que los compiladores de C presentan y generar binarios óptimos para él procesador

- Utilizar paquetes no disponibles dentro de la distribución Linux que estemos trabajando

- Usar versiones más recientes de dichos programas

Aunque los administradores de paquetes basados en código realizan las mismas tareas que los paquetes binarios, éstos basan su funcionamiento en scripts o instrucciones para construcción de los paquetes de manera local. Accediendo a las ventajas mencionadas anteriormente, pero con el costo de gastar ciclos de procesador en producir los binarios.

Para que una distribución de éstas funcione, los desarrolladores de la misma realizan estos *scripts* y los ponen a disposición a sus usuarios, con el código fuente en servidores del proyecto en algunos casos. Los *scripts* incluyen, al igual que un paquete tradicional, la dirección desde donde va a ser descargado el código fuente, programas de los cuales depende e instrucciones especiales; para luego ser compilados e instalados en el sistema. Se dice que este tipo de sistemas deberían ser utilizados por usuarios más experimentados porque a costa de versatilidad en la administración se sacrifica facilidad.

Al final de cuentas, con la evolución que han tenido los administradores de paquetes, permiten que administradores cuyo enfoque principal es la instalación de paquetes binarios puedan descargar códigos fuente e instalarlos o bien, que administradores basados en código fuente instalen paquetes de código cerrados que son necesarios para el funcionamiento del sistema, sin que esto signifique perder su enfoque principal.

O como ha sucedido últimamente, han surgido distribuciones cuyo administrador de paquetes pretende cubrir a cabalidad ambas funciones, como es el caso de pacman en Arch Linux o equo en Sabayon Linux.

4 . *CLUSTERS ICECREAM* Y DISTCC

4.1 . Estructura un *cluster* de compilación paralela

Independientemente del *software* que se utilice para la compilación paralela, un *cluster* de tipo HTC tiene una estructura “genérica” con al menos los siguientes componentes:

- Un medio físico de conexión
- Un nodo para las siguientes tareas:
  - Control central del *cluster*
  - Distribución de la carga de trabajo
  - Presentación de informes de estado acerca del *cluster*
- Nodos dedicados al procesamiento

Dependiendo de la utilidad del *cluster,* el nodo principal o administrativo sirve también como un punto de entrada hacia el *cluster*. Inclusive los *cluster* P2P tienen un nodo principal, desde el cual se descubren nuevos nodos.

Dependiendo de la importancia del *cluster*, es recomendable agregar una réplica del nodo de control porque constituye un punto único de fallo, haciendo imposible la operación del *cluster* en caso de fallo.

4.2 . Cómo se genera un ejecutable con GCC

Para entender cómo funciona el *cluster* primero debemos de entender cómo funciona un compilador de aplicaciones y, en este caso, el compilador GCC; a la hora de producir un ejecutable el compilador GCC atraviesa cuatro fases de operación y además de esto se ejecuta una fase previa de preparación del código fuente para el sistema operativo de destino[12]. Para entender este ejemplo en un sistema Linux, utilizaremos un programa básico que sólo se encargará de imprimir la oración “cuatro fases de gcc” y el archivo se llamará cuatro.c.

```
#include <stdio.h>

int
main (void)
{
  printf ("Cuatro fases de gcc\n");
  return 0;
}}
```

El compilador GCC nos permite ejecutar cada fase independientemente, aunque su modo de operación por defecto es ejecutar las cuatro fases sin intervención del usuario.

12 An introduction to GCC – An overview of compilation process http://www.network-theory.co.uk/docs/gccintro/gccintro_83.html

4.2.1 . Preprocesamiento de las directivas de compilación

Actualmente, los sistemas Unix y los sistemas diseñados con principios Unix tienen la particularidad de no ser producidos por una sola compañía de *software*. Así pues, la marca Unix, como tal, se le puede dar a cualquier sistema operativo que demuestre cumplir con los estándares del Instituto de Ingenieros Eléctricos y Electrónicos, específicamente el estándar IEE 1003 o estándar de sistemas POSIX[13]. Los estándares POSIX fueron definidos para que una aplicación desarrollada para un sistema UNIX, pueda ser ejecutada en otro sistema UNIX, utilizando las mismas interfaces de programación.

Sin embargo, esta tarea, a la larga, resulta complicada, ya que para producir un ejecutable UNIX diseñado para un otro sistema UNIX distinto, es necesario indicarle al programa dónde debe buscar por sus dependencias o, dicho de otra forma, dónde debe buscar dentro del sistema las bibliotecas de funciones y programas externos necesarios para su funcionamiento.

Por este motivo, compiladores como GCC utilizan archivos de configuración denominados *Makefiles,* cuyo objetivo es contener estas referencias hacia las bibliotecas de funciones y programas externos, haciendo el proceso de compilación en distintas plataformas mucho más práctico.

Estos archivos pueden ser producidos por varias herramientas como cmake o automake que únicamente difieren en la forma que configuran los archivos pero su objetivo es el mismo.

13 UNIX Certification Process - http://www.opengroup.org/certification/

Entonces, cuando un programa va a ser compilado en una plataforma el primer paso antes de procesar el código es generar estos archivos de configuración para asegurar que el proceso de compilación va a terminar exitosamente, y sobre todo con las referencias correctas hacia las bibliotecas de funciones y programas externos.

4.2.2 . Preprocesamiento del código fuente

Al igual que la mayoría de lenguajes de alto nivel C y C++ soportan el uso de bibliotecas de funciones mediante el uso de la sentencia #include. Lo cual en terminología de gcc se conoce como una instrucción macro. Entonces en esta primera fase el preprocesador expande estas macros hacia los archivos de código fuente. Podemos ver esta fase en funcionamiento ejecutando el siguiente comando en una terminal de Linux ubicados en el directorio que contenga nuestro código fuente (cuatro.c).

```
$ cpp cuatro.c > cuatro.i
```

Con esto vamos a lograr expandir nuestro archivo de código fuente, describiendo todas las bibliotecas necesarias que incluimos con la macro #include y a su vez las dependencias de estas bibliotecas. Si revisamos el archivo cuatro.i obtendremos un archivo con la siguiente estructura, describiendo todas las bibliotecas de funciones que serán involucradas en el proceso.

```
# 1 "cuatro.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "cuatro.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
. . .
```

4.2.3 . Compilación del código fuente

GCC se define como un compilador multiplataforma, tanto a nivel de sistema operativo, como a nivel de arquitectura de procesador. Para que esto sea posible, GCC necesita hacer una compilación de dos fases. La primer fase consiste en construir código en lenguaje ensamblador, el cual depende de la arquitectura sobre la cual se ejecutará el programa y la segunda fase donde ese código en ensamblador, se genera como un programa ejecutable binario.

Al igual que en la fase de preprocesamiento podemos ver el código ensamblador con el siguiente comando en la terminal de Linux:

```
$ gcc -Wall -S cuatro.i
```

Y con esto GCC realiza la primer compilación hacia el ensamblador, generando el archivo cuatro.s. Nuestro archivo en código ensamblador tendrá la siguiente estructura:

```
        .file   "cuatro.c"
        .section        .rodata
.LC0:
        .string "Cuatro programas de gcc"
        .text
.globl main
        .type   main, @function . . .
```

4.2.4 . Ensamble de código fuente

Una vez tenemos el archivo en ensamblador, se produce el código objeto o código máquina en un archivo ejecutable ELF específico para el procesador. Si existieran referencias a bibliotecas funciones, las direcciones en esta fase se dejan vacías las direcciones de memoria para que el enlazador haga referencia posteriormente. Ahora procedemos a ensamblar el código con el siguiente comando en una terminal de Linux:

```
as cuatro.s -o cuatro.o
```

A diferencia de las fases anteriores no podemos ver el código a simple vista, ya que hasta este punto ya tenemos código binario. De ser necesario, podemos ver el descriptor de código fuente con el lector de archivos *less*.

4.2.5 . Enlace dinámico del código fuente

Por último el compilador GCC procede a enlazar dinámicamente o dicho de otra forma a empaquetar uno o más archivos *.o en un ejecutable que hará referencia a las bibliotecas de funciones disponibles en el sistema. Este paso final lo hacemos con el siguiente comando, en una terminal de Linux:

gcc hello.o

Y con esto ya podemos ejecutar nuestra aplicacion:

./a.out
Cuatro fases de gcc

4.3 . Funcionamiento básico de *Icecream* y Distcc

Para el procedimiento de compilación sea posible tanto *Icecream* como Distcc utilizan, se emplea una arquitectura de cliente/servidor. En esta arquitectura, el nodo de control funciona a manera de cliente y en cada nodo de compilación se instala un servicio que se encargará de enviar y recibir información además producir el código objeto.

Para que el nodo de control funcione de manera transparente, *Icecream* y Distcc utilizan archivos intermediarios con el nombre original de los ejecutables de GCC (gcc, g++, cpp, c++) los cuales en lugar de elaborar las cuatro fases del compilador de manera local; únicamente preprocesan el código y envían las directivas de preprocesamiento, y los archivos necesarios para compilar y producir el código objeto en los nodos del *cluster*.

Todo el funcionamiento del *cluster* se realiza bajo la siguiente secuencia de pasos[14].

4.3.1 . Preprocesamiento del código

En el nodo maestro se procesan todas las macros “#include” que se encuentran dentro del código fuente. En este procesamiento se expanden las macros al código que será compilado.

4.3.2 . Asignación del nodo

Icecream y Distcc funcionan de manera similar, con un algoritmo básico de hilos de ejecución por nodo. La regla por defecto es que un nodo puede ejecutar N+2 tareas de compilación donde N es el número de procesadores que tiene el nodo.

14 IBM - Distributed Compilation: A programs delight - http://www.ibm.com/developerworks/aix/library/au-dist_comp/?ca=dgr-lnxw01ProgramDelite&S_TACT=105AGX59&S_CMP=GRsitelnxw01

4.3.3 . Envío del código fuente hacia el nodo

Una vez se ha preprocesado el código fuente, se envía el código fuente junto con las cabeceras de las bibliotecas de funciones y el archivo de preprocesamiento.

