# Análisis de Sistemas Embebidos HPC usando Manejadores de Paquetes

**Carlos Eduardo Gómez Hernández**

**Trabajo de Grado para Optar el Título de Maestría en Ingeniería de Sistemas e Informática**

**Director**

**Carlos Jaime Barrios Hernández**

**Ph.D en Ciencias de la Computación**

**Codirector**

**Olivier Richard**

**Ph.D en Ciencias de la Computación**

**Universidad Industrial de Santander**

**Facultad de Ingenierías Fisicomecánicas**

**Escuela de Ingeniería de Sistemas e Informática**

**Bucaramanga**

**2020**

## Dedicatoria

Le dedico este trabajo de grado a mis padres y hermana por estar siempre presentes y apoyarme de cualquier manera sin importar los problemas que hubiera tenido a lo largo de este proyecto y estar siempre pendientes de que siguiera adelante y lograra esta meta.

También a mis compañeros que siempre estuvieron presentes en este proceso y me ayudaron con ideas y propuestas para hacer más fácil el proceso de desarrollo de este proyecto.

A los profesores que me apoyaron en el proceso del desarrollo de este proyecto, así como la gente en Grenoble que me apoyó para que el proyecto fuera un éxito.

## Agradecimientos

Agradezco al profesor Carlos Jaime Barrios Hernández por su apoyo y preocupación en todo el proceso del desarrollo del proyecto.

Agradezco al profesor Olivier Richard por su apoyo y aceptación en el uso de los implementos del laboratorio DataMove de Inria en Grenoble, Francia, así como en el proceso de poder pasar con ellos un tiempo en el proceso de investigación en la misma ciudad.

De igual forma agradezco al equipo de DATAMove INRIA/LIG de Grenoble, Francia y a la colaboración CATAI Francia-Colombia en Cómputo y Tecnologías Avanzadas para la Sostenibilidad por el apoyo a este proyecto.

## Contenido

**Pag.**

## Lista de Tablas

**Pag.**

## Lista de Figuras

**Pag.**

## Lista de Apéndices

**Pag.**

## Resumen

**Título:** Análisis de Sistemas Embebidos HPC usando Manejadores de Paquetes[1]

**Autor:** Carlos Eduardo Gómez Hernández[2]

**Palabras Clave:** Sistemas Embebidos, Manejadores de Paquetes, Consumo Energético, Benchmark

**Descripción:**

El proyecto consiste en analizar las capacidades computaciones y energéticas para computación de alto rendimiento (HPC por sus siglas en inglés) de ciertos sistemas embebidos usando una herramienta de gestión de aplicaciones de tipo manejadores de paquetes. Para lograr este objetivo, se escogió un tipo de sistemas embebidos y una clase de Manejador de Paquetes. Usando una serie de algoritmos denominados benchmarks, se logra obtener resultados de rendimiento computacional del sistema. Finalmente, una vez obtenidos estos resultados se usan estos valores para obtener el consumo energético de la infraestructura.

Principalmente este proyecto se divide en dos partes: uso de software (por los manejadores de paquetes) y uso de hardware (por los sistemas embebidos). Por esta razón, se exploran por separado ambas partes, estudiando sus componentes y algunos elementos pertenecientes a esa parte y además se mencionan algunos proyectos con una configuración similar para mostrar un precedente de lo que puede hacer este tipo de infraestructura.

En la etapa final, se crea un modelo de una infraestructura, se implementa, se agregan las herramientas necesarias y con el uso de un sistema de algoritmos de prueba computacional denominados benchmark, se observa su rendimiento computacional. Sin embargo, las pruebas deben también indicar consumo energético, por lo que basados en un estudio previo, se usan esos resultados adaptados al proyecto para aproximar el valor del consumo energético en nuestro sistema. Todo lo anterior se gráfica y se analiza el rendimiento e impacto de los manejadores de paquetes dentro de los sistemas embebidos.

[1] Trabajo de Grado de Maestría

[2] Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingeniería de Sistemas e Informática. Director Carlos Jaime Barrios. Codirector Olivier Richard

## Abstract

**Title:** Analysis of Embedded HPC Systems using Package Manager[3]

**Author:** Carlos Eduardo Gómez Hernández[4]

**Keywords:** Embedded Systems, Package Manager, Energy Consumption, Benchmark

**Description:**

This project consists on analyse the computing and energy capabilities for High Performance Computing (HPC) of embedded systems using a package manager tool: a type of application management software. To achieve this objective, we chose one type of embedded systems and one type of package manager. Using a series of algorithms called Benchmarks, we obtained computational efficiency results of the system. Finally, once we got these results we used them in order to acquire the energy consumption values of the infrastructure.

In general, this project has two parts: first the choose of the software (the package manager) and second the hardware use (the embedded systems). We explored separate each part, analysing and studying the components and some elements that belonged to that part. We also referred to similar projects with similar infrastructure configuration, showing a precedent of the general attributes of these kind of systems.

On the final stage, we created a infrastructure model, we implemented and incorporated the necessary tools for this system and using some algorithms of computational test called benchmark, we obtained its computational performance. Nevertheless, the tests also have to indicate energy consumption, so based on a previous studies we adapted the author formulas to his project to approximate the system energy consumption. Finally, we make graph of the results and analyse the impact of the package manager inside the embedded systems.


[3] Master Thesis

[4] Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingeniería de Sistemas e Informática. Director Carlos Jaime Barrios. Codirector Olivier Richard

## Introducción

Hoy en día para resolver los problemas en distintas áreas del conocimiento científico se requieren altas capacidades de cómputo. Por tal razón, en los últimos años se han estado desarrollado sistemas de cómputo que puedan suplir esta necesidad. En muchos de sus desarrollos se ideó usar las tarjetas gráficas para realizar este tipo de operaciones dado la naturaleza que éstas poseen en sus cálculos. De esta manera se han desarrollado arquitecturas embebidas con estas tarjetas que poseen mejores capacidades que un nodo de un supercomputador clásico.

No obstante, el manejo de dichas arquitecturas supone un nuevo reto de investigación. La gestión de dichos elementos para explotar al máximo sus capacidades sin que se pierda el desempeño general del sistema han creado una línea de investigación específica para estos modelos. Los modelos que se investigan varían desde el hardware usado hasta el software en el que el algoritmo es ejecutado. El proyecto propuesto abarca el análisis de dichos sistemas utilizando herramientas modernas para la ejecución de algoritmos que permiten un mayor control por parte del usuario el uso de dichos recursos.

El presente proyecto tiene como referencia dos áreas diferentes de la computación pero que en el marco de las tecnologías modernas se han fusionado: analizar una infraestructura para HPC (High Performance Computing) usando hardware y software no tradicional.

Con respecto al hardware, se encuentra unas arquitecturas llamadas sistemas embebidos, que, gracias a su bajo consumo energético, resultan ideales para los nuevos retos de los sistemas HPC. Por la parte del software, se desea analizar el uso y la viabilidad de una herramienta para el control y encapsulamiento de aplicaciones llamadas manejadores de paquetes.

El auge en la utilización de los recursos de un supercomputador para realizar cualquier tarea de simulación o cálculo desde cualquier área, ha convertido la supercomputación en un área

que se debe analizar con cuidado para el mejor aprovechamiento de recursos y administración. Hoy en día es más fácil acceder a estos recursos desde cualquier parte del mundo y desde cualquier dispositivo y ya grandes empresas ofrecen estos servicios en la nube para que cualquier persona pueda usarlos.

Por lo anterior se vuelve necesario reducir costos en estos sistemas para poder competir en el mercado y además ofrecer mejor rendimiento en el sistema. Por esta razón, uno de los aspectos para mejorar es el consumo energético, que, a la fecha, es más costoso al final que la compra del mismo sistema al término de varios años.

Por otro lado, desde otra perspectiva, los constantes desarrollos de las GPUs las han convertido en un buen elemento para realizar cálculos computacionales con mejor rendimiento energético. Creados para realizar las operaciones de los elementos de video, las GPU fueron mejorados para poder realizar operaciones incluso de propósito general (GPGPU), y gracias al desarrollo de nuevas librerías para el manejo de estas tarjetas, se extendió su uso a todo sistema que requiriera de paralelismo.

Así pues, es como nace los sistemas embebidos a los que se refiere este proyecto: arquitecturas complejas que involucran procesadores con eficiencia energética y además una gran cantidad de procesadores de tipo GPU para procesamiento en paralelo.

Y finalmente pero no menos importante, las nuevas herramientas desarrolladas para poder administrar estos sistemas para HPC han puesto una nueva manera de organizar y administrar las aplicaciones evitando en lo posible los problemas de compilación, instalación e incompatibilidad entre aplicaciones y son sobre una de estas herramientas que va a ser objeto de análisis en esta propuesta.

En resumen, la idea general estará enfocada en analizar una arquitectura de sistemas embebidos usando una de estas herramientas de administración de aplicaciones, observando su comportamiento y midiendo rendimiento, tanto en la parte computacional como energética.

A lo largo de este texto se mencionará las distintas fases que tuvo lugar el proyecto para llegar a ese análisis del sistema. En el capítulo 1 se mencionarán los distintos modelos en el que se resuelve el problema de la incompatibilidad de aplicaciones, así como proyectos similares que involucran estas herramientas o arquitectura. En el capítulo 2 se explorarán más a fondo los manejadores de paquetes y el porqué de nuestra elección. En el capítulo 3 abordaremos algunos sistemas embebidos, aquellos sistemas que se analizaron al inicio de esta propuesta, con sus aspectos a favor y en contra. En el capítulo 4 se abordará los *benchmarks* usados en el análisis de sistemas incluyendo sus problemas al instalarlo. En el capítulo 5 se mostrarán los resultados de las pruebas realizadas y en el capítulo 6 se concluirá todo el experimento.

# 1. Objetivos y Alcances

## 1.1 Objetivo General

Diseñar una integración de sistemas embebidos para computación de alto rendimiento usando manejadores de paquetes funcionales con el propósito de analizar su desempeño en HPC.

## 1.2 Objetivos Específicos

Revisar los manejadores de paquetes funcionales para entender su aplicación como uso de middleware en HPC en sistemas embebidos.

Diseñar y construir un pequeño clúster de sistemas embebidos.

Realizar la integración entre los sistemas embebidos y el manejador de paquetes seleccionado con el fin de usarlo en aplicaciones de HPC.

Diseñar e implementar un método de evaluación de desempeño tanto computacionalmente como energéticamente para dicho sistema.

## 1.3 Alcances

Mediante el desarrollo de este trabajo de investigación se pretende analizar los modelos actuales de integración entre los sistemas embebidos heterogéneos y manejadores de paquetes funcionales en la ejecución de HPC, con el fin de evaluar el rendimiento de dichos sistemas y su escalabilidad.

Así mismo, se pretende buscar un método de evaluación de dichos sistemas, que, en primera instancia, está frente al rendimiento computacional y energético pero que a través del proyecto se incorporarán nuevos parámetros.

Se espera que con este proyecto se pueda concluir positivamente al uso de HPC con sistemas embebidos, además que los manejadores de paquetes sean de un uso óptimo para la integración propuesta.

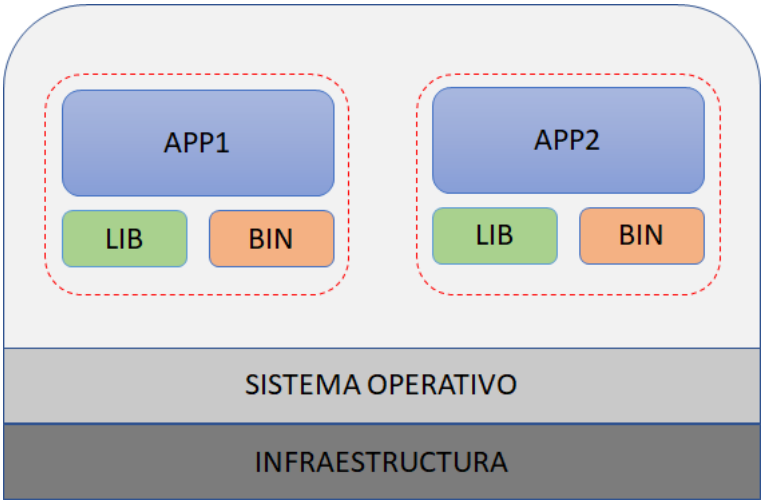
## 2. Estado del Arte

El proyecto se enmarca en analizar una infraestructura para HPC usando hardware y software no tradicional. En la parte del hardware se usa sistemas embebidos, un tipo de tarjetas que tienen la tecnología SoC y además poseen GPUs, lo que le dan un considerable bajo consumo al sistema. En la parte del software, se desea utilizar un software de tipo Manejadores de Paquetes, una herramienta que encapsula las aplicaciones de tal manera que quedan aislados de otras aplicaciones y así se evita problemas de incompatibilidad de versiones.

La razón de esta decisión se basa en una problemática básica pero muy compleja en los sistemas para High Performance Computing (HPC): la incompatibilidad de versiones de las aplicaciones debido al uso de múltiples usuarios con diferentes necesidades. Tal como se muestra en la Figura 1, usuarios distintos pueden requerir la misma aplicación, pero dependiendo de otros programas, esta aplicación puede ser de diferente versión. Y, como entonces se requiere diferentes versiones, entonces puede crease problemas entre ellas por su incompatibilidad ya sea por librerías o por eliminación de ciertas funciones que hacen que diferentes versiones no funcionen correctamente entre ellas.

**Figura 1**

*Problemática de la Incompatibilidad de Versiones*



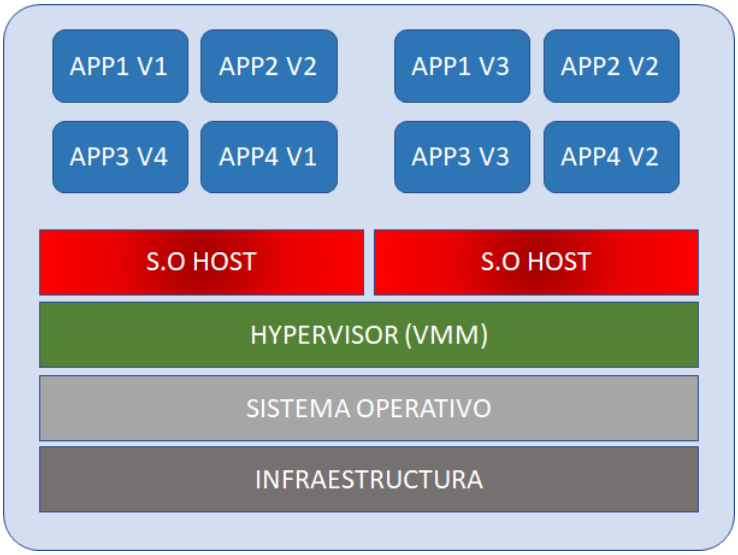

Para resolver este problema, hoy en día se tienen distintos modelos capaces de aislar estas aplicaciones, ya sea mediante el aislamiento completo del sistema (Máquinas Virtuales) o mediante un espacio de memoria (Contenedores). En la Figura 1 se indica este aislamiento en una burbuja de las aplicaciones, pero realmente para aislarlo es a través de una capa que los permita aislar. A continuación, se describirá cada modelo que hace de esta capa, con sus ventajas y desventajas.

## 2.1 Máquinas Virtuales

Las Máquinas Virtuales (MV) sirven para instanciar varios sistemas operativos en una sola máquina al mismo tiempo. Esto permite que múltiples usuarios accedan a diferentes configuraciones del sistema sin que se vean afectados entre sí. Cualquier error o nueva configuración en una máquina virtual no afecta el resto de las máquinas, lo que permite que haya mayor seguridad, además, al ser una instancia de la máquina es posible clonarla, lo que permite una portabilidad entre máquinas.