4.3.4 . Compilación en el nodo

Cuando los servicios de *Icecream* o Distcc han recibido correctamente el código, proceden a compilar el código con las opciones de compilación CFLAGS y CXXFLAGS que se indicaron en el nodo maestro. Así se produce el código ensamblador para la arquitectura de procesador correcta y se produce el código objeto.

4.3.5 . Recepción del código fuente de parte del nodo

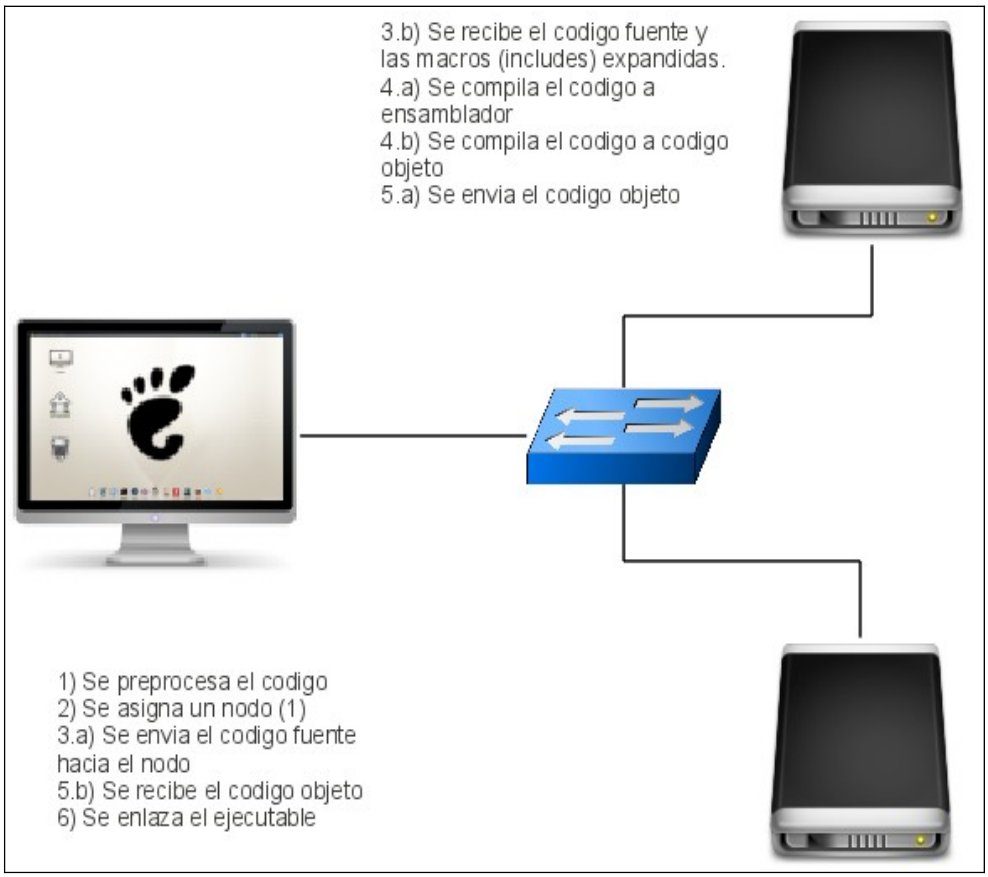
Cuando el código de algún nodo está listo se envía de vuelta al cliente, el cual tiene registrados todos los nodos y qué archivos se están compilando en cada nodo.

4.3.6 . Enlace dinámico de los archivos fuente.

Cuando todos los archivos han sido compilados y se cuenta con todos los archivos de código objeto (generalmente con extensión .o) se procede al enlazamiento dinámico para producir el ejecutable final.

Este proceso se ilustra en el siguiente diagrama:

Figura 1. **Proceso de enlace dinámico de archivos fuente**



Fuente: elaboración propia

4.4 . Diferencias entre *Icecream* y Distcc

Aunque, *Icecream* sea patrocinado por Novell y Distcc sea un proyecto 100% comunitario con apoyo de otras empresas, *Icecream* y Distcc comparten muchas partes de su lógica de funcionamiento. Esto debido a que *Icecream* inició como una bifurcación del proyecto Distcc para agregar nuevas características. Hasta la fecha, ambos proyectos han seguido su propio camino y las diferencias más importantes son:

4.4.1 . Planificación en entornos heterogéneos

A diferencia de Distcc, *Icecream* basa su funcionamiento no sólo en un cliente, sino además en un servidor central para realizar algunas tareas extras. *Icecream*, además de distribuir el código a todos los nodos, puede evaluar cuál es el nodo de compilación con más poder de procesamiento disponible, basado en el valor nominal de velocidad del procesador y la carga promedio del procesador. Lo que permite una mejor planificación para nodos que no se dedican totalmente a tareas de compilación.

4.4.2 . Soporte para distintas versiones de GCC

Icecream tiene soporte para un entorno de versiones de GCC mixtas. Esto sin embargo, con el costo de deshabilitar optimizaciones especificas de algunas arquitecturas. Distcc puede trabajar con distintas versiones de GCC, siempre y cuando sean de la misma rama, por ejemplo un nodo con compilador versión 4.4.3 y otro con compilador versión 4.4.1.

4.4.3 . Búsqueda dinámica de *hosts*

Icecream permite la reconfiguración de los *hosts* participantes sin intervención del usuario, mediante un servidor DNS, mientras que la configuración de Distcc es a partir de un listado de *host* en un archivo estático.

4.4.4 . Modo de distribución *pump*

Uno de los aportes más importantes para Distcc fue la colaboración de Google para la creación de un modo de operación *pump*. En este modo de funcionamiento, Distcc utiliza un algoritmo incremental estático para identificar los archivos necesarios para ejecutar la fase de preprocesamiento en los nodos el código fuente en los nodos y también distribuir esa carga de trabajo contrario al modo de operación normal donde el preprocesamiento únicamente se hace en el nodo maestro del cluster.

4.5 . Aplicaciones para la compilación distribuida

Antes de continuar con el experimento, vale la pena definir algunos escenarios donde este tipo de herramientas tienen una aplicación práctica. O dicho de otra manera, enumerar algunos escenarios donde el uso de compilación distribuida puede significar ahorro de tiempo, esfuerzo y dinero.

4.5.1 . Entorno 1: grupo de desarrolladores

Aunque, se piense que el uso de lenguajes como C o C++ va en decadencia comparado con lenguajes de nueva generación, lo cierto es que según la compañía TIBOE[15], los lenguajes C y C++ siempre han sido parte de los 10 lenguajes más utilizados a nivel mundial desde el año 1985 hasta el año 2010. Y de hecho hasta el mes de Junio de 2010 ocupan el segundo y tercer lugar respectivamente.

Si tomamos en cuenta estos factores, suena lógico afirmar que aún hay muchas empresas desarrollando en este tipo de lenguajes, principalmente en aplicaciones legadas o en aplicaciones nuevas, que requieren un alto nivel de control sobre el código.

15 TIBOE Programming Comunity Index - http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

En teste tipo de entornos generalmente existen grupos de desarrolladores, cada uno con una estación de trabajo y con el mismo problema de productividad: “compilar un programa demanda tiempo y recursos”. Si a esto le sumamos que se requieren construcciones para todos cada vez que alguien produce un cambio mayor.

En este tipo de entornos los compiladores distribuidos son la solución para reducir los tiempos de compilación. Ya sea creando, utilizando una poderosa computadora o creando *clusters* de computadoras, cuya única tarea sea compilar o incluso aprovechando el poder de procesamiento de las estaciones de trabajo. Imponiendo límites a los compiladores y aprovechando los ciclos donde el procesador no esté siendo utilizado a toda su capacidad.

Compilaciones rápidas se traducen en depuración de código más rápida y abre la posibilidad de pruebas de aplicaciones más frecuentes.

4.5.2 . Entorno 2: distribuciones Linux basadas en código

Uno de los aspectos más interesantes de una distribución Linux, basada en código, es la optimización de las aplicaciones acorde a la arquitectura del procesador donde será ejecutada. Distribuciones, como: Gentoo Linux o Sorceress Linux, nacen con el objetivo de brindar al usuario la opción de generar binarios ejecutables, acorde con la arquitectura de la computadora, mediante las variables de optimización que proporciona el compilador del proyecto GNU: GCC.

Sin embargo, estas optimizaciones sólo son posibles mediante la compilación especifica de los ejecutables y se le delega a la computadora del usuario la producción de estos binarios ejecutables, porque es prácticamente imposible producir binarios con los distintos tipos de optimizaciones que existen.

Para los usuarios de este tipo de distribuciones, los compiladores distribuidos son una solución para acelerar el proceso de la producción de binarios ejecutables. Basta con tener dos equipos con capacidad para compilar.

4.5.3 . Entorno 3: sistemas operativos y aplicaciones embebidas

En dispositivos inteligentes es bastante común que se utilicen versiones personalizadas de sistemas Linux o variantes de sistemas operativos BSD como sistemas y para la ejecución de aplicaciones, que controlen los dispositivos. Como por ejemplo teléfonos celulares, *routers, firewalls.*

Sin embargo, este tipo de dispositivos tiene la particularidad de utilizar procesadores de bajo consumo de energía, bajo poder y una cantidad de memoria RAM limitada. En este caso un compilador distribuido puede utilizarse para el desarrollo de aplicaciones y distribuir la tarea de compilación. Evitando con esto, que sea necesario introducir copias del código y/o binarios una y otra vez al dispositivo inteligente. Acelerando los tiempos de desarrollo y depuración de código.

Algunos usuarios notables de este tipo de compiladores:

- Google (nuevo mantenedor oficial de Distcc)[16]
- Gnome Fundation[17]
- Apple Xcode Developer Tools[18]
- Novell[19]
- MIT[4]
- Organización Europea para la Investigación Nuclear (CERN)[20]
- Seneca Centre for Development of Open Technology (CDOT)

16 http://code.google.com/p/distcc/

17 http://people.gnome.org/~michael/blog/icecream.html

18 http://distcc.samba.org/faq.html

19 http://distcc.samba.org/results.html

20 https://twiki.cern.ch/twiki/bin/view/LinuxSupport/DistccPilotService#Who_can_use_the_lxdistcc_cluster

5 . EXPERIMENTACIÓN CON LOS *CLUSTER*

5.1 . Descripción del experimento

Aunque *Icecream* está basado en Distcc, ambos han hecho sus propios aportes de código de manera independiente, por tal motivo se generan diferencias en la forma de funcionamiento, configuración y opciones entre *Icecream* y Distcc. Lo que este estudio pretende evaluar y comparar es si estos cambios han producido una mejora del desempeño a nivel técnico y también si el nivel técnico es o no un factor significativo en la elección de la plataforma. Y si no se encontrara una diferencia técnica significativa, se evaluarán otros factores como la facilidad de administración y opciones de soporte.

Para poder establecer esta información, se realizó una comparación entre ambas herramientas y varios escenarios y se modelara un modelo TAM a partir de factores clásicos de complejidad computacional. Los factores tomados en cuenta para la elaboración del modelo TAM serán:

- Tiempo total de procesamiento (compilación)
- Uso de memoria RAM
- Tamaño del paquete a compilar
- Número de nodos que forman parte del *cluster* (paralelismo)

Además de esto, el estudio también pretende establecer qué tanta diferencia existe al producir paquetes desde un determinado lenguaje. Y si la herramienta de prefiguración de los paquetes tiene o no, impacto en el desempeño.

Para realizar este experimento, la prueba se realizó produciendo en tres distintas configuraciones un entorno de escritorio KDE por los siguientes motivos:

- La mayoría de los paquetes están elaborados en C o C++
- El proyecto KDE básico está conformado por 295 paquetes, lo cual nos proporciona suficiente información y suficientes paquetes para realizar el experimento
- La compilación de estos 295 paquetes se puede hacer en Gentoo de una manera automatizada, gracias a que Gentoo es una distribución basada en código fuente y todo KDE será construido desde fuentes

Para realizar la comparativa se recrearon los siguientes escenarios de prueba:

- *Cluster* bajo Distcc sin el modo de operación *pump* habilitado
- *Cluster* bajo *Icecream*
- *Cluster* bajo Distcc con la opción *pump* habilitada
- Compilación local en una máquina virtual utilizando 3 núcleos

Con estos entornos se determinará:

- La pérdida de rendimiento de un *cluster* frente a una computadora con poder de procesamiento
- Si la opción *pump* de Distcc en verdad incrementa la eficiencia del *cluster*
- Si las diferencias entre un *cluster* Distcc y un *Cluster Icecream* sólo se limitan a facilidad de administración o los diferentes aportes de código que han tenido ambos proyectos, han logrado una diferencia a nivel técnico (sin tomar en cuenta el modo *pump* que sólo se encuentra presente en Distcc)
- Si el planificador de distribución de *Icecream* juega un papel fundamental en el rendimiento del *cluster* a la hora de ejecutar las compilaciones

5.2 . Estructura del *cluster* para la realización del experimento

Para la realización del experimento se realizó un *cluster* basado en procesadores Intel Pentium 4. La configuración de los nodos es la siguiente:

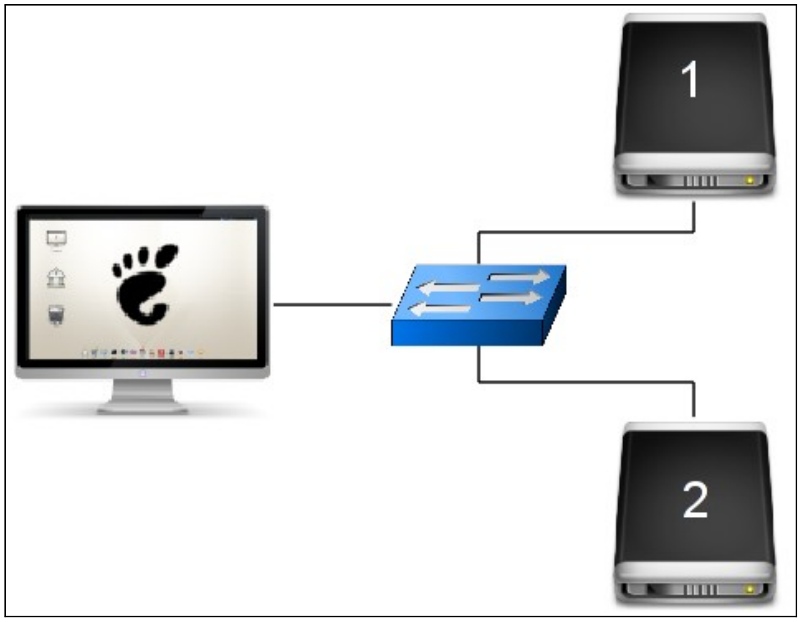
Tabla III. **Configuración de nodos en el *cluster* de prueba**

	Procesador	Cantidad de memoria RAM	Sistema operativo	Versión de *kernel*
Nodo maestro	Intel Celeron M @ 1.6 ghz	1.5 GB	Gentoo Linux inestable.	2.6.32 -optimizado-
Nodo de *cluster* 1	Intel Pentium 4 @ 2.8 ghz	768 MB	Gentoo Linux 10.0	2.6.32 -generico-
Nodo de *cluster* 2	Intel Pentium 4 @ 2.8 ghz	768 MB	Gentoo Linux 10.0	2.6.32 -generico-

Fuente: elaboración propia

Para la interconexión entre dispositivos se utilizó una red lan 10/100 básica, configurada bajo una red privada de clase c. Lo cual se demuestra mejor en el siguiente diagrama de la topología de la red.

Figura 2. **Topología de red**



Fuente: elaboración propia

5.3 . Configuración del sistema operativo de los nodos principales

Para que la medición sea más exacta, los nodos que conforman el *cluster* fueron personalizados única y exclusivamente para compilación de paquetes, para tal efecto, se eligió Gentoo GNU/Linux como la plataforma para el experimento. Según sus creadores:

> "Gentoo es un sistema operativo libre que puede estar basado tanto en Linux como en FreeBSD y tiene la capacidad de ser optimizado y personalizado automáticamente para cualquier aplicación o necesidad. La configurabilidad extrema, el rendimiento y una comunidad de usuarios y desarrolladores de primerísima son todas características de la experiencia Gentoo"[21]

21 About Gentoo - http://www.gentoo.org/main/en/about.xml

Para efectos del experimento, Gentoo se eligió sobre otras distribuciones Linux por los siguientes motivos:

- Es un sistema altamente personalizable y da toda la flexibilidad necesaria para personalizar las características del sistema operativo.

- Las herramientas que incluye Gentoo facilitan las tareas de medición y estadística de compilación de paquetes, bastante útiles para este experimento.

- Gentoo se denomina como una metadistribución Linux, lo que significa que el proyecto y su código brindan un sistema base y las herramientas necesarias para que cada usuario pueda crear su propio sistema, de acuerdo a sus necesidades.

La instalación de Gentoo se realiza en una serie de pasos bien definidos en la documentación del proyecto, la cual queda fuera del alcance de este estudio. Sin embargo, el proceso básico para preparar los nodos fue el siguiente:

5.3.1 . Instalación del sistema base

Gentoo utiliza dos herramientas básicas para su instalación: un CD *minimal* y un *Stage* de instalación Gentoo. Un CD *minimal* es un CD que se ejecuta en modo “vivo”, el cual no es más que un sistema Linux configurado para arrancar en memoria RAM, sin necesidad de instalar software en el disco duro de la computadora. Éste se utiliza para realizar los pasos de instalación ya que Gentoo no cuenta con un instalador oficial.

Entre los pasos más importantes se encuentra la copia y descompresión de una imagen genérica de un sistema Gentoo, lista para servir de base en la construcción de nuestro sistema personalizado. En esta imagen genérica se incluyen los ejecutables básicos de todo sistema Linux, así como compiladores para la construcción de los paquetes y el sistema de administración de paquetes de Gentoo denominado portage.

5.3.2 . Instalación de herramientas de compilación

Aunque Gentoo incluye la colección de compiladores del proyecto GNU (GCC por sus siglas en inglés), se agregan las herramientas para la compilación distribuida: Distcc e *Icecream*. Las cuales se configuraron después dependiendo del experimento.

5.3.3 . Instalación de herramientas para el monitoreo

Además de las herramientas para la compilación de paquetes, se necesitaron herramientas para capturar la información de los nodos. Entre las cuales se incluyen:

- Sar: es un monitor de rendimiento existente desde los primeros UNIX de Sun Microsystems, su funcionalidad es recolectar, almacenar y generar reportes de las actividades de un sistema.

- Genlop: es un analizador de registros de eventos del sistema de administración de paquetes portage de Gentoo. En el cual se pueden obtener datos respecto a la instalación de un paquete (tiempo de duración, inicio, fin, veces instalado).

- Distcc-monitor: como su nombre lo indica, Distcc-monitor es un monitor de estado para monitorizar el estado de los trabajos de compilación específicamente.

- Icecreammon: al igual que Distcc-monitor, éste está diseñado para monitorizar el proceso de compilación y distribución de código en los nodos del *cluster*, tiene las mismas atribuciones que Distcc-monitor sólo que con una interfaz gráfica más refinada.