**Figura 2**

*Estructura de una Máquina Virtual*




Tal como se muestra en la Figura 2, la máquina virtual separa las aplicaciones incompatibles dividiendo la máquina en distintos sistemas operativos, así las aplicaciones solo se ven dentro de cada sistema operativo.

Las máquinas virtuales son muy usadas en servidores que ejecutan servicios de aplicaciones de alto desempeño, por lo que es bien conocido como la forma más directa de crear servidores para HPC instanciando una máquina virtual por usuario. Sin embargo, existe un inconveniente en esta tecnología, y es que requiere más procesos (cálculo) debido a que es necesario una capa entre el hardware y el sistema operativo que sea capaz de traducir. Esta traducción se hace a través del *Hypervisor*, lo que significa que va a requerir una traducción entre el sistema operativo invitado al sistema operativo anfitrión, haciendo que el rendimiento general del sistema decaiga. Además de lo anterior, el uso de máquinas virtuales es muy pesado ya que se requiere la imagen completa del sistema operativo junto con las aplicaciones que la acompañan.

Viendo esos inconvenientes, para hacer procesos HPC con un mejor rendimiento, es necesario tener una nueva tecnología que no requiera una capa tan pesada en traducción de

procesos y además que su imagen sea liviana (o casi nula). Dicho lo anterior, se han desarrollado nuevas tecnologías que evitan el uso del *Hypervisor* y en cambio usan un aislamiento de memoria. Esta tecnología viene de una idea llamada Contenedor, que más adelante se le conocería como Manejadores de Paquetes.

## 2.2 Contenedores

El concepto de Contenedor se remonta hace décadas cuando se planeó una forma de poder llevar la mercancía a otros países por medio de los barcos. En ese entonces había personas que recogían y levantaban con mallas la mercancía y lo ponían en el barco, teniendo cuidado que entre ellas no se dañaran. Lo que se les ocurrió después fue crear un objeto estándar para cualquier mercancía, así, lo que le pusieran adentro de este objeto dependía del dueño de la mercancía y el objeto (el contenedor) dependía de quien lo llevaba. Así se crea el Contenedor: un objeto que se puede transportar y almacenar en cualquier medio (camión, barco, entre otros) y que además permite separar el medio de transporte con lo que lleva. En (Lucas, Ballay, & McManus, 2012) cuentan esta historia y además le ponen una alegoría como una API en el mundo real.

De esta idea surge el pensar en un estándar para llevar la información. Ya en (Lucas, Ballay, & McManus, 2012) indican que existen 53 protocolos diferentes para manejar la información, que una alegoría sería tener un contenedor para cada tipo de mercancía. El objetivo final es, entonces, llegar a un “Contenedor de Información”, una “API” para manejar los datos que pasan entre las redes.

Tal como lo menciona en el libro Trillions (Lucas, Ballay, & McManus, 2012), este concepto fue propuesto en 1997 por Michael Dertouzos en el MIT.

De este pensamiento entonces se crea la “contenerización” de procesos. Los contenedores son un tipo de tecnología que permite apartar a los procesos dentro del entorno del sistema operativo. De forma general, esta separación se logra gracias a que encapsula todas las librerías necesarias para que la aplicación corra, por lo que se puede separar de las demás y que ésta no pueda ser vista por otros procesos (Lucas, Ballay, & McManus, 2012).

La idea de este tipo de “encapsulamiento” no es nueva. La contenerización (del inglés containerization o mejor OS-level virtualization) ya se encontraba desde hace tiempo como una tecnología en los sistemas basados en UNIX. La librería LXC (Linux Container) era la que se usaba para esta tarea. Sin embargo, hoy en día ha tenido un auge programas basados en estos modelos ya que representan un cambio frente al modelo de virtualización usando *hypervisor* (Lucas, Ballay, & McManus, 2012).

De la librería LXC se crea una llamada “libcontainer” que es aquella que hace la tarea de contenedor para el programa más conocido de este tipo llamado Docker. Entre otros programas se encuentran rkt de CoreOs, Oracle Solaris Containers y Nix, un manejador de paquetes que usa el paradigma funcional.

Aunque existen muchos contenedores, muchos de ellos no están pensados para HPC, son simplemente aplicaciones para aislar otros procesos ya que el sistema operativo lo requiere. Sin embargo, de esta idea surgen un grupo de contenedores que están pensados específicamente para HPC. Éstos contenedores reciben el nombre de Manejadores de Paquetes, y con el auge del acceso a sistemas HPC en la nube, han podido desarrollarse muy rápido.

## 2.3 Manejadores de Paquetes

Los manejadores de paquetes son una colección de herramientas que automatizan procesos de instalación, configuración, actualización y eliminación de programas de manera consistente, como los conocidos RPM y APT de las distribuciones Linux. (Package Manager, s.f.)

Para nuestro caso, se entenderá el "Paquete" como el encapsulamiento (el contenedor) de una aplicación o aplicaciones que sean aisladas del resto y que no se afecten entre ellas. A continuación se expondrán algunas soluciones de este tipo en el manejo de la instalación y ejecución de procesos aislados.

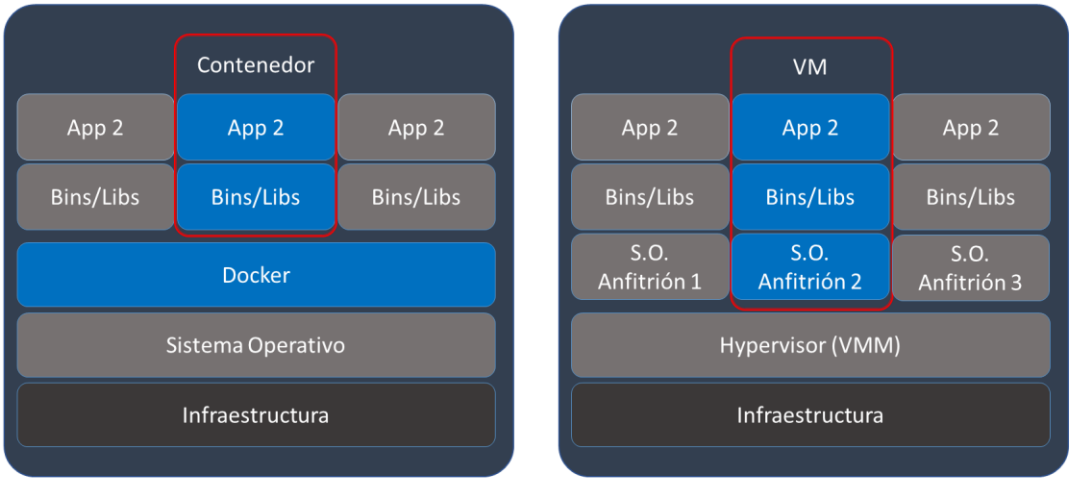
### *2.3.1 Docker*

Docker es un manejador de paquetes que tiene como insignia usar algo denominado "contenedores". Éstos Contenedores son una abstracción del sistema que encierra los programas y permiten que puedan ser ejecutados en cualquier máquina, simplemente la herramienta los encapsula y pueden ser ejecutados sin problemas.

Tal como lo menciona la página de Docker, una imagen de contenedor es un paquete ejecutable liviano de una pieza de software que incluye todo lo necesario para "correr": código, herramientas del sistema, librerías, entre otros. Contenedores separan el software de sus alrededores y ayuda a reducir conflictos entre equipos que estén corriendo diferente software en la misma infraestructura. (Docker Web Page, s.f.)

**Figura 3**

*Docker vs. Máquina Virtual*




*Nota.* Tomado de https://www.docker.com/what-container#/virtual_machines.

Como lo muestra la Figura 3, la diferencia entre Docker y las máquinas virtuales radica en la abstracción de dichos sistemas: Los contenedores son una abstracción de la capa de aplicación que empaqueta el código y las dependencias juntas. Se puede tener corriendo múltiples contenedores en la misma máquina ya que cada uno de ellos "corren" en procesos aislados del espacio de usuario. Por tal motivo, los contenedores pesan mucho menos que una imagen de máquina virtual (aproximadamente unos cuantos Mb) y además inician casi de inmediato. (Docker Web Page, s.f.)

Las máquinas virtuales, por el contrario, son una abstracción del hardware físico. Esto significa que cada máquina virtual contiene una copia total del sistema operativo, de las aplicaciones, archivos binarios y librerías, dejando una imagen que puede pesar muchos GBs. Aunque el *hypervisor* permite la ejecución de múltiples máquinas virtuales en una misma máquina, el inicio de cada una de ellas podría ser muy lento. (Docker Web Page, s.f.)

### *2.3.2 Nix/Nixos*

Nix también es un manejador de paquetes, pero en vez de llamar “contenedores” a la abstracción de las aplicaciones, los llaman “paquetes”.

Nix es un “manejador de paquetes puramente funcional”. Esto significa que trata a los paquetes como valores tal como en el paradigma funcional. Los paquetes son construidos por funciones que no tienen efectos secundarios, y que nunca cambian después de que son construidos. Nix guarda los paquetes en un directorio denominado “Nix Store” ubicado generalmente en */nix/store* donde cada paquete se guarda en un único subdirectorio con un código hash en él. (Nix Package Manager, s.f.)

También existe NixOS, una distribución Linux basado en Nix, que realiza toda la configuración del sistema operativo con el paradigma de manejador de paquetes puramente funcional, completamente declarativo y, según su página oficial, hace muy confiable las actualizaciones. (NixOs, s.f.)

### *2.3.3 Spack*

Spack permite que cualquier número de compilaciones pueda coexistir en el mismo sistema, asegurando que los paquetes instalados puedan encontrar sus dependencias, independientemente del entorno (Gamblin, et al., 2015).

Spack está pensado específicamente para supercomputadores. Los paquetes quedan aislados de cualquier otro paquete, así que se permite múltiples versiones de un mismo programa sin afectar el entorno. Además, lo más llamativo de Spack es la forma de instalar las versiones. En una sola línea de comandos y de manera casi transparente, es posible especificar el programa de

instalación, su versión, el compilador y su versión y muchas otras opciones de compilación y banderas, haciéndola muy versátil en su aprendizaje (Gamblin, et al., 2015).

Por otro lado, si se desea crear un paquete (por ejemplo, un programa especializado del área de estudio), es necesario crear una especie de "receta" donde le indica a Spack las instrucciones de compilación, luego, Spack le crea las dependencias de compilación automáticamente.

Es necesario resaltar que entre Spack y Nix existe una similitud en la creación de esta "receta". Obviamente las recetas difieren en su lenguaje, sin embargo, tratan de indicar las mismas definiciones. A mi parecer es mucho más entendible la especificación de compilación en Spack y además el hecho de que en una línea de comando se pueda instalar con muchas opciones un paquete lo hace más sencillo de aprender. El único inconveniente resulta en su portabilidad, ya que está "amarrado" a la máquina ya que usa un programa propio de creador de entornos que es instalado antes de instalar esta aplicación.

### *2.3.4 EasyBuild*

Según su página, EasyBuild es un compilador de software y *framework* de instalación que permite administrar software (generalmente científico) en sistemas HPC de manera eficiente (Easy Build Web Page, s.f.).

Al igual que los manejadores de paquetes anteriores, EasyBuild controla como compilar un programa, además permite la existencia de múltiples versiones y compilaciones. Un punto especial que comentar es que permite la construcción de software en paralelo, y hay que señalar que su audiencia principal es a los usuarios HPC.

## 2.4 Proyectos de Sistemas Embebidos

En la parte de Sistemas Embebidos, han existido proyectos que, viendo el poder de cómputo y eficiencia energética de estos sistemas, han desarrollado infraestructuras que explotan estas ventajas, analizando sus características y observando sus limitaciones a través de los años. En los siguientes puntos, se indicarán algunos proyectos involucrados en este tema.

### *2.4.1 Tibidabo*

Bajo el problema que la energía total de un sistema a lo largo de los años puede ser más costosa que lo obtenido del hardware en la infraestructura, este proyecto construye lo que según ellos es el primer clúster a gran escala usando procesadores ARM (Rajovic, Rico, Puzovic, & Adeniyi-Jones, 2013).

En esta propuesta, los autores señalan que los CPU usados en estos sistemas ARM no están construidos para HPC debido a que fallan en muchos aspectos de comunicación y calculo vectorial entre otros. Sin embargo, lo que más llama la atención es el uso de la GPU que, al consumir menos energía que la parte de CPU, se puede usar este recurso para los cálculos HPC. (Rajovic, Rico, Puzovic, & Adeniyi-Jones, 2013)

Su arquitectura era un clúster de NVIDIA Tegra2 SoC con un procesador Dual-Core ARM Cortex-9 a 1Ghz. Al ser un clúster de ARM antiguos, este hardware no soportaba modelos de programación como son CUDA o OpenCL, no obstante, en las pruebas realizadas se observó unos valores competitivos con los Intel Xeon X5660 (Rajovic, Rico, Puzovic, & Adeniyi-Jones, 2013).

Como era de esperarse, el clúster no fue lo suficientemente eficiente para competir con otros sistemas que se emplean para HPC. Sin embargo, los autores proyectan un buen futuro en el

desarrollo de esta arquitectura debido a que la utilidad se puede explotar si el hardware se modifica para este fin.

Por esta razón, señalan algunas recomendaciones para el desarrollo de futuros sistemas basados en ARM, que principalmente se ven enfocados en los puntos que no tenía esta versión o que tenía en carencia. Estas recomendaciones son tener un procesador ARM que soporte HPC y además que su diseño tenga múltiples *cores* (como 16 por ejemplo), tener unidades SIMD que soporten cálculos de puntos flotantes de doble precisión, soporte de controladores de memoria ya sea DDR3 o DDR4 y que soporte redes de alta capacidad para HPC como *Infiniband*. Todo lo anterior unido al hecho que desde la parte del software debe existir librerías, compiladores, drivers y sistemas operativos que soporten este tipo de sistemas para ARM y HPC (Rajovic, Rico, Puzovic, & Adeniyi-Jones, 2013).

Este proyecto es plausible debido a que, según los autores, es el primero en crear un clúster ARM, lo que significa que son los primeros en incursionar en la creación de *clústers* para HPC utilizando chips ARM que se especializan en ser de bajo consumo de energía. Sin embargo, tal como ellos lo mencionaron, este sistema no llegó a poder competir con los sistemas hechos para HPC, debido a falencias en el desarrollo de dichos chips. Cabe resaltar que ellos conocían con anterioridad estas deficiencias y señalan que otros sistemas como el Cortex-A15 (el mismo que tiene la NVIDIA Jetson TK1) poseen ya una arquitectura que es capaz de tener un mejor rendimiento para HPC, y además, señalan que el Cortex-A57 (el de la NVIDIA Jetson TX1) es 4 veces mejor que el Cortex-A15 y al tener una implementación del ARMv8 y 8 doble-precisión punto flotante, es una buena candidata para llegar aún más lejos en el rendimiento del sistema.

### *2.4.2 Icarus*

El proyecto ICARUS (Geveler, et al., 2017), es un centro de HPC escalable de energía eficiente basado en sistemas embebidos. Su enfoque único para High Performance Computing (HPC) se basa en el uso de hardware no convencional y un sistema que produce su propia energía con el fin de tener una infraestructura HPC que sea capaz de proveerse su propia energía para todo el sistema.