5.3.4 . Configuración de la red

Una vez configurados los nodos, el paso final fue definir la configuración de red que tendrían los nodos, siendo asignados todos sobre una red clase C simple con cable Ethernet de comunicación 10/100. Las direcciones IP se asignaron de la siguiente manera:

Tabla IV. **Tabla de asignación de direcciones IP en los nodos**

	Dirección IP
Nodo maestro	192.168.0.1
Nodo de *cluster* 1	192.168.0.2
Nodo de *cluster* 2	192.168.0.3

Fuente: elaboración propia

5.3.5 . Configuración del sistema operativo del nodo maestro

El sistema operativo en el nodo maestro no requería de un nivel de personalización porque su única tarea será la de ordenar la compilación de los paquetes a los nodos del *cluster*, así que las únicas instalaciones extras fueron las herramientas de monitoreo del estado de los nodos del *cluster*:

- Configuración de Distcc-monitor

  Como su nombre lo indica Distcc-monitor es un monitor de estado para monitorizar el estado de los trabajos de compilación específicamente. Gentoo tiene entre sus repositorios de *software* una versión preconfigurada de Distcc-monitor así que basto únicamente con instalarla y la integración con el nodo maestro se realizó de manera automática.

- Configuración de Icecreammon

  Al igual que Distcc-monitor, Icecreammon está diseñado para monitorizar el proceso de compilación y distribución de código en los nodos del *cluster*, tiene las mismas atribuciones que Distcc-monitor pero con una interfaz gráfica más refinada. A diferencia de Distcc-monitor la configuración de Icecreammon se hace preconfigurando el paquete con la instrucción *configure,* compilando el paquete en nuestro sistema con la instrucción *make* y por último instalándolo con las instrucciónes *make install.*

6 . RECOLECCIÓN DE DATOS

6.1 . Descripción general del proceso de captura de datos

El modelo TAM en el cual se basa este experimento tiene la particularidad que sus variables dependientes e independientes se obtienen a partir de datos que genera una computadora al procesar información. Sin embargo, si bien es cierto que estos datos son relativamente fáciles de capturar, son altamente variables en periodos de tiempo pequeños, especialmente la utilización de memoria o el porcentaje de utilización del procesador, los cuales varían dependiendo de la velocidad de actualización que tenga la memoria RAM (en el orden de millones de ciclos por segundos) y el CPU (en el orden de miles de millones de ciclos por segundos).Por tal motivo, los datos se van a capturar de distinta manera para dos grupos de variables:

a) Variables que pueden ser medidas con precisión
b) Variables que pueden ser medidas con base en promedios

Tabla V. **Tipos de variables a utilizar en el experimento**

Tipo	Variables
Variables precisas	-Tiempo de compilación de un paquete -Tamaño total de paquete
Variables con base en promedios	- Uso de memoria promedio -Uso de procesador promedio

Fuente: elaboración propia

6.2 . Captura de tiempos de compilación y tamaño de los paquetes

Los datos de tiempo de compilación son almacenados en el registro del administrador de paquetes de Gentoo, donde el tiempo se registra en el formato epoch de tiempo utilizado en todos los sistemas UNIX.

Por ejemplo, para la instalación de axel, un acelerador de descargas en línea de comandos para UNIX, la estructura del registro sería la siguiente:

1256399842: Started emerge on: Oct 24, 2009 09:57:22
1256399842: *** emerge --ask --verbose --buildpkg axel
1256399845: >>> emerge (1 of 1) net-misc/axel-2.3-r1 to /
1256399846: === (1 of 1) Cleaning (net-misc/axel-2.3-r1::/usr/portage/net-misc/axel/axel-2.3-r1.ebuild)
1256399846: === (1 of 1) Compiling/Packaging (net-misc/axel-2.3-r1::/usr/portage/net-misc/axel/axel-2.3-r1.ebuild)
1256399849: === (1 of 1) Merging (net-misc/axel-2.3-r1::/usr/portage/net-misc/axel/axel-2.3-r1.ebuild)
1256399849: >>> AUTOCLEAN: net-misc/axel:0
1256399850: === (1 of 1) Updating world file (net-misc/axel-2.3-r1)
1256399850: === (1 of 1) Post-Build Cleaning (net-misc/axel-2.3-r1::/usr/portage/net-misc/axel/axel-2.3-r1.ebuild)
1256399850: ::: completed emerge (1 of 1) net-misc/axel-2.3-r1 to /

Aunque, para un usuario experimentado de Gentoo es fácil deducir lo que significan esa lineas, el proceso de reunir la información de manera manual sería muy complicado.

Los desarrolladores de Gentoo pensando en este tipo de tareas ponen a disposición de sus usuarios una colección de *scripts* denominada gentoolkit, entre las cuales se encuentra el *script* genlop.

Genlop es un *script* cuyo único objetivo es analizar los archivos de registro de portage en búsqueda de datos significativos. Específicamente tiempos de instalación de los paquetes en el sistema.

En el caso particular de esta investigación, basta con utilizar genlop en conjunto con las siguientes opciones:

-l – listado de paquetes
--*date* - fecha de inicio y/o finalización

Por ejemplo, si quisiéramos obtener el listado de paquetes que se instalaron desde el día uno de enero de 2010 hasta el 10 de enero de 2010 podríamos ejecutar:

genlop --*date* 01/01/2010 --*date* 01/10/2010 -l

Y con esto obtendríamos un listado de paquetes con el siguiente formato:
fecha completa en formato UTC >>> categoría/nombre_del_paquete-versión
Por ejemplo:

Fri Jan 8 00:26:52 2010 >>> dev-libs/libnl-1.1-r2

La captura del tamaño de los paquetes es bastante menos complicada, gracias a las herramientas de línea de comando y los sistemas de Unix, basta con ubicar la carpeta donde se almacenan los archivos y ejecutar un comando ls para obtener un listado del contendido de la carpeta que junto a las opciones correctas, puede mostrar toda la información referente al paquete.

Cuando Gentoo descarga el código fuente de un paquete para ser instalado, se almacena dentro de la carpeta de sistema /usr/portage/distfiles. Asi que los pasos para obtener el tamaño de cada uno de los paquetes son:

a) Eliminar todos los paquetes que contiene /usr/portage/distfiles

b) Descargar los paquetes que corresponden al experimento

c) Ejecutar el comando ls para que despliegue el nombre de los archivos y su tamaño de la forma

categoría/nombre_del_paquete tamaño

Para obtener los datos necesarios se utilizó el siguiente comando:

ls -Rgh

Con esto se obtendrá:

- R - El listado de archivos dentro de la carpeta y las subcarpetas si es que existieran
- h – mostrar el tamaño de los paquetes en un formato de fácil comprensión humana
- g – mostrar todas las opciones del paquete sin incluir el grupo

6.3 . Captura del uso promedio de memoria y del CPU

Ya que estos datos únicamente pueden ser capturados por medio de promedios, es necesario capturarlos periódicamente y luego calcular estos promedios con + base en la información capturada en los periodos de tiempo delimitados por el inicio y finalización de la compilación del paquete, los cuales ya obtuvimos con genlop.

Estos datos deben registrarse con una tarea que se ejecute en periodos fijos de tiempo, un procedimiento genérico para recolectar estos datos seria; escribir una serie de *scripts* que almacenen la información actual de uso de memoria, procesador, y cualquier variable de la cual quisiéramos tener información en ese periodo de tiempo; sin embargo, en UNIX muchas de estas cosas son posibles mediante el comando sar y su versión en demonio del sistema sard.

Sar es un monitor de actividades del sistema, que se encuentra usualmente en la mayoría de sistemas UNIX, cuyo objetivo es mostrar información del estado actual del sistema, la característica más interesante de Sar es exportar todos los datos recolectados mientras se monitorea el uso de recursos del sistema hacia archivos de datos, lo cual hace posible retener los datos de uso y analizarlos posteriormente[22].

22 Sar manpage disponible con la ejecucion del comando “man sar” en cualquier sistema Unix con sar instalado.

Aunque en UNIX se pueden programar tareas para ejecutarse en periodos y momentos específicos de tiempo mediantes demonios cron, éstos presentan una limitante, no pueden ejecutar tareas en periodos de tiempo menores a un minuto. Lo cual hizo que automáticamente se descartara esta posibilidad.

Sin embargo, los demonios cron, a la larga, son sólo procesos que se ejecutan cada 30 segundos y que ven en sus registros si tienen tareas para ejecutar y este comportamiento es fácilmente replicable, mediante un simple *script* en bash mediante la instrucción "*sleep*".

Por ejemplo, un simple *script* en bash para imprimir la cadena de texto "soy un *script* en bash" en un intervalo de 30 segundos se reduce a las siguientes tres simples líneas:

```
#!/bin/bash
echo "Soy un script en bash"
sleep 30
```

Asi pues, si combinamos la instruccion *sleep* con la recolección de datos de sar y lo ejecutamos como un proceso en segundo plano, tendremos nuestra herramienta para la recolección de datos mediante simples herramientas disponibles en cualquier sistema UNIX, para poner en funcionamiento nuestro *script* basta con ejecutarlo desde cualquier terminal con el símbolo & en bash que indica al intérprete de comandos cuando un proceso debe ejecutarse en segundo plano.

Por ejemplo, cuando ejecutamos nuestro script, cuyo nombre es script_recolector_datos.sh obtenemos lo siguiente:

```
# script_recolector_datos.sh&
[1] 19398
```

Este procedimiento se conoce cómo hacer un *fork* al proceso original, ya que el proceso que ejecutamos en lugar de ejecutarse se copió hacia el segundo plano y continuó su ejecución en esta área.

7 . ANÁLISIS DE DATOS

7.1 . Descripción de los casos de estudio

7.1.1 . Prueba 1 – Distcc

La primera prueba que se realizó fue sobre un *cluster* configurado con el *software* Distcc sin ningún tipo de configuración adicional.

7.1.2 . Prueba 2 – Distcc con ccache y *pump*

La segunda prueba también se realizó mediante la configuración de un *cluster* con el *software* Distcc, pero esta vez, con dos tipos de optimizaciones, ccache, que utiliza las cabeceras pre procesadas comunes de compilaciones previas y modo *pump* que, además de enviar código fuente a compilar, envía los archivos de preconfiguración para que sean procesados por los nodos del *cluster*.