Su infraestructura consta de dos partes: por un lado, el hardware usado son los sistemas embebidos de NVIDIA Jetson TK1, un sistema SoC que tiene GPU de bajo consumo, ensamblados en una hélice dentro de un pequeño almacén, y por otro lado, un sistema de 2 módulos de energía fotovoltaica (los paneles solares) al lado de este.

En términos de poder de rendimiento, el clúster consta de 60 NVIDIA Jetson TK1, que posee un chip que tiene un CPU quad-core de 32 bit ARM cortex-A15 y un GPU Kepler que juntos se comparten utilizando la DRAM. Con su rendimiento teórico de 300GFlops/s y un poder de disipación de 10 W. Los 60 Jetson TK1 equivalen a 250 ARM CPU *cores* y 60 GPUs que, según la fuente, ofrece un pico teórico de rendimiento de 21 TFlop/s con un poder de disipación de menos de 1 Kw, y por otro lado tienen un sistema de paneles solares que entregan la energía que este sistema requiere (Geveler, et al., 2017).

La red de ICARUS se compone de *switches* de 28 puertos GiB Ethernet con una capacidad de 56GB/s. El acceso al clúster se hace por medio de un nodo dedicado. La memoria eMMC que posee el nodo tiene una capacidad de 16GB y es usada por el sistema operativo y los datos primarios. También el proyecto indica que a cada nodo le proporcionaron una tarjeta SD de 128GB Ultra SDXC 40MB/s Class 1.

Por el lado de la granja fotovoltaica, se usaron 30 módulos solares con una generación individual de poder máxima de 255W cada uno, resultando en 7.65kWp

Como en el proyecto es mencionado, ICARUS es un diseño piloto de energías renovables con el objetivo de usarse con software de simulación. No está pensado para explotar todas las capacidades de las tarjetas en conjunto, pero sí para ver su desempeño energético. El proyecto no indica como fue el proceso de acoplar todas las tarjetas o que programa se utilizó para poder hacer funcionarlas como un todo, pero si es un buen proyecto para observar cómo es posible utilizar en HPC sistemas diferentes a los tradicionales.

### *2.4.3 Montblanc*

El proyecto MONTBLANC (Montblanc-Project, s.f.), es una propuesta europea que se basa en un supercomputador de sistemas ARM, lo que propone un bajo consumo energético y a la vez busca mejorar el rendimiento computacional de dichos sistemas en cada nueva fase que se encuentra. El proyecto inició en el 2011 con pocas empresas como Bull o ARM y hoy en día se han sumado más laboratorios y centros europeos para su desarrollo en la 3ra fase que es en la que se encuentra en este momento.

MONTBLANC, coordinado por el centro de supercomputación de Barcelona, este supercomputador que inició en el 2011 integra un sistema de 1080 tarjetas que en conjunto forman 2160 CPUs y 1080 GPUs. El sistema está basado en el procesador Samsung Exynos 5 Dual, mientras que los CPU son Cortex-A15 dual *core*, la misma gama del Tegra TX1.

La tercera fase está en ejecución hasta septiembre del 2018 e involucra aproximadamente 3000 *cores* distribuidos en 48 nodos o 96 procesadores Cavium ThunderX2 de 64-bit. Esta fase tiene un presupuesto de 7.9 millones de Euros (Montblanc-Project, s.f.). Actualmente está

desarrollando una aplicación llamada MUSA que es una simulación de multi nivel: hace trazas de la simulación con estrategias de ejemplo que explora como la arquitectura afecta la eficiencia computacional a larga escala.

Al igual que ICARUS, MONTBLANC también es un proyecto que tiene énfasis en observar los limites en las tarjetas embebidas. Sin embargo, al igual que el primero, no hace énfasis en cómo se organizó las tarjetas de tal manera que pudieran funcionar en sincronía, pero podría hacer hipótesis siendo software libre modificado o algún software privado hecho a la medida en el centro en el cual pertenece.

### *2.4.4 Proyectos Varios con Raspberry Pi*

Un proyecto de menor escala también se ha propuesto. Usando un clúster de 8 tarjetas Raspberry Pi interconectadas con un *switch*, este diseño hace evaluación de rendimiento para minería de datos, específicamente los algoritmos de *Big Data A priori* y *K-means*, concluyendo que en el algoritmo *A priori* tenía mucho mejor desempeño este clúster que un sistema convencional con un procesador Intel Xeon Phi (Saffran, et al., 2017).

La universidad de Glasgow también creó un proyecto de Raspberry Pi, haciendo un clúster de 56 Raspberry Pi divididos en 4 torres de 14 cada uno. Este clúster utiliza un sistema operativo denominado Raspbian que está optimizado para estas tarjetas. (Tso, White, Jouet, Singer, & Pezaros)

En este caso el proyecto sí indica algo relacionado con las tarjetas embebidas: tienen un sistema operativo que es optimizado para las propias tarjetas que se han usado y además lo usan como prueba en algoritmos de probabilidad. A pesar de todo no menciona el software selecto para la interconexión de las tarjetas, pero sí hay una mención de algún algoritmo HPC.

Este proyecto en lo personal tiene una mayor atención debido a que también hay una modificación en el sistema operativo que además se vuelve una parte importante a la hora de elaborar la integración de todas las tarjetas embebidas. No obstante, la decisión de si usar un software libre o no para la integración depende de cómo esté desarrollado el software general (manejador de paquetes) ya que es un tipo de paradigma nuevo que promete mejorar el desempeño de los programas.

### *2.4.5 Nix en un Ambiente HPC*

En el centro de supercomputación GRICAD en Grenoble se ha usado Nix, el manejador de paquetes funcional para la interconexión de un clúster de prueba en revisión de su operación.

En (Bzeznik, Henriot, Reis, Richard, & Tavard, 2017) indican que usaron Nix dentro de un clúster en sus laboratorios durante 12 meses, reportando sus experiencias, con respecto al clúster y con respecto a los usuarios quienes los usaban. Ellos mismos crearon ciertos programas con el fin de optimizar ciertos procesos del clúster y sus conclusiones están más dadas a la parte humana en el aspecto de hacerlo más accesible a los programadores.

Este proyecto es la mejor aproximación de lo que se desea hacer en el proyecto de grado. Sin embargo, no usa ni lo menciona en ninguna parte los sistemas embebidos, ya que su objetivo era probar el funcionamiento desde el punto de vista de la experiencia de usuario, y además se hizo en un clúster de HPC tradicional. No obstante, es una buena guía para entender algunos aspectos de Nix y posibles problemas a la hora de utilizarlo en los sistemas embebidos.

## 2.5 Resumen del Capítulo

En esta sección se mencionó que el proyecto está enfocado en la infraestructura HPC usando sistemas embebidos y Manejadores de Paquetes. Se utilizan estos dos elementos debido a que se espera un gran rendimiento tanto computacionalmente como energéticamente debido a que los Sistemas Embebidos consumen menos energía y con las GPUs pueden realizar las tareas para las cuales HPC está enmarcado.

Se realizó un recorrido de las tecnologías y herramientas actuales para HPC. Se conoce que existen incompatibilidades entre versiones de aplicaciones y se espera de alguna manera aislar estas aplicaciones para que de alguna manera coexistan en una misma máquina. De esta premisa entramos a las Máquinas Virtuales que, aunque resuelven nuestro problema, genera nueva complejidad algorítmica por su capa de traducción al Sistema Operativo anfitrión (*Hypervisor*). Mas adelante hablamos de la definición de Contenedores que se refiere a encapsular de alguna manera las aplicaciones y con ella se menciona Manejadores de Paquetes que exactamente hace ésto pero aislando entornos de memoria para permitir que coexistan diferentes versiones de aplicaciones.

Finalmente, expusimos varios proyectos que se han adentrado al uso de HPC usando Sistemas Embebidos: Tibidabo siendo la primera precursora a una infraestructura de este tipo, ICARUS explotando el bajo consumo energético usando paneles solares para no requerir consumo de energía, MontBlanc siendo un proyecto de sistemas ARM a gran escala, los proyectos de Raspberry Pi como Glasgow que usa Sistemas Embebidos para hacer operaciones estadísticas y el uso de Nix dentro de un supercomputador tradicional GRICAD que principalmente su revisión se enfocaba en la curva de aprendizaje de la herramienta Nix, un Manejador de Paquetes.

Es importante conocer estas propuestas debido a que permiten explorar los distintos frentes en los que este proyecto se desarrolla. Permite conocer la posibilidad de la creación de una infraestructura de este tipo, comprendiendo las ventajas y limitaciones de la misma, además de que permite comprender el funcionamiento de los Manejadores de Paquetes.

A grandes rasgos se presenta en la Tabla 1 los manejadores de paquetes y contenedores que se dieron a conocer en este capítulo con algunas de sus características. Así mismo en la Tabla *2* se presentan los proyectos similares a la propuesta presente mostrados en el presente capítulo con algunas características generales relacionadas con el proyecto que se realiza.

**Tabla 1**

*Resumen General de los Manejadores de Paquetes Explorados en este Proyecto*

| | **Docker** | **EasyBuild** | **Spack** | **Nix** |
|---|---|---|---|---|
| **Generalidad** | Capa intermedia que encapsula la app | Crea ambientes modulares | Aborda el problema de instalación de la app | Capa intermedia que encapsula la app |
| **Propiedades Positivas** | Popular en las empresas | Aborda el problema de la compleja instalación de la app | Sencilla interfaz de usuario ideal para HPC | Concepto de paquete: las app se vuelven funciones |
| **Conceptos** | Contenedor de app | Easyblock: Automatiza build de app | 3 Fases: Normalizar, Concretizar, Almacenar | Los paquetes se almacenan en 1 solo directorio |

**Tabla 2**

*Un Resumen General de Algunos Proyectos Similares a la Presente Propuesta*

| Tibidabo | ICARUS | MontBlanc | Nix-GRICAD |
|---|---|---|---|
| Primer clúster HPC usando procesadores ARM | Centro de HPC escalable con energía eficiente | Proyecto europeo de HPC con sistemas ARM | Prueba de Nix dentro de un centro HPC |
| Usaron NVIDIA Tegra2 SoC dual core ARM Cortex-9 | Usaron 60 NVIDIA Jetson TK1 y 2 módulos de energía fotovoltaica | Usaron 96 procesadores Cavium ThunderX2 64-bit | Usaron Nix durante 12 meses para ver el desarrollo de la herramienta. Analizaron aspectos como la comodidad del usuario y el aprendizaje. |
| No soportaba ni CUDA ni OpenCL | Enfocado en tener un centro autosuficiente en energía | La 3ra fase involucra 3000 cores distribuidos en 48 Nodos | |
| Posee resultados que competían con Intel Zeon X5660 | Proyecto piloto que sirve como clúster de simulación | Objetivo es HPC con bajo consumo energético | Objetivo es la viabilidad de Nix en HPC |

Como se puede apreciar en la Tabla 1, a pesar de que cada herramienta tenga ciertas características diferentes frente al manejo de paquetes, todas ellas buscan crear ambientes modulares que permitan el aislamiento de procesos frente a otros incompatibles. Incluso Spack, que, a pesar de que no aísla procesos, tiene un sistema que detecta la mejor instalación para el sistema dado, que en el caso de administración de HPC, es lo necesario para poder entregar los servicios a los usuarios.

En el caso de las propuestas similares al presente proyecto, indicados en la Tabla 2, se observa que todas tratan de explorar un lado diferente sobre este tipo de arquitecturas, desde el uso de las mismas arquitecturas en un diseño piloto (Tibidabo) pasando por el uso de bancos de energía sostenible para sus modelos (ICARUS), hasta el uso masivo para el público usando arquitecturas ARM (Montblanc). En el caso de Nix-GRIDCAD su exploración es más en el entorno de usuario

dentro de un sistema tradicional, que para el plan de esta propuesta es necesario por su parte de software en estos sistemas.

En el siguiente capítulo se mencionará con más detalle los distintos Manejadores de Paquetes, su funcionamiento, ventajas y desventajas, indicando el Manejador de Paquetes más apto para este proyecto.

## 3. Manejadores de Paquetes Funcionales

Como mencionamos en el anterior capítulo, el problema de la incompatibilidad entre aplicaciones o versiones de aplicaciones se puede resolver encapsulando o aislando de alguna manera estas aplicaciones. Un modelo que resuelve este problema son las Máquinas Virtuales que a través de la capa del *Hypervisor* o VMM (Virtual Machine Monitor) que separa por sistemas operativos todas las aplicaciones. Sin embargo, esta capa es muy "pesada", es decir, se requiere hacer más cálculos para poder ir de lo que pide la aplicación al hardware, además la imagen que se crea es muy pesada ya que se necesita el almacenamiento del sistema operativo en el que las aplicaciones están contenidas.

Es por esta razón que se amplían los modelos y se encuentran con uno que disminuye esta capa de traducción, eliminando sistemas operativos invitados y traduciendo el código casi directamente al hardware: los Manejadores de Paquetes.

En esta sección hablaremos de los Manejadores de Paquetes conocidos y la razón por la cual escogimos Nix como el Manejador de Paquetes para hacer el análisis de rendimiento en este proyecto. Mostraremos las ventajas y desventajas de usar cada Manejador de Paquetes desde el punto de vista del High Performance Computing (HPC).

Cabe mencionar los beneficios generales de la “Contenerización”. Aunque se menciona en la página de inicio de Docker, aplica también a cualquier Manejador de Paquetes, después de todo usan el mismo modelo. Las ventajas de la Contenerización son las siguientes (Docker Get Started, s.f.):

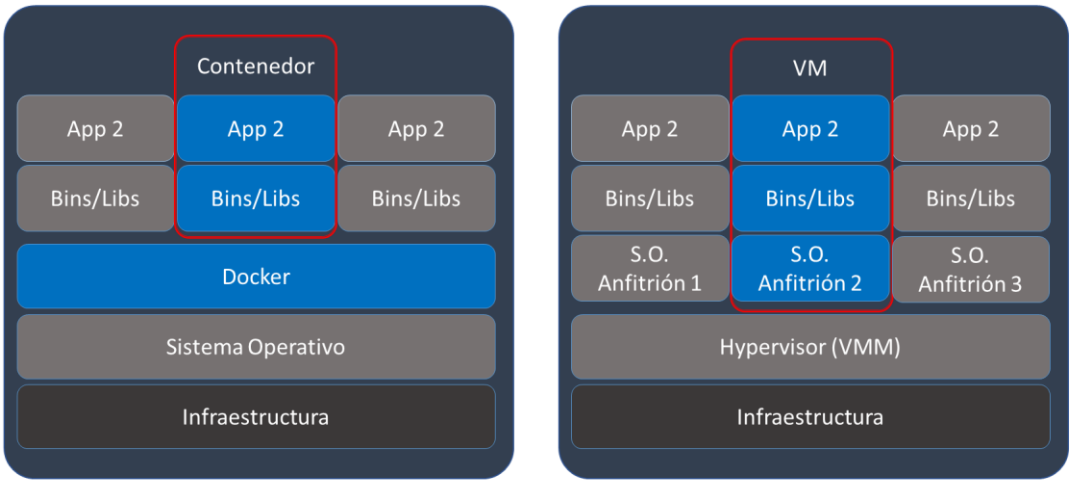
- Flexible: En teoría puede encapsular cualquier aplicación.
- Ligero: Su imagen es más liviana que las máquinas virtuales ya que comparten el *Host Kernel* con la aplicación.
- Intercambiable: Los paquetes se pueden actualizar y cambiarse entre ellos sin tener que modificar los otros paquetes.
- Portable: La imagen o el paquete creado, puede ejecutarse en otro lado, ya sea en servidores u otros sistemas compatibles con los Manejadores de Paquetes.
- Escalable: Al tener alta portabilidad, se pueden crear múltiples réplicas en muchos sistemas.