7.1.3 . Prueba 3 – *Icecream*

La última prueba se realizó con una herramienta totalmente distinta, *Icecream*, proyecto patrocinado por Novell que comparte una base de código común con Distcc. Los autores de *Icecream* argumentan, que sin ninguna configuración adicional, *Icecream* es un serio competidor frente a Distcc.

7.2 . Enfoques del análisis

Según la Universidad de York (Canadá), la teoría de la complejidad computacional trata de analizar, definir y sobre todo predecir los recursos que serán utilizados a la hora de resolver un problema computacional. En el caso del presente experimento la producción de paquetes en un entorno de alta eficiencia[23].

Para analizar esta tarea debemos analizar los recursos que se utilizan durante la compilación de paquetes, como se mencionaba antes, los recursos más importantes en la creación del paquete fueron: tiempo de duración de la compilación, porcentaje de memoria utilizada en la compilación y tamaño del paquete a compilar.

Una vez teniendo los datos se procedió finalmente a analizarlos mediante *software* estadístico, en nuestro caso mediante el *Software* Paquete Estadístico para las Ciencias Sociales o *SPSS* por sus siglas en inglés. La principal característica por la cual se utilizó *SPSS* es por sus múltiples puntos de vista sobre una matriz de información.

23 York Complexity Theory-
http://www.fsc.yorku.ca/york/istheory/wiki/index.php/Complexity_theory

7.3 . Análisis descriptivo con base en categorías

En estudios estadísticos, como es el caso de este trabajo, siempre encontramos dos tipos de variables; variables continuas y variables categóricas, la diferencia entre una y otra es, que las variables continuas son valores puntuales obtenidos a través de la muestra de casos de estudios y las variables categóricas son variables cuyos valores se determinan por la ubicación del valor puntal, dentro de un rango o la equivalencia de un valor categórico hacia un valor numérico fijo.

Para utilizar este tipo de análisis dentro del presente estudio fue necesaria la elaboración de una tabla de códigos

La idea detrás de esta tabla de códigos es simple: establecer una escala numérica para rangos de valores y/o valores que representen a variables no numéricas.

La escala definida para este experimento fue la siguiente:

Tabla VI. **Tabla de códigos para el análisis estadístico mediante SPSS**

Variable	Descripción	Nombre variable	Instrucciones de codificación
Memoria al compilar un paquete	Cantidad de memoria utilizada en la creación del paquete (Porcentaje promedio entre todos los nodos)	Mem	1= [0-20%) 2= [20-40%) 3= [40-60%) 4= [60-80%) 5= [80%-100%]
Tiempo	Cantidad de tiempo que duro la tarea	Tiempo	1=0-5 minutos 2=5-10 minutos 3=10-15 minutos 4=15-20 minutos 5=20 en adelante
Tamaño	Cantidad de Megabytes que consistencia el paquete	Tam	1= 00-5 Megabytes 2= 5-10 Megabytes 3= 10-15 Megabytes 4= 15-20 Megabytes 5= 20 Megabytes en adelante
Procesador	Porcentaje de utilización promedio para los nodos del *cluster*	Proc	1= [0-20%) 2= [20-40%) 3= [40-60%) 4= [60-80%) 5= [80%-100%]
Herramienta de configuración	Herramienta utilizada para configuración del paquete	Herramienta	1=Automake 2=Cmake 3=No aplica / instalación de *script*

Fuente: elaboración propia

Mediante esta tabla de códigos y el *software* estadístico *SPSS* se pudieron obtener los siguientes datos y realizar distintos análisis preeliminares, para la conclusión de las preguntas de estudio.

7.3.1 . Descriptivos generales

Inicialmente, se obtuvo la estadística descriptiva básica con base en los tres casos de estudio, en los cuales ya se puede evidenciar las diferencias entre uso de procesador, memoria y de CPU. Los resultados se presentan en las siguientes tablas:

Tabla VII. **Estadísticos descriptivos prueba 1 – Distcc original**

Estadísticos descriptivos

	N	Mínimo	Máximo	Suma	Media	Desv. típ.
Tam	372	1	5	475	1,28	,835
Tiempo	372	1	5	422	1,13	,548
Proc	372	2	5	1030	2,77	,801
Mem	372	1	3	750	2,02	,381
N válido (según lista)	372					

Fuente: elaboración propia

Tabla VIII. **Estadísticos descriptivos prueba 2 – Distcc con ccache y modo pump**

Estadísticos descriptivos

	N	Mínimo	Máximo	Suma	Media	Desv. típ.
Tam	372	1	5	475	1,28	,835
Tiempo	372	1	5	431	1,16	,622
Proc	372	1	5	964	2,59	,843
Mem	372	1	3	793	2,13	,483
N válido (según lista)	372					

Fuente: elaboración propia

Tabla IX. **Estadísticos descriptivos prueba 3 – *Icecream***

Estadísticos descriptivos

	N	Mínimo	Máximo	Suma	Media	Desv. típ.
Tam	372	1	5	475	1,28	,835
Tiempo	372	1	5	420	1,13	,539
Proc	372	2	5	1072	2,88	,799
Mem	372	1	3	880	2,37	,504
N válido (según lista)	372					

Fuente: elaboración propia

Se hace evidente que el tamaño de los paquetes es una constante debido a que en las tres pruebas se produjeron los mismos paquetes ejecutables. Sin embargo, de esta información podemos obtener varias conclusiones.

- Contrario a lo esperado y lo dicho por los creadores del proyecto Distcc, al activar las optimizaciones en el *cluster* conseguimos el resultado contrario, ya que el tiempo de producción de paquetes aumentó y el poder de procesamiento fue mejor aprovechado por el *cluster*, sin las optimizaciones, obteniendo un valor mayor, con lo cual podemos concluir que Distcc, sin optimizaciones realiza un mejor trabajo.

- Respecto a *Icecream*, también observamos que fue el que mejor tiempo obtuvo de los tres, y además de eso con un consumo más alto de procesador, lo que se traduce en una eficiencia mayor en las tareas de compilación, y se ubica como la solución de *software* más eficiente para la configuración y compilación de paquetes de distribuciones GNU/Linux, aprovechando de mejor manera los recursos disponibles en los nodos.

7.3.2 . Influencia de la herramienta de configuración

Ahora que ya sabemos cuál herramienta realiza, de una manera más eficiente su trabajo, también es importante analizar qué herramienta es la más utilizada y con cuál se consiguen mejores tiempos de compilación.

Tabla X. **Resumen de procesamiento de datos para las tres pruebas efectuadas**

Resumen de procesamiento de datos

		Recuento	Porcentaje
Herramienta	automake	231	62,1%
	cmake	137	36,8%
	script	4	1,1%
Global		372	100,0%
Excluido		0	
Total		372	

Fuente: elaboración propia

Tabla XI. **Relación Tamaño/Tiempo por herramienta de configuración prueba 1**

Estadísticos de la razón para Tam / Tiempo

Grupo	Diferencial	Coeficiente de dispersión	Coeficiente de variación
			Mediana centrada
automake	1,032	,185	61,0%
cmake	1,027	,213	77,5%
script	1,000	,000	,0%
Global	1,030	,193	67,2%

Fuente: elaboración propia

Tabla XII. **Relación Tamaño/Tiempo por herramienta de configuración prueba 2**

Estadísticos de la razón para Tam / Tiempo

Grupo	Diferencial	Coeficiente de dispersión	Coeficiente de variación
			Mediana centrada
automake	1,050	,182	60,5%
cmake	1,031	,222	79,1%
scripts	1,000	,000	,0%
Global	1,044	,194	67,5%

Fuente: elaboración propia

Tabla XIII. **Relación Tamaño/Tiempo por herramienta de configuración prueba 3**

Estadísticos de la razón para Tam / Tiempo

Grupo	Diferencial	Coeficiente de dispersión	Coeficiente de variación
			Mediana centrada
automake	1,024	,188	61,4%
cmake	1,035	,212	77,4%
script	1,000	,000	,0%
Global	1,028	,195	67,3%

Fuente: elaboración propia

A partir de los datos anteriores podemos hacer las siguientes conclusiones:

- Automake, históricamente, ha sido la herramienta predilecta para configuración de paquetes y esto se ve evidenciado en su porcentaje de utilización frente a cmake.

- Si observamos los datos de diferencial, en ambas pruebas podemos ver que en el caso de las pruebas con Distcc automake tiene una mejor relación tamaño/tiempo, o dicho de otra manera se procesa más información en menor tiempo; en cambio en *Icecream* cmake presenta una mejor relación tamaño tiempo y debería ser la primera elección a la hora de producir paquetes que utilizan cmake (paquetes del proyecto kde generalmente).

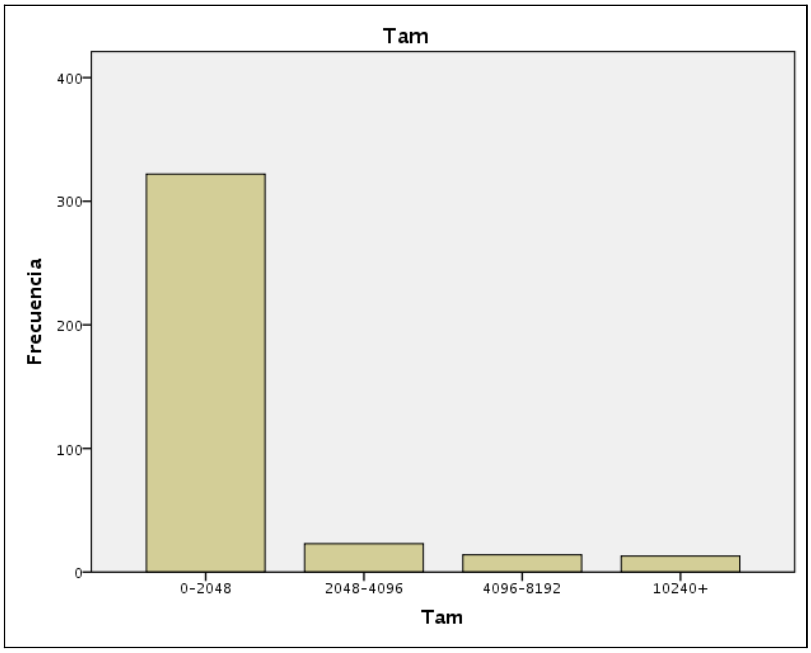
- Recordemos que las relaciones y datos están con base en rangos, así pues los paquetes marcados como *script* siempre va a dar uno porque no son paquetes grandes y ni siquiera son compilados, por lo tanto, su producción fue de corta duración y, se produjeron dentro de los primeros 10 minutos.