## 3.1 Docker

En la sección anterior se había mencionado acerca de Docker como Manejador de Paquetes que encapsulan las aplicaciones con las librerías necesarias para su funcionamiento. Como se había mencionado, la diferencia entre Docker y las Máquinas Virtuales radican en la capa que traduce las aplicaciones tal como se muestra en la Figura 4:

**Figura 4**

*Diferencias entre un Contenedor y una Máquina Virtual*




*Nota.* Tomado de https://www.docker.com/what-container#/virtual_machines.

Como lo muestra la imagen, la Máquina Virtual contiene una capa intermedia entre el sistema operativo invitado y el sistema operativo anfitrión llamada *Hypervisor*. Esta capa es la que aísla los distintos sistemas operativos incluyendo su configuración y aplicaciones, sin embargo, esta capa genera un sobre uso de recursos por la traducción de los comandos a la infraestructura, además de que es muy pesada su imagen por lo que se requiere mucho espacio para mantener un sistema con este modelo.

De la misma forma, la Figura 4 muestra que Docker usa una capa superior pero sin Sistema Operativo invitado, lo que permite aislar las aplicaciones y sus binarios sin necesidad de dividirlos en sistemas operativos. Esta aproximación genera un mejor manejo de recursos y una imagen más liviana.

Según (Bashari Rad, John Bhatti, & Ahmadi, 2017), Docker es una plataforma abierta que puede ser usada para compilar, distribuir y ejecutar aplicaciones en una herramienta portable,

ligera y empaquetadora conocida como Docker Engine. Su principal ventaja es en su uso en la Nube que, al ser portable, no requiere una re-compilación en el servidor gracias a que es portable. Su forma estándar de empaquetar las aplicaciones con sus binarios es el Contenedor.

Docker se puede dividir en 4 ramas diferentes: Docker Client and Server, Docker Images, Docker Registries y Docker Containers. Docker puede ser considerado una aplicación basado en Cliente-Servidor en el que hay un demonio (Daemon) que espera y envía las peticiones entre los contenedores de Docker. Estos contenedores vienen primero en la creación de imágenes, ya sea copiado directamente de uno ya hecho o con la herramienta Docker Build. Éstas imágenes están dentro del Docker Register (registros Docker) donde están todas las imágenes que pueden generar contenedores, ayudados por Docker Hub que contiene la información de Contenedores privados y públicos. El usuario escoge una imagen y mediante operaciones dentro de Docker genera el Contenedor (*Docker Container*) (Bashari Rad, John Bhatti, & Ahmadi, 2017).

Igualmente, el *paper* menciona que su adaptabilidad es menor, debido a que requiere menos tiempo de instalación, esto, principalmente porque la imagen es mucho menor.

Uno de los problemas que menciona este *paper* es que “cuando se reduce el aislamiento, se disminuye la seguridad pero se gana rapidez”. El aislamiento de Docker se reduce frente a la máquina virtual con la nueva capa, lo que al final puede comprometer el sistema al no estar aislado al generar errores.

Cabe mencionar que no todas las aplicaciones podrían ejecutarse en Contenedores.

En resumen, el articulo menciona que como ventajas Docker posee velocidad, portabilidad, escalabilidad, entrega rápida (gracias a la escalabilidad), y densidad (debido a que puede haber más contenedores en un mismo host). Por el contrario, sus desventajas radican en que Docker no posee virtualización completa ya que requiere el Linux Kernel para funcionar, solo soporta las

máquinas de 64-bits y la seguridad de los paquetes se puede ver comprometida con respecto a las VM. (Bashari Rad, John Bhatti, & Ahmadi, 2017)

### 3.2 EasyBuild

El inconveniente de instalar software científico en máquinas para High Performance Computing (HPC) radica en que los usuarios de estos sistemas prefieren diferentes versiones específicas de un mismo programa, lo que podría afectar entre ellos el sistema general debido a la incompatibilidad de algunos programas. La forma más sencilla es instalando "a mano" el software teniendo en cuenta estas restricciones, usando *Environment Modules* para separar estos programas dependiendo de los requerimientos de cada usuario.

Sin embargo, "a mano" no se puede realizar en sistemas muy grandes ya que genera mucho trabajo y costo de tiempo al personal que administra el lugar. Así que otra forma es usar *Scripts* que automaticen todo el proceso, haciendo más rápido la instalación pero siendo difícil de mantener y de hacerlo portable a otros sistemas HPC. Otra solución son los *Package Manager* como yum, RPM y apt-get que automatizan todo el proceso, revisando las dependencias entre paquetes, pero no soporta múltiples compilaciones y versiones de un mismo paquete, algo que se requiere en sistemas HPC (Geimer, Hoste, & McLay, 2014).

Otras soluciones mencionadas en (Geimer, Hoste, & McLay, 2014) son las *Custom Tools*, herramientas que hacen instalaciones personalizadas para los sistemas HPC. No obstante, estas herramientas mueren silenciosamente ya sea porque ya no se encuentra el desarrollador o porque la aplicación no es flexible y la gente ya no lo usa. Y la otra solución es crear *Modules Files*, pero estos archivos solo los podrán mantener una serie de personas debido a su complejidad en el diseño.

Para solucionar este inconveniente, se usa un modelo denominado *Hierarchical Module Naming Scheme*, en el que organiza las dependencias modulares bajo un árbol, pudiendo así indicarle al usuario que módulos son válidos dentro del módulo en el que se encuentra. Éste modelo requiere 3 características: Visibilidad de Módulos, Conocimiento de las extensiones Modulares y la Disponibilidad de un Módulo en diferentes caminos del directorio. (Geimer, Hoste, & McLay, 2014)

Según (Geimer, Hoste, & McLay, 2014), EasyBuild es un framework de compilación e instalación de software escrito en Python, cuyo objetivo primario es aliviar la eterna carga de compilar e instalar software científico.

EasyBuild fue un proyecto que inició en el 2009 por el equipo HPC de la universidad de Ghent en Bélgica. Hoy en día es un proyecto altamente documentado, con miles de paquetes que soporta y una comunidad sólida. Con la frase "construir software con facilidad" EasyBuild provee una forma fácil pero poderosa para automáticamente instalar software (científico) de manera robusta, consistente y reproductiva (Geimer, Hoste, & McLay, 2014).

EasyBuild consiste en una colección de Módulos Python y paquetes que interactúan entre ellos. Tiene su propio *framework*, donde el usuario ejecuta los comandos para interactuar e instalar con la herramienta, utiliza algo denominado EasyBlocks que es una implementación de cómo se compila e instala un paquete de software. Posee un archivo específico llamado Easyconfig Files en el que se especifica los parámetros del paquete.

Algunas características a favor de EasyBuild son la Resolución Automática de Dependencias, la Configurabilidad de la propia herramienta a necesidades específicas y la Extensibilidad Dinámica, que viene siendo el agregar a un módulo otros módulos como un rompecabezas (Geimer, Hoste, & McLay, 2014).

A esta herramienta se le suma otra que vienen casi juntas, llamada Lmod, que es una herramienta de modulos que su principal soporte es monitorear y revisar los cambios en los módulos dados por el path de módulos. Tal como lo menciona (Geimer, Hoste, & McLay, 2014), "la habilidad para mantener un grupo consistente de módulos jerárquicos y navegar el árbol de módulos fuera del presente *ModulePath* son algunos de las fortalezas clave de Lmod".

A pesar de sus ventajas, hay que señalar que EasyBuild aún posee ciertas propiedades que debe incluir, como incluir versiones no-restrictivas, por ejemplo, especificar que debe instalar desde una versión. También debe incluir el análisis de la resolución de dependencias que se encuentran en las subcadenas de paquetes.

### 3.3 Spack

Otra herramienta que fue creada pensando en el mismo inconveniente de la multiplicidad de versiones en un entorno de sistemas de High Performance Computer (HPC) es Spack. La herramienta Spack permite que cualquier cantidad de compilaciones puedan coexistir en un mismo sistema, y asegura que los paquetes instalados puedan encontrar sus dependencias, independientemente del entorno (Gamblin, et al., 2015). Según el autor, el entorno de instalación de Spack solo incurre en un 10% de sobretiempo en la compilación comparado con su forma nativa.

Spack (viene de la frase Supercomputing Package Manager) está construido en Python como EasyBuild, además sus paquetes pueden referirse con un *hashtag* como Nix y soporta un sin número de instalaciones de software. Sus principales características, dado por (Gamblin, et al., 2015) son la Combinatoriedad de Paquetes, parametrizados por versión, plataforma, compilador, opciones y dependencias, una Sintaxis de Especificaciones nueva donde se encarga de indicar los

parámetros de la compilación del programa (algo así como una receta), entregar versiones dependencias virtuales para manejar interfaces con versiones, un proceso de Concretizacion (*Concretization*) para traducir la especificación abstracta (dado en la receta) a una especificación de compilación concreta y Compile Wrappers para asegurar consistencia y simplicidad al escrito de paquetes (quiere decir que todo estará adentro de Spack).

Lo que realiza Spack es revisar las especificaciones de compilación y revisar el árbol de dependencias para saber si se puede realizar esta operación, luego revisa las versiones disponibles, los parámetros que se indicaron como número de versión o alguna forma restrictiva de la misma (por ejemplo versión superior a 2.2) y al final crea y ejecuta los pasos necesarios para instalar este programa.

Incluso Spack puede instalarse programas que ya tienen esta receta pre-definida (aunque también usted puede crearla o modificarla), todo esto en una sola línea de comandos, lo que lo hace muy intuitivo al usuario final que no se debe preocupar por las dependencias ni incompatibilidades de la instalación, ejemplo:

*spack install hdf5*

*spack install hdf5@1.10.1*

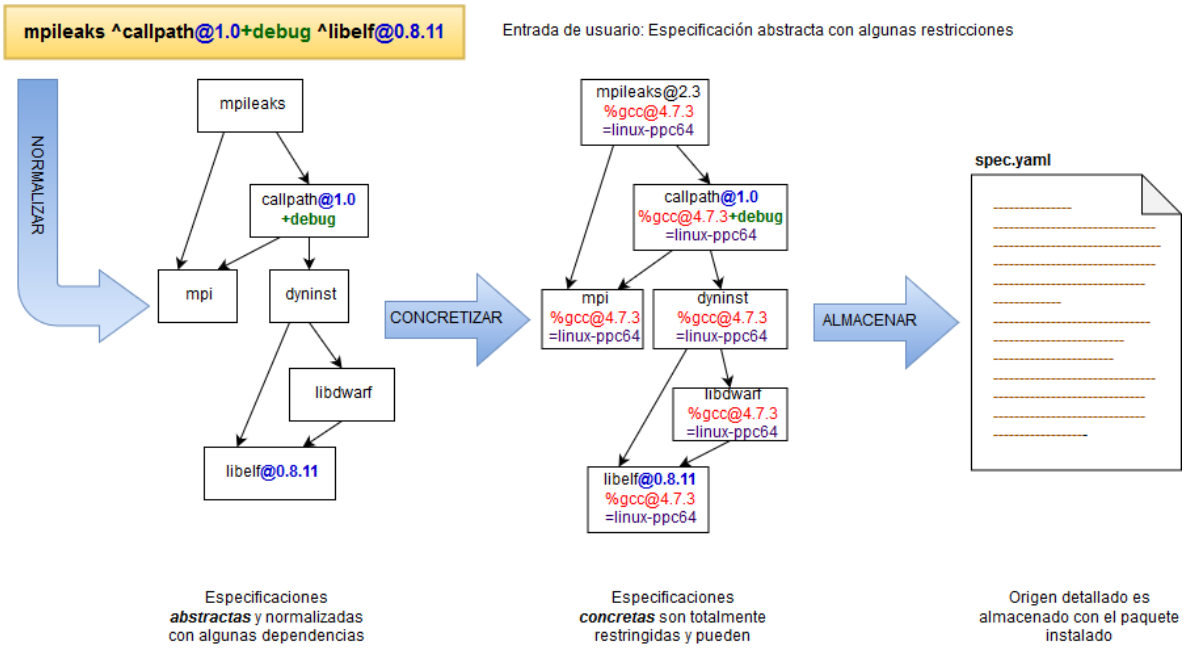
El primer comando instala hdf5 en su versión más reciente, el 2do comando instalaría hdf5 pero una versión específica (hdf5 ver. 1.10.1). También se puede especificar aún más la compilación:

*spack install hdf5@1.10.1 %gcc@4.7.3*

Donde el comando instalaría hdf5 versión 1.10.1 y usando el compilador GCC en su versión específica 4.7.3. (Spack, 2018). Por lo que se puede observar, esta forma intuitiva de poder especificar una compilación e instalación de algún paquete es lo que hace a Spack muy atractivo.

**Figura 5**

*Proceso Completo de Creación del Paquete en Spack*




*Nota.* Tomado y rediseñado de la exposición de Spack del año 2015 que se encuentra en https://github.com/spack/spac.

Como se muestra en la Figura 5, a partir de un comando en Spack, la herramienta convierte este comando en un árbol de dependencias pero con paquetes abstractos, es decir, sin especificar versiones ni opciones de compilación (al menos que se haya especificado). A este proceso se llama Normalizar. El siguiente proceso, Concretizar, convierte ese árbol de dependencias sin versiones ni opciones de compilación, a un árbol de dependencias completo con su respectiva versión, y opciones de compilación. Finalmente este árbol se guarda en disco donde se especifica cómo hay que instalar el software.

El *core* de Spack es el algoritmo que ellos llaman de Concretización (*Concretization algorithm*), ya que es la encargada de resolver las dependencias especificadas en la receta y la que decide qué y cómo instalar las dependencias. En síntesis, del modelo que genera la receta, el algoritmo empieza a buscar por cada rama las dependencias necesarias y generando los pasos para instalar cada una de ellas. Es importante resaltar que este proceso es una instancia del *Constraint*

*Satisfaction Problem (CSP)*, por lo que es NP-Completo. A pesar de lo anterior, los autores mencionan que no esperan que en el futuro cercano existan paquetes con miles de dependencias, por lo que esta parte no se ve amenazada por ser un cuello de botella. (Gamblin, et al., 2015)

En conclusión, Spack es una herramienta de Manejador de Paquetes que provee una compilación parametrizada de los paquetes (especificaciones detalladas de cómo lo quiere instalar), construyendo el software independientemente del entorno (Gamblin, et al., 2015). Utiliza un algoritmo que se encarga de revisar las dependencias y luego con los pasos de instalación simplemente lo hace. Aunque tiene un 10% de ineficiencia con respecto a la instalación nativa, es más intuitiva para los usuarios que estar revisando las dependencias, lo que lo hace portable y flexible.

### 3.4 Nix/NixOS

Finalmente, se llega a Nix, con su variante completa en sistema operativo NixOS. Nix es un Manejador de Paquetes funcional, lo que trata a los paquetes como funciones (tal como lo hace el paradigma funcional). Desarrollado por Eelco Dolstra en su tesis de doctorado (Dolstra, 2006), esta herramienta desea mantener todos los paquetes aislados y funcionales sin importar las dependencias que entre ellos se encuentren.

Una vez más, Nix parte del problema del manejo de programas en versiones distintas e incompatibles entre ellas. El objetivo final de Nix es que se puedan crear paquetes que sean portables y ligeros, que no consuma tantos recursos su creación y que tenga reproductividad (el cambio en el repositorio cuando se actualiza algo).