7.3.3 . Tendencias

Para la fase final del experimento se analizaron algunas tendencias en el comportamiento de los datos, mediante un análisis de frecuencias para cada una de las variables, en los tres experimentos. Los resultados del análisis y sus correspondientes gráficas se presentan a continuación:

7.3.3.1 Tendencias generales

Figura 3. **Tamaño de los paquetes en las tres pruebas efectuadas**



Fuente: elaboración propia

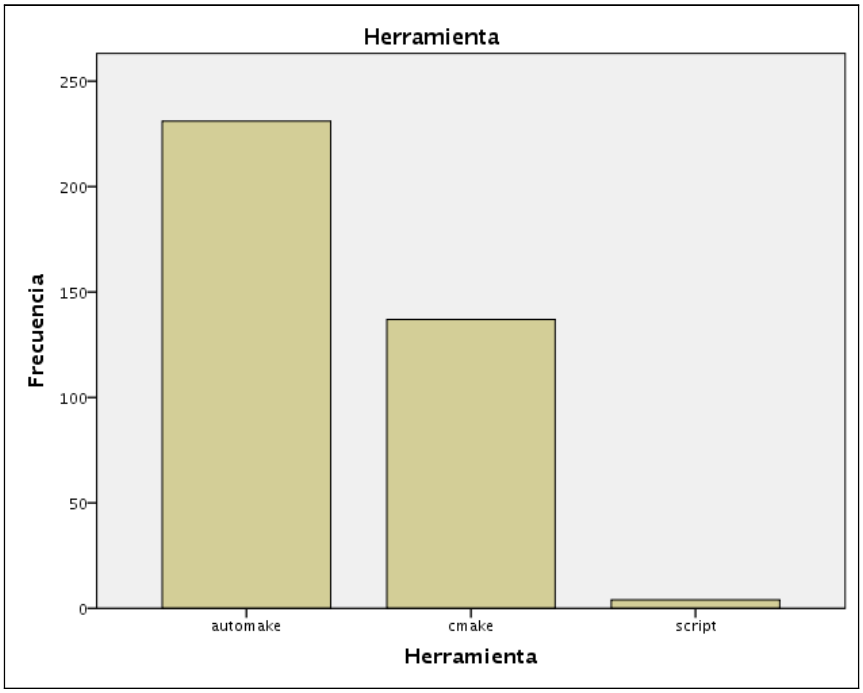
Tabla XIV. **Tamaño de los paquetes de *software* durante las 3 pruebas**

Tam

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	0-2048	322	86,6	86,6	86,6
	2048-4096	23	6,2	6,2	92,7
	4096-8192	14	3,8	3,8	96,5
	10240+	13	3,5	3,5	100,0
	Total	372	100,0	100,0	

Fuente: elaboración propia

Figura 4. **Gráfica de la frecuencia de las herramientas de configuración utilizadas en las tres pruebas**



Fuente: elaboración propia

Tabla XV. **Herramientas de configuración utilizadas en las 3 pruebas**

Herramienta

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	automake	231	62,1	62,1	62,1
	cmake	137	36,8	36,8	98,9
	script	4	1,1	1,1	100,0
	Total	372	100,0	100,0	

Fuente: elaboración propia

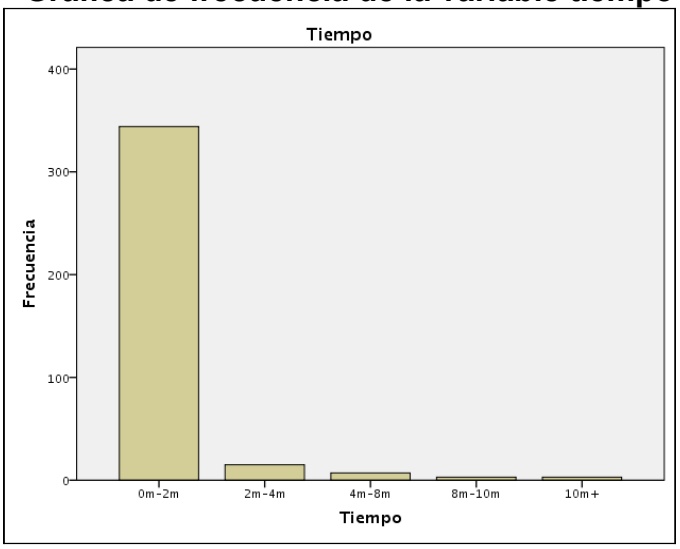
Las herramientas de configuración y el tamaño de los paquetes son datos que no cambian en el set de pruebas y, como ya se mencionaba antes, la herramienta con mayor uso fue automake.

Respecto al tamaño de los paquetes, aunque es un número considerablemente grande, se evidenció que la mayoría de paquetes no supera los 2 *Megabytes*, esto derivado del hecho que el desarrollo colaborativo de distribuciones GNU/Linux se hace a través de pequeños desarrollos de distintos proyectos que en su conjunto forman el sistema operativo que conocemos.

7.3.3.2 Tendencias prueba 1

Las tendencias de comportamiento durante la prueba 1 fueron las siguientes:

Figura 5. **Gráfica de frecuencia de la variable tiempo en prueba 1**



Fuente: elaboración propia

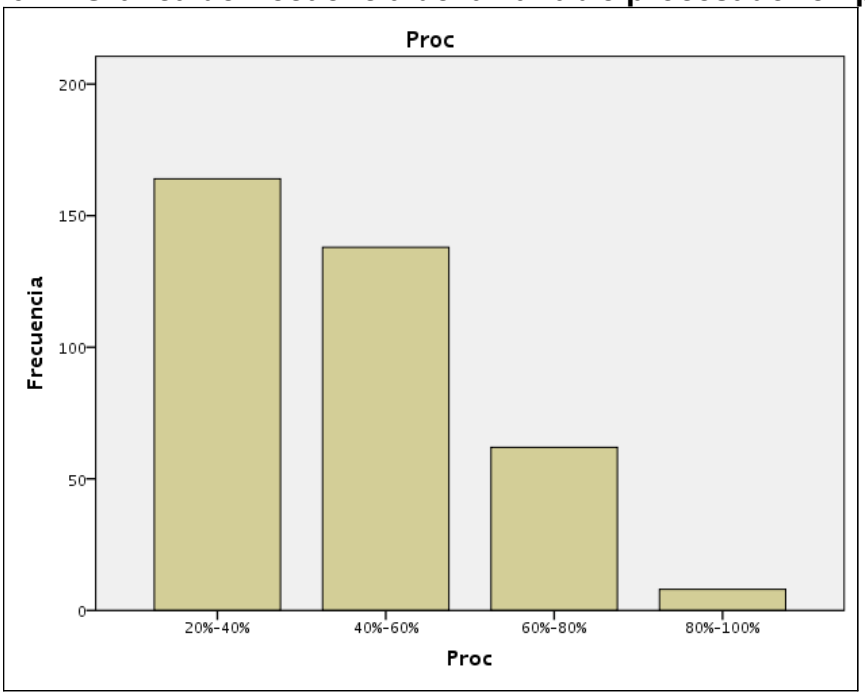
Tabla XVI. **Estadística descriptiva variable tiempo, prueba 1**

Tiempo

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	0m-2m	344	92,5	92,5	92,5
	2m-4m	15	4,0	4,0	96,5
	4m-8m	7	1,9	1,9	98,4
	8m-10m	3	,8	,8	99,2
	10m+	3	,8	,8	100,0
	Total	372	100,0	100,0	

Fuente: elaboración propia

Figura 6. **Gráfica de frecuencia de la variable procesador en prueba 1**



Fuente: elaboración propia

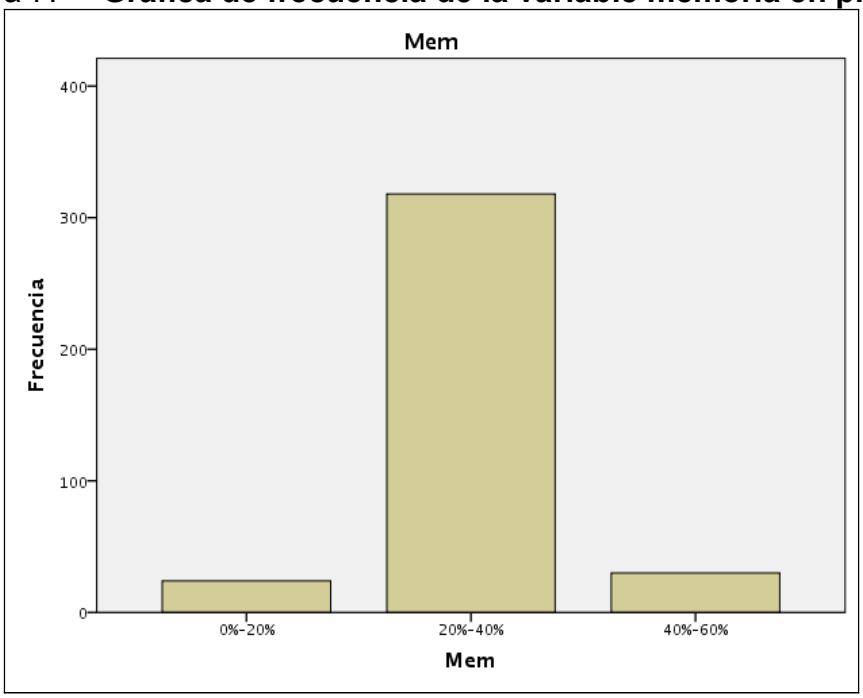
Tabla XVII. **Estadística descriptiva variable procesador, prueba 1**