El sistema Nix se instala fácil: solo con un comando. Tras instalarlo Nix posee en el sistema un directorio (generalmente en */Nix/Store*) donde almacenará todas las aplicaciones que vayan a

instalar, haciendo que todo se encuentre en un solo lugar y los archivos no se mezclen a través de otros directorios.

La "receta" mencionada en otros Manejadores de Paquetes en Nix se le llama Derivación. Una Derivación en Nix es un conjunto de funciones en donde se exponen los distintos parámetros para construir el paquete. Esta derivación es escrita en el propio lenguaje de Nix llamado *Nix Expression* y al final la derivación queda como un paquete con un *hash* criptográfico, como por ejemplo (Nix Package Manager, s.f.):

*/nix/store/b6gvzjyb2pg0kjfwrjmg1vfhh54ad73z-firefox-33.1/*

El usuario puede acceder a este paquete solo por el nombre básico del paquete, por ejemplo, "Firefox". Sin embargo, en las Derivadas sí se debe especificar el valor *hash* criptográfico debido a que debe especificarse una variable con su versión y parámetros específicos (Nix Package Manager, s.f.).

Nix no requiere usuario *root* para la instalación, además una vez creado se puede crear *Profiles* para que cada usuario pueda tener su propio entorno encapsulado. Además de eso, Nix posee un *Garbage Collector* para liberar espacio en disco de las derivaciones que no tengan dependencias y posee una serie de repositorios globales llamados canales (*Channels*) donde existen ya un conjunto de derivaciones "oficiales" (*Nix Channel*). Sin embargo, los usuarios pueden crear sus propios canales y se puede decidir si dejarlo público o privado. Esta característica se usó dentro de las pruebas para analizar el rendimiento de Nix en el GRICAD, donde se usó un canal privado que se encontraban derivaciones propias del sistema que se estaban haciendo las pruebas (Bzeznik, Henriot, Reis, Richard, & Tavard, 2017).

### 3.5 Conclusiones: Características Generales

En este capítulo mostramos las características de Docker, EasyBuild, Spack y Nix. Todas estas herramientas son Manejadores de Paquetes; herramientas que se encargan de encapsular las librerías, binarios y programas en un solo objeto llamado paquete. Cada uno de ellos lo desarrolla de distinta manera: Docker usa Contenedores y un sistema intermedio, EasyBuild separa los programas en módulos usando Lmod, Spack crea un árbol de dependencias para instalar sus programas y Nix encapsula los programas Paquetes y lo modulariza para que otros programas puedan usarlo.

Si bien estas herramientas son capaces de instalar y reproducir paquetes (conjunto de software, librerías y binarios) correctamente, cada una de ellas tiene diferencias importantes en su portabilidad y manejo. Spack principalmente realiza la tarea de instalación correcta de un programa pero no hay un encapsulamiento, solo revisa que no haya incompatibilidades de software, Docker encapsula los paquetes aislándolos y evitando este problema, pero su sistema general tiene una capa de traducción al sistema operativo y además requiere usuario elevado para ejecutar ciertos comandos, EasyBuild encapsula igual pero basados en la separación por memoria y Nix hace un encapsulamiento para aislar paquetes, lo modulariza y además no requiere usuario elevado para su ejecución.

De estas herramientas se escogió Nix debido a las siguientes propiedades:

- Es una herramienta en desarrollo y de fácil instalación y ejecución de comandos.
- No requiere permisos de instalación de usuario elevado (administrador).
- No requiere una extensa búsqueda para revisar incompatibilidades entre paquetes porque entre ellos no se “ven” (aislamiento de paquetes).

- Las derivaciones (la “receta”) son portables, lo que permite la escalabilidad del sistema usando esta herramienta.
- Construye desde el binario, así que si la instalación es correcta no habrá errores futuros.
- A pesar de estar en versión beta para los sistemas embebidos, sus propiedades resultan útiles incluso para estos sistemas.
- Es una herramienta que está en constante mejora, donde en el laboratorio de Grenoble están analizando posibles nuevos desarrollos.

A manera de referencia, la

***Tabla 3*** resume las principales características de los Manejadores de paquetes mencionados en este capítulo, así como sus ventajas y desventajas con respecto a sus pares. Aunque hayan algunas diferencias entre el concepto de manejo de paquetes que ellos resuelven, todos siguen el concepto de los Contenedores (encapsulamiento de librerías).

**Tabla 3**

*Resumen de Manejadores de Paquetes Mencionados en este Capítulo Indicando sus Ventajas y Desventajas*

| | **Docker** | **Easy Build** | **Spack** | **Nix** |
|---|---|---|---|---|
| **Tipo de Aplicación** | Contenedor | Módulos | Resolución de dependencias | Encapsulamiento de paquetes |
| **Resuelve problemas de Dependencia** | Si, Aislamiento de librerías | Si, recetas | Si, con árbol de dependencias | Sí, usando Módulos |

| | | | | |
|---|---|---|---|---|
| **Concepto de Aplicación** | Contenedor (kernel + librerías) | EasyBlock (Automatizar build con "recetas") | Combinatoriedad de paquetes (ver., compilador, etc…) | Paquete (funciones aglomeradas) |
| **Características Restrictivas** | Muy popular entre las empresas | No posee versiones no restrictivas | La resolución de dependencias (+10% costo compilación) | Aprendizaje alto en derivadas |
| **Características Limitantes** | Requiere usuario root para su ejecución | No resuelve dependencias en las subcadenas de paquetes | La fase de concretizar es NP-Completo | No compatible con ciertos S.O. |

Una vez hablado del software que ayuda al manejo en HPC, se debe ahora mencionar el hardware en el que se ejecutará. En el siguiente capítulo se hablará del proceso de revisión de ciertos sistemas embebidos, sus características, ventajas, desventajas y del por qué al final se escoge la NVIDIA Jetson TX1 como sistema embebido para las pruebas y análisis en HPC.

## 4. Integración de los Sistemas Embebidos

Luego de conocer los diferentes tipos de Manejadores de Paquetes, es necesario conocer la infraestructura que se usaría en el proyecto. Para ello se inició con ciertas tarjetas de programación general, esto con el fin de entender el poder o lo que son capaces de hacer estas tarjetas.

A través del capítulo se especificará las tarjetas que probamos, y nuestros resultados, ventajas o desventajas de ellas. Al final del capítulo se indicará qué sistema embebido se usó para hacer las pruebas de rendimiento computacional y energético.

### 4.1 Adapteva Parallella Board

Las tarjetas Parallella de la empresa Adapteva (The Parallella Board, s.f.) son pequeñas tarjetas embebidas del tamaño de una tarjeta de crédito ideales para la gente que desee empezar en

el mundo de la computación paralela. Desde una perspectiva más general, estas tarjetas son buenas para la enseñanza de la computación paralela debido a su fácil configuración y su infraestructura.

Consta de un procesador Zynq-Z7010 Dual core ARM A9 y un co-procesador Epiphany de 16 cores. Tiene 1GB de memoria RAM, *slot* para tarjeta MicroSD, conexión USB 2.0, ethernet y soporta Linux como sistema operativo (probamos ubuntu) (The Parallella Board, s.f.).

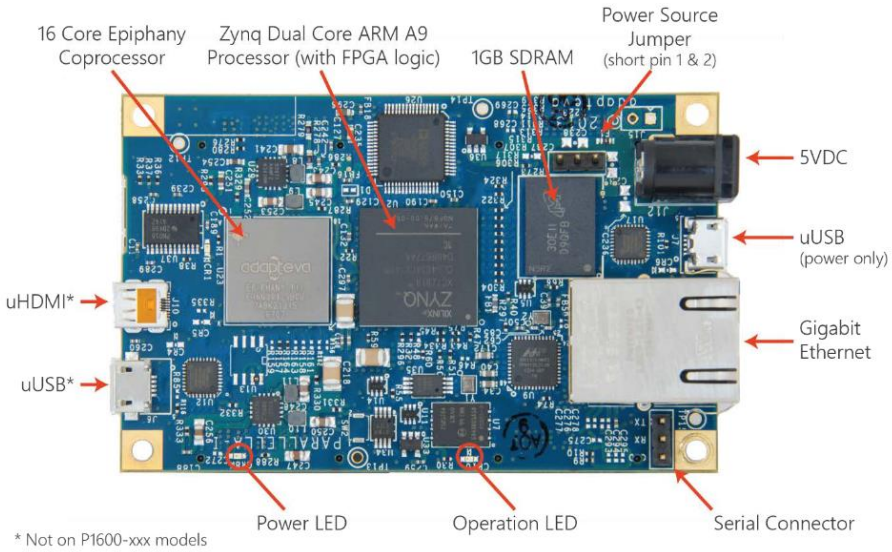
Al momento de usarla nos dimos cuenta de ciertos aspectos. Los procesadores se sobrecalientan mucho, así que, al hacer operaciones muy pesadas, el fabricante indica que requiere un ventilador o algún objeto para mantenerlo frio. En el caso del proyecto, se utilizó dentro de un entorno frío (laboratorio con aire acondicionado) y luego se usó un ventilador al tenerlo fuera del laboratorio.

En forma general se instaló una imagen del sistema operativo en la tarjeta SD y luego se pudo instalar y probar Python y OpenMPI/MPI para uso paralelo con los ejemplos del paquete.

La *Parallella* es un buen sistema integrado que tiene múltiples aplicaciones. Desde el punto de vista del proyecto es ideal para operaciones de alto rendimiento debido a su posibilidad de realizar operaciones paralelas. Ésta gran característica permite que laboratorios pequeños puedan realizar pruebas de sus operaciones o que los colegios puedan acceder al aprendizaje práctico de la computación paralela usando un dispositivo con un bajo costo.

**Figura 6**

*Tarjeta Parallella de la Empresa Adapteva con sus Principales Componentes*




*Nota.* Tomado de https://www.parallella.org/board/.

No obstante, y a pesar de que no es una desventaja en sí sino cuando se compara con otros sistemas, a falta de GPU no se puede acelerar mucho los procesos y a pesar de que el consumo energético sea poco con respecto a otros sistemas de procesadores, siguen siendo procesadores que gastan un mayor consumo que una GPU.

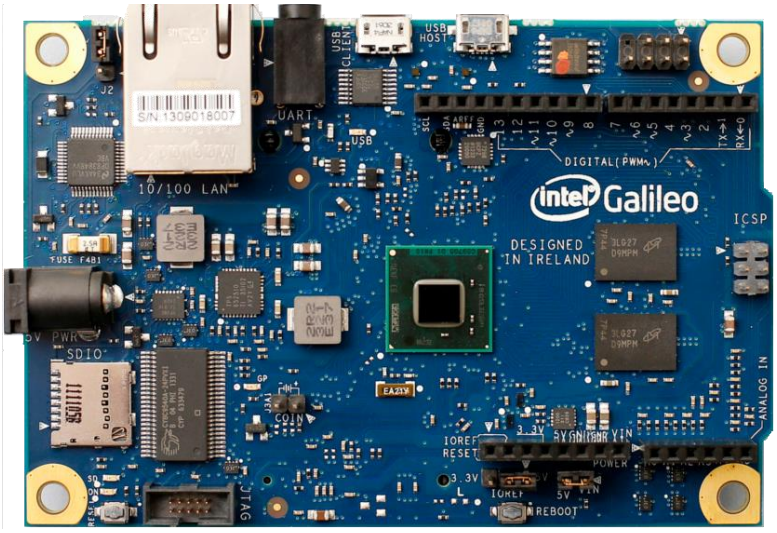
Además de lo anterior, si los programas requieren escalarse no va a ser tan sencillo y tendrá una limitante: su memoria. Las tarjetas SD son la única fuente de memoria que tiene la tarjeta, incluso allí es donde se quema la imagen del sistema operativo con todos sus archivos. Por consiguiente, usarlo para procesos de HPC en donde los datos sean muy extensos será una gran limitante para que el sistema funcione. No obstante, cabe volver a resaltar que para otras operaciones computacionales este sistema es capaz de realizarlo.

### 4.2 Intel Galileo Board (Gen 1)

De la página oficial de Arduino, la Galileo es un microcontrolador basado en el procesador Intel Quark SoC X1000 de 32-bits (Intel Galileo (Arduino web page), s.f.). Es una tarjeta usada para propósito general que tiene funcionalidades expansivas gracias a que tiene entradas que posee Arduino (por eso es certificada por Arduino). Por lo anterior entonces se conoce que esta tarjeta abarca aplicaciones generales y algoritmos electrónicos (aquellos que se realizan con tarjetas como el Arduino).

**Figura 7**

*Vista Superior de la Intel Galileo Gen 1*




*Nota.* Tomado de: https://en.wikipedia.org/wiki/Intel_Galileo.

Como se observa en la Figura 7

**Figura 7**, la Intel Galileo se compone de un procesador, conexión ethernet y un sin número de pines de tipo Arduino. Tiene conexiones USB para depuración, funciona con 5V de energía y tiene entrada para memoria externa micro SD hasta de 32 Gb.

Las ventajas de esta tarjeta es otra vez el consumo energético y su extensibilidad de funcionalidad para otro tipo de operaciones. Sin embargo, para *High Performance Computing*

*(HPC)*, esta tarjeta carece de potencia suficiente para realizar estas operaciones; no está diseñada para este tipo de problemas, por lo que no soporta siquiera operaciones paralelas.

Además de lo anterior, se quiso usar un mini clúster de estos sistemas con el fin de explotar su bajo consumo energético, pero la Generación 1 de Intel Galileo (la que está en la

Figura **7**), no soporta ser un nodo esclavo, por lo que no existe manera de usarlo en sistemas distribuidos.

Sin embargo, la generación 2 sí se podía usar como un clúster por su soporte a ser nodo esclavo. Existe una propuesta que logró conectarlos y hacer pruebas en ellos como sistema distribuido, pero basados en herramientas como Hadoop ((UC-Lab), s.f.).

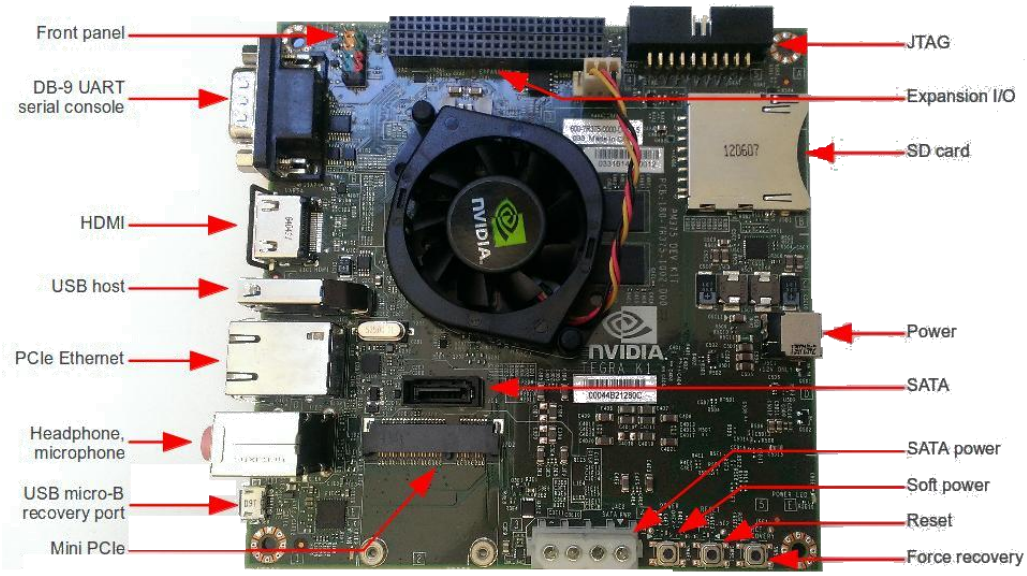
Al iniciar este proyecto, la Galileo Generación 1 ya se encontraba descontinuada, y la Generación 2 se descontinuó a mediados del 2017 (Intel Galileo Gen 2 Board (Specs), s.f.), dejando sin salida el uso de este sistema.

### 4.3 NVIDIA Jetson TK1

La NVIDIA Jetson TK1 (Jetson TK1, s.f.), es un sistema embebido de propósito general cuyo principal objetivo es explotar las capacidades de cómputo de sus 192 CUDA Cores que posee. Este sistema embebido tiene la idea de poder usar algoritmos paralelos y de alto rendimiento, explotando sus GPUs y consumiendo menos energía.

**Figura 8**

*NVIDIA Jetson TK1 Mostrando sus Principales Componentes*




*Nota.* Tomado y modificado de https://elinux.org/Jetson_TK1.

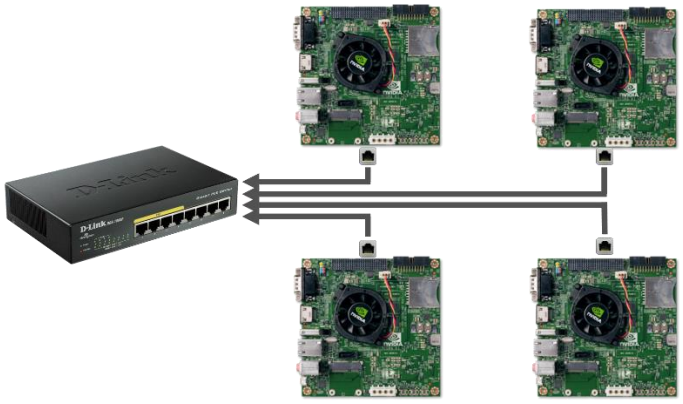
La Jetson TK1 consta de una Tegra K1 SoC (de ahí el nombre) con un procesador NVIDIA ARM Cortex-A15 Quad-core y NVIDIA Kepler GPU con 192 *CUDA Cores*. Posee una memoria eMMC de 16GB, una memoria RAM de 2GB, puerto HDMI, conexión ethernet, slot para tarjeta SD, puerto USB para teclado y ratón (uno solo) y entrada serial para otro tipo de conexiones (cuando falla la USB, por ejemplo).

Las nuevas características de esta arquitectura son la *Hyper-Q* que hace que el GPU trate de estar ocupado siempre que pueda y *Dynamic Paralelism* que permite que el GPU cree nuevas tareas, aliviando al procesador de crearlas.

En el proyecto, se quemó el sistema operativo usando *Nvidia Linux 4Tegra (L4T) Board Support Package* (BSP), y *Sample Root Filesystem from Nvidia*. Este sistema viene con Ubuntu pero tiene todas las librerías y programas para realizar algoritmos paralelos, distribuidos y el uso de las GPU con CUDA.

**Figura 9**

*Infraestructura de los 4 Jetson TK1 Hechos como Prototipo de Pruebas*



Con respecto a su infraestructura, tal como lo muestra la Figura 9, se usaron 4 Nvidia Jetson TK1 conectados por cable ethernet a un pequeño *router* (distinto a la de la imagen). Cada uno de ellos se le instaló el L4T, y, aunque se puede instalar fácilmente mediante la herramienta Jetpack de NVIDIA, se requiere usar solamente la distribución Ubuntu para poder usarla, así que se requirió hacerlo de otro modo.

Cada TK1 se revisó que funcionara las librerías de CUDA y de MPI. Para ello, luego de instalar CUDA (que a propósito también tuvo problemas en su instalación al no tener la Jetpack) se usaron los algoritmos de prueba que vienen con la distribución para revisar su funcionamiento, y luego de instalar MPI se siguió un tutorial para configurar correctamente el sistema y probarlo con el mismo algoritmo de prueba que en ella se encuentra (MPI Tutorial, s.f.).

El proceso de su instalación y acople fue exitoso. No obstante, tardó demasiado tiempo acostumbrarse y encontrar soluciones a los problemas, y todo esto debido a su madurez del sistema (es el primero de este tipo de NVIDIA, por lo que los errores y compatibilidades se estaban apenas analizando). Luego del éxito usando librerías tradicionales se pasó a empaquetarlas en Nix y lastimosamente la versión de Nix para ARM no era compatible para este sistema embebido y sin

forma alguna de instalarlo se decidió probarlo con NixOS, teniendo un resultado peor ya que requería el uso de un conector Serial para poder comunicarse con el sistema, debido a que ningún otro puerto era compatible (y habría que volverlos compatibles a mano, decidiendo que esta tarea estaba lejos de los objetivos del proyecto).

Estos sistemas son fácilmente escalables, realizan paralelismo y además explotan las capacidades de la GPU. También efectúan operaciones con bajo consumo de energía por lo que era una buena candidata para el análisis de un sistema para HPC usando manejadores de paquetes.

La desventaja de este sistema fue su tiempo. Al día de hoy la tarjeta ya dejó de producirse, por lo tanto ya se encuentra obsoleta. Esto repercute en las aplicaciones y librerías que ya no se actualizan con soporte a este tipo de sistemas sino a las versiones posteriores. Esto se vio reflejado en el uso de NIX dentro de este sistema, que al inicio del levantamiento de la infraestructura no era compatible por estar en versión beta el soporte a sistemas embebidos, pero más adelante el soporte se incluyó para las Jetson TX1 y TX2. En los capítulos siguientes se ampliará con detalle este proceso y sus alti-bajos.

### 4.4 NVIDIA Jetson TX1

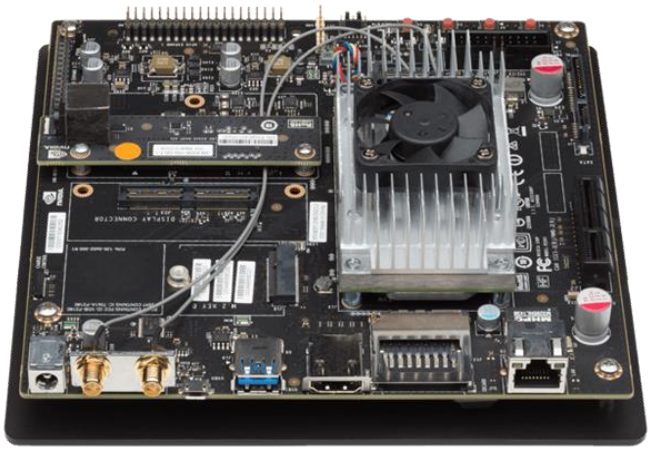
La NVIDIA Jetson TX1 (o TX1 para abreviar), es un sistema embebido desarrollado por NVIDIA que es la versión siguiente a la Jetson TK1. Posee la arquitectura Maxwell con 256 *CUDA Cores*, un procesador Quad ARM A57 con 2MB de cache, 16GB de memoria eMMC, entrada SDIO y SATA, conexión USB 3.0 y 2.0, conectividad LAN, WLAN y bluetooth, además de que reproduce videos hasta 4K.

Este sistema embebido mejora su versión de la TK1 de gran manera gracias a su nueva arquitectura, su nuevo procesador y la nueva cantidad de GPUs. Está diseñado para ejecutar

eficientemente programas de *Deep Learning*, visión por computadora, *GPU Computing* y gráficas, siendo ideal para Inteligencia Artificial (Nvidia Jetson Systems, s.f.). De hecho, la publicidad sobre este sistema ha sido su gran capacidad de cálculo en AI (*Artificial Intelligence*).

**Figura 10**

*NVIDIA Jetson TX1*



*Nota.* Es la sucesora de la Jetson TK1 con arquitectura Maxwell y 256 CUDA Cores. Imagen tomada de https://www.amazon.in/NVIDIA-Jetson-TX1-Development-Kit/dp/B017NWO6LG.

Como la Jetson TX1 es el sistema embebido que tiene mejores especificaciones que cualquier otro sistema probado, se decide hacer con este sistema las respectivas pruebas de rendimiento. Además de tener mejores características, se tenían a disposición más sistemas embebidos en el Laboratorio de Grenoble, por lo que las pruebas de escalabilidad funcionarían mejor entre más sistemas se tuvieran.

Se usaron 8 NVIDIA Tegra TX1 interconectadas con un *switch* y dejando un portátil como sistema maestro. Cada uno de ellos se instaló las librerías CUDA y MPI, probándolas por separado y luego en conjunto como un sistema distribuido. Cada prueba se basó en los ejemplos que vienen por defecto en las instalaciones de CUDA y de un tutorial de MPI que incluso mostraba cómo configurar el sistema (MPI Tutorial, s.f.).

La Tabla 4 resume en gran parte los distintos sistemas embebidos mencionados en este capítulo. Aunque se menciona la Jetson TX1 como el sistema embebido más actualizado, en el momento de hacer este proyecto ya existía el NVIDIA Jetson TX2 que posee, además de las características de la Jetson TX1, un nuevo procesador NVIDIA Denver2 *dual core*, arquitectura Pascal con 256 *CUDA Cores*, 8GB de memoria, 32 GB de disco con menos uso energético promedio: 7.5W (contra 10W de la Jetson TX1). Este sistema no se puso probar nada de él debido a su indisponibilidad en los laboratorios en donde se desarrolló el proyecto.

**Tabla 4**

*Comparación de los Sistemas Embebidos Mostrados en este Capítulo*

| | **Parallella** | **Intel Galileo** | **Jetson TK1** | **Jetson TX1** |
|---|---|---|---|---|
| **Procesador** | Zynq-Z7010 Dual core ARM A9 + Epiphany 16 cores | Intel Quark SoC X1000 de 32 bits | NVIDIA ARM Cortex-A15 Quad-core | NVIDIA Quad ARM A57 |
| **GPU** | No tiene | No tiene | NVIDIA Kepler GPU con 192 CUDA Cores | NVIDIA Maxwell GPU con 256 CUDA Cores |
| **RAM** | 1 GB | 256 MB | 2 GB + eMMC 16GB | 4 GB + eMMC 16 GB |
| **USB** | 2.0 | 2.0 | 2.0 | 2.0, 3.0 |
| **SD Slot** | Si | Hasta 32 GB | <= 2TB* (UHS-1 U3) | <= 2TB (UHS-1 class 10) |
| **Conex. Inalambrica** | LAN | LAN | LAN | LAN, WLAN, Bluetooth |
| **S.O.** | Linux Ubuntu | Linux Yocto | L4T (Ubuntu 14.04) | L4T (Ubuntu 16.04) |
| **Herra. Paralelas** | MPI, OpenMP | MPI, OpenMP | MPI, OpenMP, CUDA,… | MPI, OpenMP, CUDA,… |
| **Consumo Energético** | 10 W | 15 W | 12 W | 10 W |
| **Otras Conexiones** | No | No | Serial Console, HDMI | SDIO, SATA |
| **Limitaciones** | No tiene GPU. Poca memoria (sin expansión). | No tiene GPU No soporta uso como nodo esclavo | Incompatibilidad con NIX. Sistema obsoleto (EOL). | Arquitectura reemplazada por Pascal. Hoy en día existe una nueva generación (TX2). |

### 4.5 En Síntesis: La Elección de la NVIDIA Jetson TX1

En este capítulo se hizo un análisis de ciertos sistemas embebidos que, sin que logren mostrar todo el abanico de este tipo de sistemas, sí muestran en su gran mayoría una arquitectura similar y un funcionamiento común. Se probó sistemas embebidos sin GPU integrado a sus sistemas (Parallella o Galileo Gen 1) y también la gama de sistemas embebidos donde explotan la capacidad de la GPU (NVIDIA Tegras).

En el transcurso de las pruebas de tarjetas, nos dimos cuenta que la Intel Galileo Gen 1 no soportaba la realización de sistemas distribuidos por no soportar ser nodo esclavo. Este inconveniente, y sumado a que estaba descontinuado ya a la fecha del inicio del proyecto se decidió dejar de probarlo para HPC. Un sistema parecido, la Parallella de Adapteva, se veía como muy buena candidata dado a que soportaba paralelismo en sus distintos *cores*, sin embargo, al no tener GPU, haría que su desempeño se viera considerablemente impactado al compararlo con otros sistemas que sí tuvieran estos GPUs.

Por tal razón se decidió ir por las NVIDIA Jetson. NVIDIA había sacado por fin sus sistemas embebidos donde, además de tener bajo consumo energético como las anteriores, poseía una arquitectura con GPU Cores que le permitirían alcanzar desempeños mucho mejores que las otras. La NVIDIA Jetson TK1 fue el primer sistema embebido creado por NVIDIA y con 196 CUDA Cores y la arquitectura Pascal, tenía buenos desempeños en computación paralela y distribuida, necesarios para el HPC. Sin embargo, no se contaba que el manejador de paquetes Nix no pudiera ser instalado de forma nativa sino de forma de prueba y error y hasta usando NixOS de forma beta.

Más adelante, y gracias al laboratorio de Grenoble, se pudo probar la NVIDIA Jetson TX1. Con arquitectura Maxwell y 256 Cuda Cores, este sistema embebido se convertiría en el mejor candidato para las pruebas en el proyecto, y de hecho fue el seleccionado para dichas pruebas.

Se seleccionó a la NVIDIA Jetson TX1 como sistema embebido para analizar en esta propuesta debido a su gran capacidad de cálculo, la nueva arquitectura permitiría un mejor desempeño con respecto a su antecesora y además se disponían muchos más nodos, sin mencionar que instaló sin problemas Nix.

Finalmente, cabe destacar que en la realización del proyecto ya existía otro sistema embebido NVIDIA llamado NVIDIA Jetson TX2, y aunque tuviera mejor desempeño debido a su nueva arquitectura (Maxwell y un co-procesador), en el momento de la realización de esta propuesta no se tenía disponible ninguna tarjeta de estas, esperando que como futuros proyectos se puedan hacer las respectivas pruebas. En la Tabla 5 se observa la comparación entre ambas Jetson, mostrando generalmente las diferencias más significativas entre ellas como lo es el procesador extra, la nueva arquitectura de GPU, la memoria RAM, la memoria interna y su consumo energético.

**Tabla 5**

*Comparación Entre las Jetson TX1 y TX2 con sus Elementos más Significativos que los Diferencian*

| Jetson TX1 | Jetson TX2 |
|---|---|
| ARM Cortex-A57 (quad core) @1,73GHz | ARM Cortex-A57 (quad core) @2GHz + NVIDIA Denver2 (dual core) @2GHz |
| 256-core Maxwell @998MHz | 256-core Pascal @1300MHz |
| 4GB 64-bit LPDDR4 | 8GB 128-bit LPDDR4 |
| 16GB eMMC | 32GB eMMC |
| Not supported CAN bus controller | Dual CAN bus controller |
| USB 3.0 + USB 2.0 | USB 3.0 + USB 2.0 |
| 10 watts | 7,5 watts |

Al observar ambas arquitecturas en la Tabla 5 se aprecia una mejora más en el anexo de un co-procesador ARM y además de una actualización de generación arquitectura, lo que le da a la Jetson TX2 un poder de cómputo más grande que la Jetson TX1 y a menor costo energético. Esto implica que el solo cambio del diseño del sistema puede impactar enormemente con su desempeño.