Proc

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	20%-40%	164	44,1	44,1	44,1
	40%-60%	138	37,1	37,1	81,2
	60%-80%	62	16,7	16,7	97,8
	80%-100%	8	2,2	2,2	100,0
	Total	372	100,0	100,0	

Fuente: elaboración propia

Figura 7. **Gráfica de frecuencia de la variable memoria en prueba 1**



Fuente: elaboración propia

Tabla XVIII. **Estadística descriptiva variable memoria prueba 3**

Mem

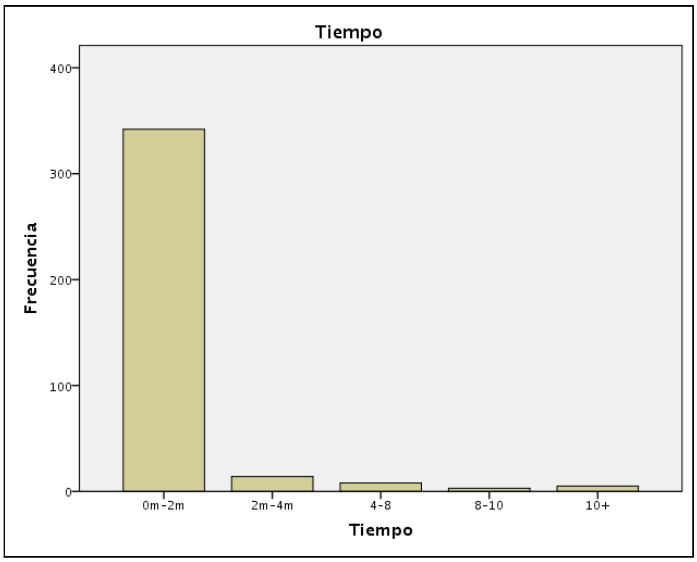
		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	0%-20%	24	6,5	6,5	6,5
	20%-40%	318	85,5	85,5	91,9
	40%-60%	30	8,1	8,1	100,0
	Total	372	100,0	100,0	

Fuente: elaboración propia

7.3.3.3 Tendencias Prueba 2

Las tendencias de comportamiento durante la prueba dos fueron las siguientes:

Figura 8. **Gráfica de frecuencia de la variable tiempo en prueba 2**



Fuente: elaboración propia

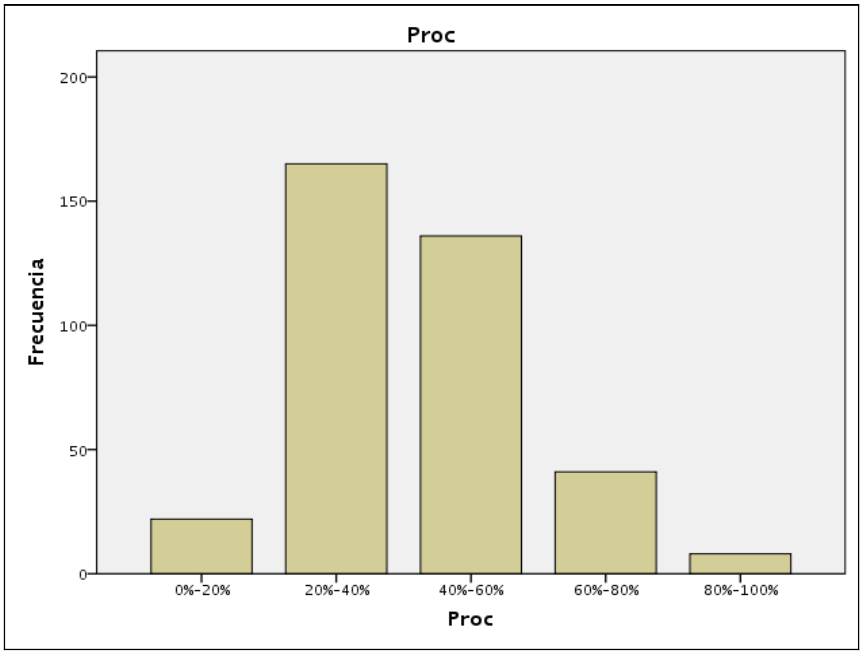
Tabla XIX. **Estadística descriptiva variable tiempo prueba 2**

Tiempo

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	0m-2m	342	91,9	91,9	91,9
	2m-4m	14	3,8	3,8	95,7
	4-8	8	2,2	2,2	97,8
	8-10	3	,8	,8	98,7
	10+	5	1,3	1,3	100,0
	Total	372	100,0	100,0	

Fuente: elaboración propia

Figura 9. **Gráfica de frecuencia de la variable procesador en prueba 2**



Fuente: elaboración propia

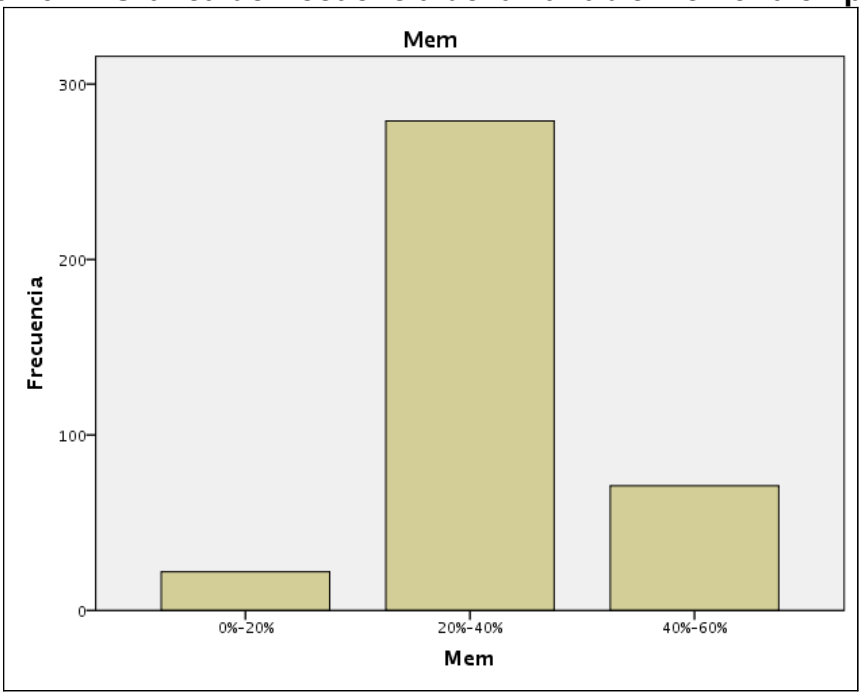
Tabla XX. **Estadística descriptiva variable procesador prueba 2**

Proc

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	0%-20%	22	5,9	5,9	5,9
	20%-40%	165	44,4	44,4	50,3
	40%-60%	136	36,6	36,6	86,8
	60%-80%	41	11,0	11,0	97,8
	80%-100%	8	2,2	2,2	100,0
	Total	372	100,0	100,0	

Fuente: elaboración propia

Figura 10. **Gráfica de frecuencia de la variable memoria en prueba 2**



Fuente: elaboración propia

Tabla XXI. **Estadística descriptiva variable memoria prueba 2**

Mem

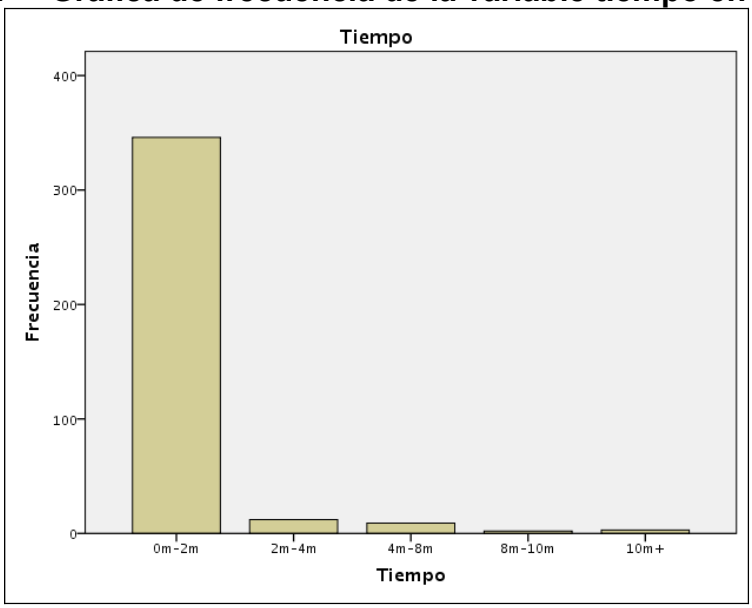
		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	0%-20%	22	5,9	5,9	5,9
	20%-40%	279	75,0	75,0	80,9
	40%-60%	71	19,1	19,1	100,0
	Total	372	100,0	100,0	

Fuente: elaboración propia

7.3.3.4 Tendencias prueba 3

Las tendencias de comportamiento durante la prueba dos fueron las siguientes:

Figura 11. **Gráfica de frecuencia de la variable tiempo en prueba 3**



Fuente: elaboración propia

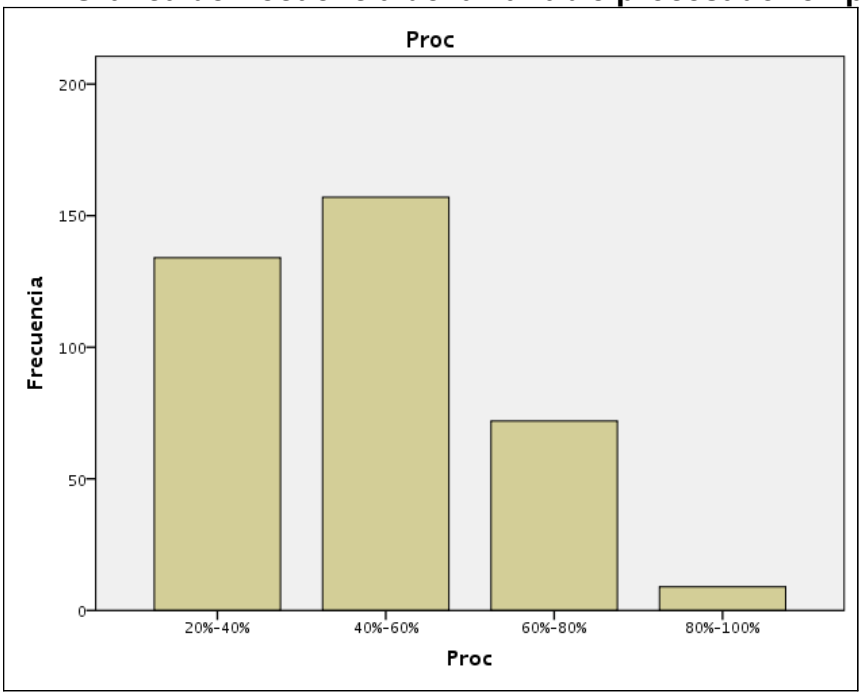
Tabla XXII. **Estadística descriptiva variable tiempo prueba 3**

Tiempo

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	0m-2m	346	93,0	93,0	93,0
	2m-4m	12	3,2	3,2	96,2
	4m-8m	9	2,4	2,4	98,7
	8m-10m	2	,5	,5	99,2
	10m+	3	,8	,8	100,0
	Total	372	100,0	100,0	

Fuente: elaboración propia

Figura 12. **Gráfica de frecuencia de la variable procesador en prueba 3**



Fuente: elaboración propia

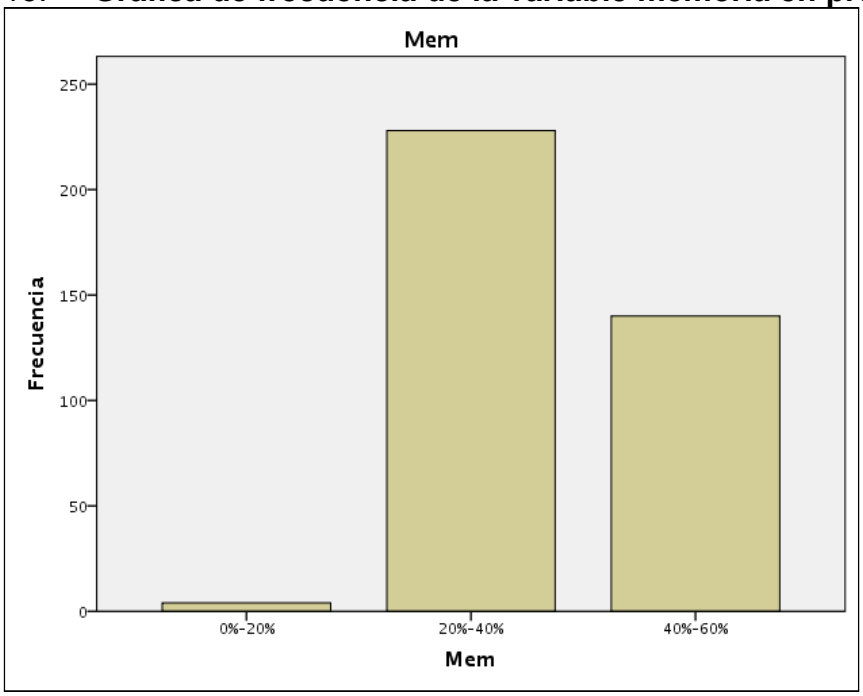
Tabla XXIII. **Estadística descriptiva variable procesador prueba 3**

Proc

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	20%-40%	134	36,0	36,0	36,0
	40%-60%	157	42,2	42,2	78,2
	60%-80%	72	19,4	19,4	97,6
	80%-100%	9	2,4	2,4	100,0
	Total	372	100,0	100,0	

Fuente: elaboración propia

Figura 13. **Gráfica de frecuencia de la variable memoria en prueba 3**



Fuente: elaboración propia

Tabla XXIV. **Estadística descriptiva variable memoria prueba 3**

Mem

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	0%-20%	4	1,1	1,1	1,1
	20%-40%	228	61,3	61,3	62,4
	40%-60%	140	37,6	37,6	100,0
	Total	372	100,0	100,0	

Fuente: elaboración propia

7.3.3.5 Consideraciones acerca de las tendencias de las pruebas

En las tres pruebas se evidenció que los paquetes se compilaron en un tiempo menor a dos minutos; sin embargo, los datos más importantes respecto a una prueba frente a otra es la relación que existe entre el aprovechamiento de los recursos del *cluster* y el tiempo.

Las tres pruebas se efectuaron en las mismas computadoras, y como se observa en las gráficas y en las tablas, *Icecream* fue la solución que mejor utilizó los recursos, marcando mejores punteos en porcentaje de utilización de procesador y memoria (donde más es mejor).

CONCLUSIONES

1. Existe una tendencia a producir con mayor eficiencia, ejecutables configurados con automake mediante el uso de Distcc, esto se demostró desde el punto de vista estadístico con una mejor relación tamaño/tiempo; en ambas pruebas con Distcc (prueba 1 automake - 1 032 cmake - 1 027, prueba 2 automake – 1 050 cmake 1 031, prueba 3 automake-1 024 cmake-1 035); sin embargo, si nuestro experimento cuenta con demasiados paquetes configurables vía cmake, la mejor elección es *Icecream*.

2. El análisis se realizó a nivel escalar, la mejor opción a nivel de tiempo de producción de paquetes fue *Icecream*, con un punteo de 420 (sumatoria de punteos escalares) frente a 422 de Distcc y 431 de Distcc con optimizaciones siendo el ganador *Icecream*.

3. Además de la diferencia en rendimiento, *Icecream* tiene otra ventaja frente a Distcc. La opción a soporte técnico comercial de parte de Novell. Porque, aunque la diferencia de rendimiento se hizo evidente, es demasiado pequeña como para descartar Distcc como opción. La elección final de la herramienta debe involucrar factores administrativos, como el nivel de conocimiento del personal de TI para la configuración de este tipo de herramientas.

4. Aunque *Icecream* superó en todas las pruebas a Distcc, los promedios de ambas pruebas estuvieron dentro de la misma escala, tanto para procesador (2,77, 2,59, 2,88) como para memoria (2,02, 2,13, 2,37) lo que estadísticamente refleja que el cambio no es significativo con una diferencia no mayor de 0,3 en todas las pruebas. Así que, podemos afirmar que ambos proyectos han mantenido la calidad en su código y eso se refleja en los resultados de las pruebas bastante similares.

5. Quedó demostrado que el uso de Ccache impactó negativamente en la producción de paquetes en los *clusters*; además de esto, hay que tomar en cuenta que Ccache sólo es factible para su uso en entornos de producción de paquetes y no en entornos de desarrollo, donde su efecto será mínimo por el constante cambio en el código fuente.

RECOMENDACIONES

1. A menos que sea una limitante en el entorno de desarrollo, se debe preferir el uso de *Icecream* en el entorno de programación, por la eficiencia y velocidad demostrada a la hora de producir un paquete distribuido.

2. Dado que las dos herramientas de configuración mayormente utilizadas por los paquetes del experimento fueron automake y cmake, se debe tratar de diseñar el paquete para que sea configurado con automake, con la salvedad que la mayoría de paquetes escritos en lenguaje C++ dependientes de las bibliotecas de funciones QT deben ser configurados utilizando cmake.

3. Aunque las optimizaciones de código y los entornos de compilación distribuida están más que preparados para la compilación de paquetes, este tipo de herramientas deben ser utilizadas por usuarios avanzados con necesidades específicas de rendimiento. Ya que el proceso de generación de un sistema operativo básico se tardó alrededor de 12 horas sin tomar en cuenta la configuración, mientras que un usuario final puede conseguir un disco de 700Mb listo para su instalación en un tiempo promedio de 2 horas, con una conexión residencial básica.

BIBLIOGRAFÍA

1. GOUGH, Brian J. *An Introduction to* GCC. Stallman, Richard (revisor). Estados Unidos: Network Theory Ltd, 2005. 144 p. ISBN: 978 0954161798.

2. IBERICO HIDALGO, Martín Alberto. “Administrador de Proyectos de Grid Computing que Hacen Uso de la Capacidad de Cómputo Ociosa de Laboratorios Informáticos”. Director: Leopoldo Genghis Rios Kruger. Universidad Católica del Perú, Dirección de Informática Académica, 2009.

3. MECKLENBURG, Robert. *Managing Projects with GNU Make.* 3a. ed. Estados Unidos: O'Reilly Media, 2004. 304 p. ISBN: 978-0-596-00610-5.

4. RUI, Xu. *Clustering.* Estados Unidos: Wiley & Sons IEEE Press, 2008. 358 p. ISBN: 978-0-470-27680-8.

5. SLOAN, Josep D. *High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI.* Estados Unidos: O'Reilly Media, 2004. 368 p. ISBN: 978-0596005702.

6. TROYER, Matthias. *SuSE Linux Cluster Solutions, Experience with the Asgard Cluster at ETH Zürich* [en línea]. Department of Physics, ETH Zürich. [ref. de 12 de diciembre de 2010] Disponible en *web.* <http://www.asgard.ethz.ch/whitepapers/troyer00a.pdf>.