En el siguiente capítulo se detallará más el proceso que se realizó de la conexión y las pruebas de la Jetson TX1, además de su implementación con Nix, el manejador de paquetes funcional escogido en el capítulo anterior.

## 5. Levantamiento y Pruebas de la Infraestructura

Después de haber revisado algunos sistemas embebidos (Adapteva Parallella, Intel Galileo Gen 1, Nvidia Jetson TK1 y Nvidia Jetson TX1), y luego de haber conocido sus características

más sobresalientes y sus limitaciones, se escogió por realizar toda la infraestructura en el sistema embebido que tenía las mejores capacidades para el cómputo en alto rendimiento: La NVIDIA Jetson TX1.

En colaboración con la Universidad de Grenoble (Université Grenoble Alpes) y el laboratorio LIG en Francia, se pudo levantar la infraestructura para realizar las respectivas pruebas del sistema. En este capítulo se explicará con detalle las características de dicha infraestructura y la inclusión del manejador de paquetes en ella, indicando también las ventajas y desventajas del uso de esta herramienta.

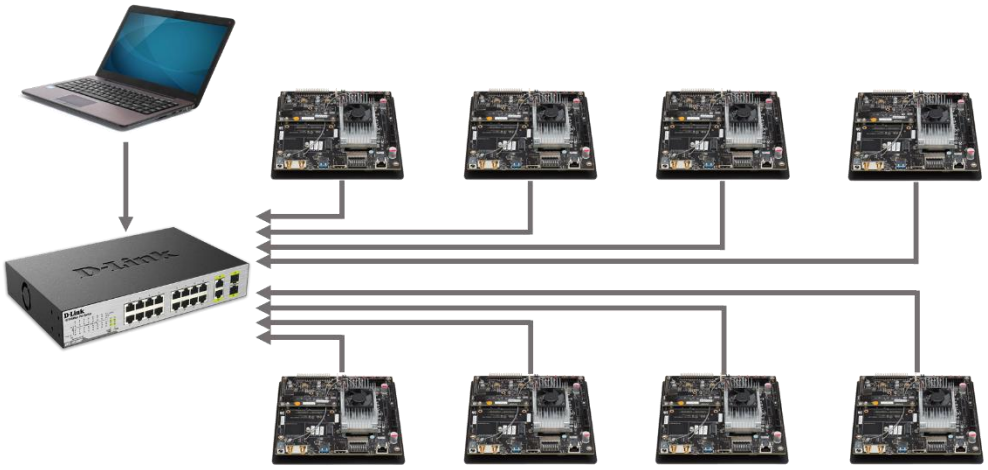
## 5.1 Infraestructura de NVIDIA Jetson TX1

Una vez escogido como sistema embebido la NVIDIA Jetson TX1 para realizar las pruebas de rendimiento computacional y energético y Nix como manejador de paquetes funcional para encapsular los programas, se procedió a levantar la infraestructura de dicho sistema.

La infraestructura consta de 8 NVIDIA Jetson TX1, donde para fines prácticos cada uno lo llamaremos un Nodo. Cada uno de estos nodos estaba conectado directamente a un *switch* a través de un cable UTP tradicional. Un PC, en este caso de tipo portátil se conectó al *switch* para ser el Nodo Maestro. El portátil tenía como sistema operativo Debian. Un esquema gráfico resumido del sistema en general se muestra en la Figura 11:

**Figura 11**

*Esquema de la Infraestructura de las NVIDIA Jetson TX1*



El Nodo Maestro (el portátil) se conecta a los demás Nodos por medio del *switch*. Para poder lograr lo anterior se requirió primero probar que cada nodo que estuviera funcionando correctamente y por aparte. Para ello se instaló el sistema operativo con las mismas condiciones en cada nodo ejecutando los comandos correspondientes, luego se prueba su conexión a la red y al Nodo Maestro y luego se prueba instalando las otras librerías como OpenMPI y CUDA.

Como se mencionó en capítulos anteriores, CUDA posee un problema de instalación. Para poder instalar CUDA de forma casi transparente se requiere una herramienta de NVIDIA llamada Jetpack en la versión que soporte la NVIDIA Jetson que se vaya a usar (TX1 en nuestro caso). Lastimosamente la herramienta Jetpack solo se puede instalar si se tiene el sistema operativo Ubuntu 14.06. Afortunadamente existe en los foros una forma alterna de realizar esta instalación saltándose la Jetpack, sin embargo NVIDIA recomienda hacerlo por ella (Devtalk nvidia (foro nvidia), s.f.).

Luego de la instalación y prueba de los ejemplos de CUDA, se dispone a realizar la configuración de la red de los Nodos. Para ello se sigue la guía de OpenMPI (MPI Tutorial, s.f.)

para que cada nodo pueda conectarse y “verse” a los demás. Además de lo anterior, se configuró los nodos para que pudieran conectarse entre ellos sin clave usando otra guía para tal fin (MPI Tutorial: Running an MPI Cluster within a LAN, s.f.).

Una vez que los Nodos están conectados, con las librerías instaladas y que se puedan comunicar entre sí, se procedió a hacer las pruebas de rendimiento, primero sin ninguna herramienta extra y luego con Nix, el Manejador de Paquetes Funcional. Esto se realizó de esta manera para poder comparar de primera mano los escenarios de construcción, compilación y ejecución de los *Benchmarks* de prueba de este sistema.

## 5.2 Benchmarks: La Prueba de Rendimiento Computacional

Para poder probar el rendimiento del sistema es necesario usar alguna herramienta o un programa que permita revisar esta capacidad del sistema, además de ello, al ser un programa también se encapsularía por lo que al final las pruebas de uso igual funcionarían.

La mejor forma de probar el rendimiento de un sistema es utilizando alguna herramienta o programa que pueda medir y exigir al máximo el sistema. Para el caso del proyecto se usan los *Benchmarks*, una serie de programas (una *suite*) que tiene como objetivo probar el sistema de distintas maneras. Aunque existen un incontable número de *benchmarks* y una amplia discusión de si estos son o no válidos en el día de hoy, se utilizará los *benchmarks* en los que se prueban y clasifican los supercomputadores en el mundo: *Stream Benchmark* y *High Performance Linpack Benchmark.*

### *5.2.1 STREAM Benchmark*

El *Stream Benchmark* (STREAM benchmark (web page), s.f.), es un *benchmark* que se encarga de medir el ancho de banda (en Mb/s) de un sistema a través de operaciones simples, pero con un gran tamaño. Las operaciones que se usan son las siguientes (McCalpin, 1995):

$$COPY \quad A[i] = B[i]$$

$$SCALE \quad A[i] = k * B[i] \quad ; \quad k \in \mathbb{R}$$

$$ADD \quad A[i] = B[i] + C[i]$$

$$TRIAD \quad A[i] = B[i] + k * C[i] \quad ; \quad k \in \mathbb{R}$$

Este *benchmark* se usa para medir la transferencia de los datos, específicamente del procesador con las distintas memorias, principalmente con el disco duro. La idea es tener una gran cantidad de operaciones, se aconseja que sea 4 veces mayor que el tamaño total de las memorias o 1 millón de operaciones, lo que sea más grande.

Existen dos formas de ejecución: Lineal y Paralelo. La forma Lineal es la más sencilla de todas, hay que tener cuidado con la memoria caché y los datos no deben ser *cacheables*. Esta forma es para cuando se tiene un solo Nodo. La forma paralela es menos sencilla de ejecutar debido al uso de múltiples librerías (para la comunicación entre nodos), siendo la ideal para probar múltiples Nodos.

### *5.2.2 High performance Linpack (HPL)*

El benchmark HPL (*High Performance Linpack*) (Dongarra, Luszczek, & Petitet, 2003), se encarga de realizar y medir el rendimiento de un sistema basado en operaciones que involucran la resolución de una matriz. Inicialmente fue diseñado para revisar errores en los llamados de las operaciones de Linpack y tiempos de ejecución, luego empezaron a hacer *suites* completas para

que pudiera funcionar con cualquier computador. Actualmente es el indicador de rendimiento que usan para probar la eficiencia de un supercomputador, aunque poco a poco se están complementando con otros *Benchmarks*, incluidos los de energía.

El programa se encarga de resolver un sistema de ecuaciones de una matriz densa (sin muchos valores nulos) de 64-bit utilizando operaciones lineales y en paralelo, según el sistema pueda realizarlo. La matriz es la siguiente:

$$Ax = b$$

Siendo A una matriz nxn, x un vector de n elementos y b otro vector de n elementos. El algoritmo computa el LU siendo $LU = A$ y hace los cálculos correspondientes para llegar al resultado. El algoritmo es de orden $O(n^3)$, más exactamente:

$$O(n^3) \Rightarrow \frac{2}{3}n^3 + 2n^2 + O(n)$$

operaciones de punto flotante de sumas y multiplicaciones.

Para este *Benchmark* se requiere algunas librerías especializadas. En nuestro diseño se usaron las librerías *MPICH v2* (MPI), *gfortran* (Fortran), *CBLAS* (BLAS Library) y CUDA. Una vez instalado estas librerías se debe modificar un archivo llamado *HPL.dat*, que es aquel donde se especifica las rutas de estas librerías y los valores de entrada que van a tener nuestras pruebas.

Algunos valores para tener en cuenta son N, NB, P y Q. N representa el tamaño del problema, depende de la memoria total del sistema y más o menos se rige por la siguiente fórmula:

$$N = \left( \sqrt{\frac{Tamaño\ Memoria\ (Bytes)}{8}} \right) * (\%\ Memoria)$$

Donde el dividendo “8” se refiere a que se está manejando valores de doble precisión (8 Bytes) y el “% de memoria” debe estar entre 80%-90% (0.8 – 0.9).

NB se refiere al tamaño del bloque del *Grid*, generalmente es una lista de múltiplos de 8 que inicia en 96 y termina en 256. Una optimización de N depende del valor de NB, de tal manera de que estén alineados de la siguiente manera:

$$NB = [96,104,112,120, \dots ,224, \dots ,256]$$

$$N_{Optimizado} \cong \left(\left\lfloor \frac{\lfloor N \rfloor}{NB} \right\rfloor\right)(N)$$

Finalmente, los valores P y Q indican el tamaño del *Grid* y dependen de los procesadores totales. Estas variables multiplicadas deben indicar el total de los procesadores del sistema, incluyendo todos los nodos, además P debe ser ligeramente menor o igual a Q.

$$P * Q = No. Procesadores\ sistema$$

$$No. Procesadores\ sistema = (No. Nodos)\left(\frac{No. Procesadores}{Nodos}\right)$$

$$P \approx Q\ ; P < Q\ (Muy\ poco)$$

Los resultados muestran, el desempeño máximo que alcanzó la máquina realizando algunas operaciones del problema (Rmax medido en GFlops/s), que puede compararse con el rendimiento teórico del sistema (Rpeak) para dilucidar cuanto se está perdiendo de rendimiento en el problema.

Los valores exactos de entrada y de salida que se obtuvieron en este proceso, se especificarán más detalladamente en el capítulo de los resultados.

### *5.2.3 Jacobi Benchmark.*

El Jacobi Benchmark (Tamuli, Debnath, Majumdar, & Ray, 2015), es una implementación en paralelo del método de Jacobi para resolver sistemas de ecuaciones lineales. Una vez más utiliza la ecuación de matrices:

$$Ax = b$$

Y con ello se reescribe para poder encontrar la variable x según sus valores anteriores, volviéndolo un algoritmo de relajamiento, de la siguiente manera:

$$x_{k+1} = D^{-1}b + D^{-1}(L + U)x_k$$

Ya de esta forma se tiene una forma discreta para resolver el sistema de ecuaciones. Al igual que los otros benchmarks, el algoritmo contiene operaciones de cuenta del rendimiento de los sistemas, y, posee un valor máximo de iteraciones o valor mínimo de error.

Los valores de entrada son esencialmente 2, "-t x y" y "-d num.". Los valores de "-t x y" son obligatorios y expresan el número de procesos en X y Y (filas y columnas) respectivamente. La variable "-d <num>" es opcional, y representa el tamaño del dominio local, si no se ingresa su valor es predeterminado, y en este caso es 4096.

En el caso del proyecto se realizó este Benchmark en los Nodos, iniciando solo en 1 Nodo y luego conectándolo con 2, 4 y 8 Nodos. Al final de tomar los valores se pasó a realizar este mismo Benchmark con el Manejador de Paquetes Nix. Sin embargo, este tema se contará más adelante con un más amplio detalle en el capítulo de Resultados.

### 5.3 Integración con Nix: El Manejador de Paquetes Funcional

Después de realizar todas las pruebas para observar el comportamiento "tradicional" (por así llamarlo) del sistema que se levantó, es necesario ahora probarlo usando el Manejador de Paquetes Funcional Nix. La razón del uso de esta herramienta es su flexibilidad, portabilidad y por ende su escalabilidad con una fácil instalación y configuración del entorno.

Aunque era muy tentador encapsular todos los Benchmarks mencionados anteriormente, y hacerles las pruebas de desempeño, cabe considerar la ventana de tiempo del uso de los Tegra TX1. Por lo consiguiente se decidió encapsular un solo Benchmark, y el Jacobi Benchmark fue el

elegido para su uso por considerarse un Benchmark estandarizado y sencillo de configurar. No obstante, cabe resaltar que aunque se decidió un solo Benchmark, el análisis no se pierde, puesto que esta prueba se puede extrapolar a los otros dominios y dar conclusiones respecto a ello.

Si el software ya está empaquetado y puesto en un canal (ya sea el global o uno privado) simplemente se ejecuta el comando de instalación y ya tiene el software en su sistema:

*Nix-build -A OpenMPI*

Pero si por el contrario (y en nuestro caso) no se encuentra las librerías, tocará anexarlas y encapsularlas primero. La idea general es *derivar* el software. Para lograr este objetivo se usa el Lenguaje Nix dentro de algo que se le conoce como receta. La receta es un conjunto de descripciones que se le indica a Nix cómo debe derivar el software, con qué librerías y en qué versión. Nix luego se encargará de armarlo, empaquetarlo y almacenarlo en el Store Local.

### *5.3.1 Integración Jacobi con Nix*

Como se mencionó anteriormente, el objetivo es hacer el encapsulamiento del *Jacobi Benchmark* con todas las dependencias que requiera, analizando el proceso según su dificultad de empaquetar todo y luego revisar su desempeño computacional y energético.

Lo primero que se realiza es crear el proyecto dentro de *nixpkgs* clonando por wget y dentro de la carpeta se agrega el archivo *default.nix* que va a ser nuestra receta. Las fases principales de Nix al ejecutar la derivación son las siguientes:

1. Unpack Phase: Realiza la descompresión de archivos y entra a su carpeta.
2. Configure Phase: Realiza la operación "*./configure*".
3. Build Phase: Realiza la operación *"Make"*.
4. Install Phase: Realiza la operación "*Make Install*".

Sin embargo existen otras fases, y además de ello se puede especificar qué fases utilizar y si desea hacer algo específico en esa fase (modificar lo que hace la fase). El uso de fases se encuentra dentro del manual de Nix (Nix Package Manager Guide, s.f.).

Los problemas o inconvenientes encontrados fueron en el uso de librerías. Nix no "ve" las librerías nativas de CUDA para hacer el JacobiBench. Para ello se hace un proceso llamado *Dynamic Linker* para que pueda conectarse Nix con esas librerías.

También hubo problemas con el uso de GCC, ya que se requería una versión más antigua que la que el sistema tenía, así que se decidió instalar GCC < 5 en Nix y así Nix derivaría con ese GCC.

Finalmente, el problema más grande que se tuvo fueron las librerías de CUDA que no se encontraban al compilar. Errores como "*libcuda.so.1 not found*" y "*libnvrm_gpu.so not found*" eran muy comunes, por lo que era necesario agregar las condiciones de compilación de "-lcuda" y copiando cada librería que no encontraba en la carpeta "/lib" de CUDA. En general, fueron las siguientes librerías que se tuvo que hacer este proceso:

- *Libcuda.so -> .../lib/stubs -> -L*
- *Libcuda.so.1 -> /usr/lib/aaron64-linux-gnu/tegra/*
- *Libnvrm_gpu.so -> /usr/lib/aaron64-linux-gnu/tegra/*
- *Libnvrm.so -> /usr/lib/aaron64-linux-gnu/tegra/*
- *Libnvidia-fatbinaryloader.so.28.1.0 -> /usr/lib/aaron64-linux-gnu/tegra/ -> -li*
- *Libnvos.so -> /usr/lib/aaron64-linux-gnu/tegra/*

Ya cuando no generaba más errores de compilación, finalmente funcionó el encapsulamiento, por lo que se podía ejecutar de la siguiente manera:

*/mpirun /result/.../bin/jacobi-mpi -t 1 1*

*/nix/store/<hash>/mpirun ./result/.../bin/jacobi-mpi -t 1 1*

En resumen, el software quedó de la siguiente manera:

- *~/nixpkgs/pkgs/misc*
    - */OpenMPI3 (11Kb)*
    - */Jacobi-bench (1.1Gb)*
- *GCC5*

Mientras hace la derivación con pruebas, el sistema de archivos pesa 130Gb aproximadamente, pero luego el *Nix/Store* queda reducido a 21Gb.

El Nix-Store finalmente queda pesando 4.1Gb, y copiándolo a otro Nodo (Tegra TX1) que no tenía esta derivación funciona sin problemas. Sin embargo, esto nos pone a pensar acerca del tamaño en la derivación de los archivos, lo que al final se hizo fue poner un disco que se conecte por NFS y que en este disco se derive y se tenga el *Nix Store* para todos los Nodos.

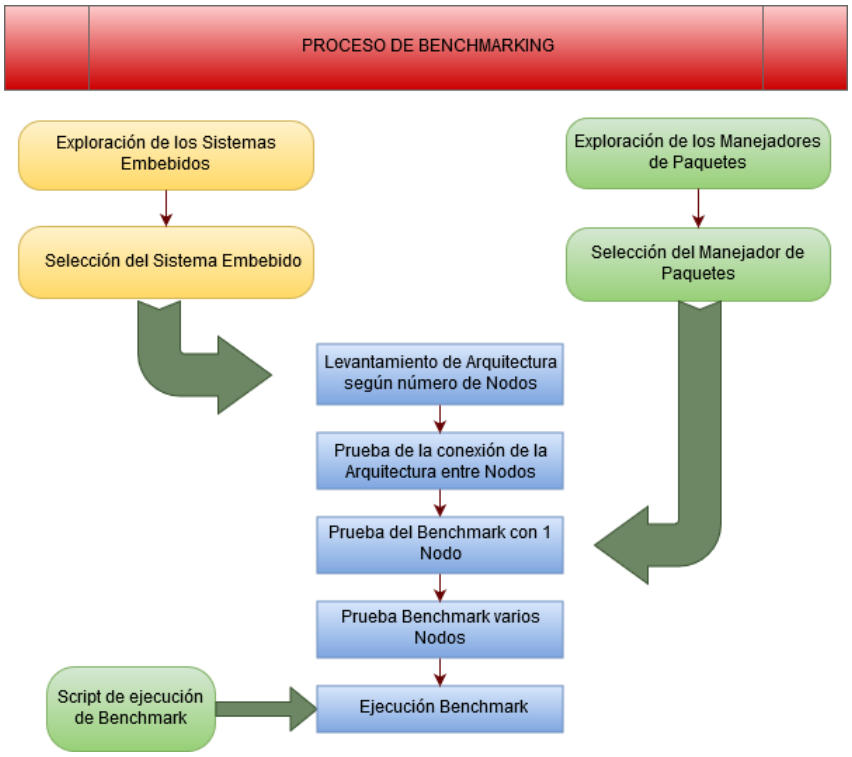
Con el NFS activado y Nix instalado, las ejecuciones de este software funcionan sin problemas, la comunicación de los nodos es correcta con el encapsulamiento y al mismo tiempo éste es portable (e incluso único en el sistema). Lo único que hay que tener en cuenta es tener cuidado con la eliminación de archivos o el montaje de esta unidad, ya que un mal montaje puede eliminar toda la información.

En la

Figura **12** se muestra el proceso general para la realización del benchmark de Jacobi y HPL. Esta imagen representa el paso a paso realizado para poder llegar a hacer los benchmarks y obtener así los resultados.

**Figura 12**

*Paso a Paso del Proceso de Benchmarking*




*Nota.* Esta grafica muestra los pasos realizados para poder ejecutar exitosamente los benchmarks de Jacobi y HPL.

En el siguiente capítulo se indicará los resultados que se obtuvieron en cada prueba, desde los *Benchmarks* en estado tradicional así como usando el Manejador de Paquetes Nix, y a partir de ellos se darán las conclusiones del uso de este tipo de herramienta.

## 6. Resultados de las Pruebas

En el capítulo anterior se especificó las características y el funcionamiento de las pruebas que se realizaron para el análisis de los Sistemas Embebidos. De forma general se usaron principalmente dos: *High Performance Linpack Benchmark* y *Jacobi Benchmark*. Ambos

resuelven un sistema lineal denso que viene dado por matrices. Sin embargo, entregan algunos datos diferentes, que sumadas se pueden dar una idea general de la eficiencia del sistema.

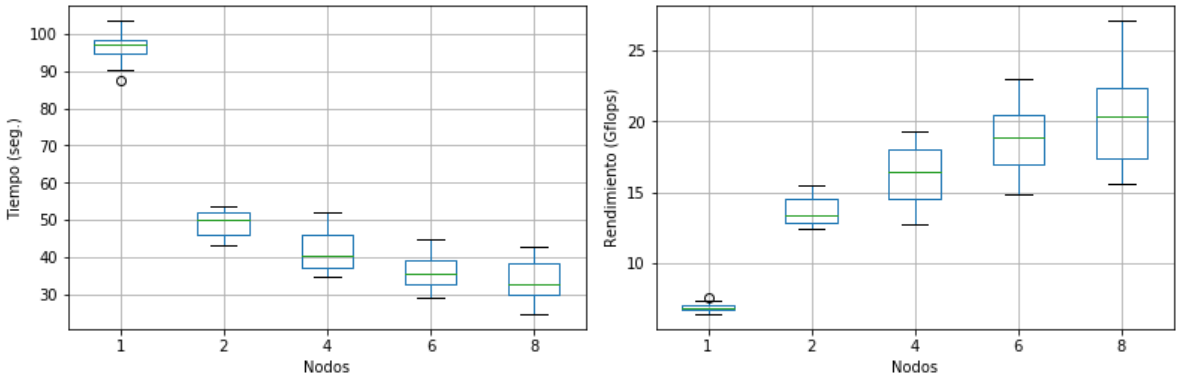
### 6.1 High Performance Linpack

La mayoría del tiempo en este *Benchmark* se usa la operación denominada SAXPY, que consiste en la multiplicación de un escalar a un vector. Esta operación se realiza casi en un 90% del *Benchmark*, lo que indica que esta prueba se enfoca más en el rendimiento del Nodo que de su comunicación.

Al realizar el respectivo *Benchmark* varias veces se obtuvieron unos resultados que se graficaron y se explicarán a continuación:

**Figura 13**

*Diagramas de Caja de los Resultados de HPL Benchmark*




La

Figura ***13*** muestra un diagrama de caja que relaciona la cantidad de Nodos y el tiempo dado en segundos de la prueba de *HPL Benchmark* (lado izquierdo), al igual que el resultado de la eficiencia en GFLOPS. Se puede observar que a medida que va aumentando los nodos, el tiempo gastado va disminuyendo, y de la misma manera el rendimiento va aumentando. Estos resultados

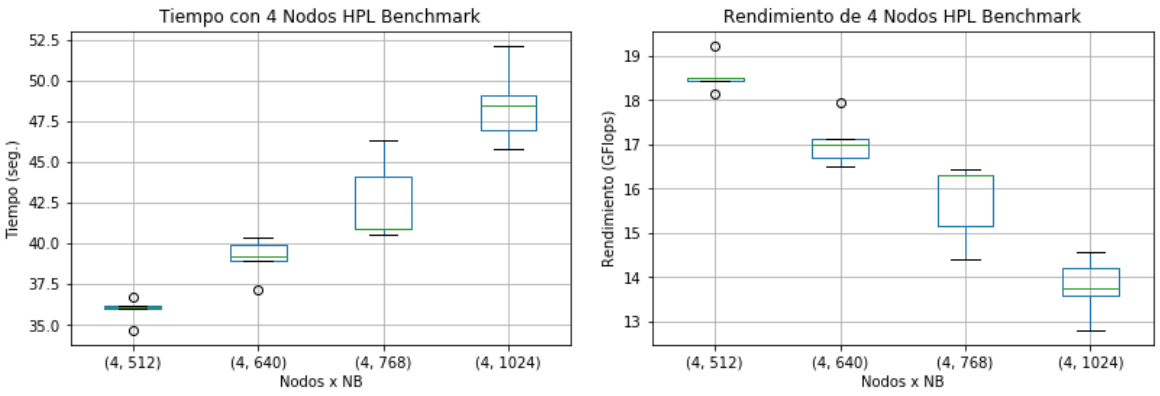
son esperados ya que a mayor cantidad de nodos de una misma cantidad de tareas debe mejorar el tiempo en acabarlo y por ende su razón de cantidad de tareas por tiempo (tareas/tiempo). Además de lo anterior, se puede observar una tendencia asintótica en los resultados que se verán con más detalle en otras gráficas, pero esta tendencia también es esperada debido a la ley de Amdahl.

Una respuesta no esperada es el rango de valores que se obtienen al aumentar el número de nodos. Entre más nodos se hacen las pruebas, aumentan las variables condicionales y los datos obtenidos tienen mayor varianza. De hecho, el valor de la desviación estándar es de 3.45 GFLOPS para 8 Nodos y puede deberse a la comunicación entre Nodos y el uso de la memoria RAM por parte de la GPU en cada Nodo.

Si revisamos los datos según los valores NB, notamos que dichos valores entre más altos sean más aumenta el tiempo y por ende el rendimiento:

**Figura 14**

*Diagrama de Caja de HPL 4 Nodos y Diferentes NB*



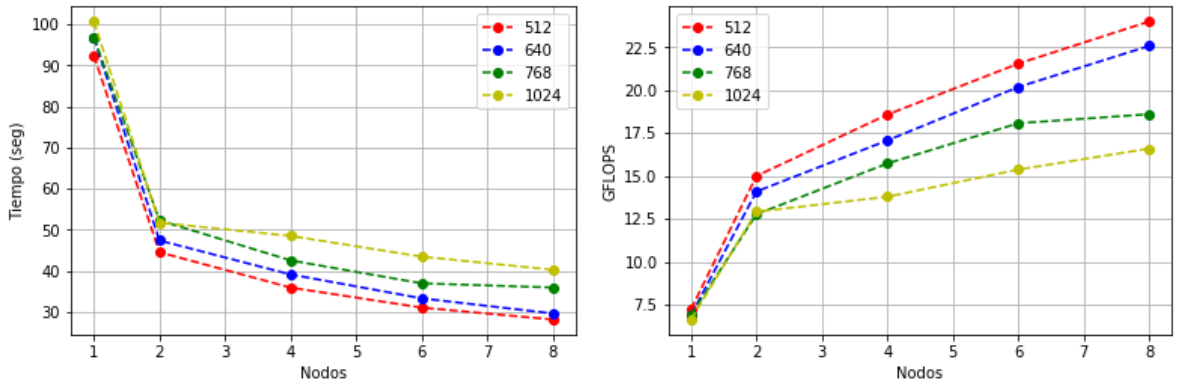

Esto es debido a que al aumentar NB efectivamente debe aumentar el tiempo, ya que son más cálculos que efectúa en la misma cantidad de Nodos. La Figura 14 muestra esta tendencia con las pruebas de 4 Nodos, pero es la misma tendencia en cualquier cantidad de nodos, después de todo es un aumento de tareas por Nodo. No hay que olvidar que igualmente los Nodos tienen un

límite; si les entregan mucha carga de trabajo no podrán realizarlos eficientemente porque deberán pasarlo a disco y si es muy corta entonces se mostraran resultados subestimados. En la Figura 14 derecha se refleja ese aumento del tiempo de ejecución en los resultados de la eficiencia del sistema.

De forma general, en promedio y según los NB y Nodos, se tienen los siguientes resultados mostrados en la Figura 15:

**Figura 15**

*Promedio de Tiempo (izq.) y Rendimiento (der.) de HPL de 1 Hasta 8 Nodos.*




En la Figura 15 se observa en el lado izquierdo, el tiempo promedio del *HPL Benchmark* según cada Nodo y cada NB. Se puede apreciar la disminución general del tiempo a medida que los Nodos aumentan y además de observa que el valor óptimo es NB=512 (línea roja). De la misma manera, se ve los resultados con respecto a la eficiencia en GFlops (lado derecho), donde una vez más NB=512 es el valor óptimo de este sistema. En ambas gráficas se observa que cada vez más la función se acerca a un valor asintótico indicando una disminución de mejora en el rendimiento por cada nodo extra.

Así, en valores numéricos, teniendo en cuenta que cuando NB=512 se obtiene mejores resultados, con esta condición, a 1 Nodo el *Benchmark* se ejecuta a 92.33 segundos con 7,222

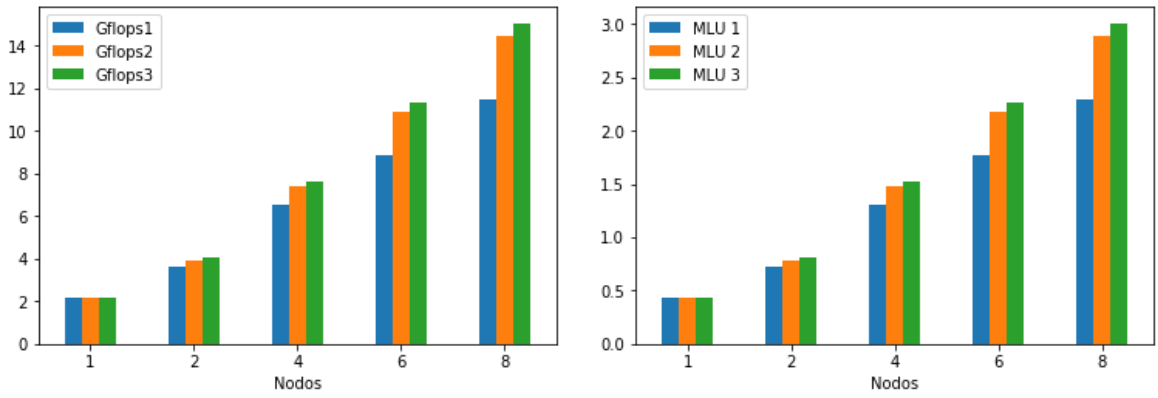
GFlops y con 8 Nodos (todo el clúster), el *Benchmark* termina su tiempo de ejecución en 28,164 segundos a 23,986 GFlops.

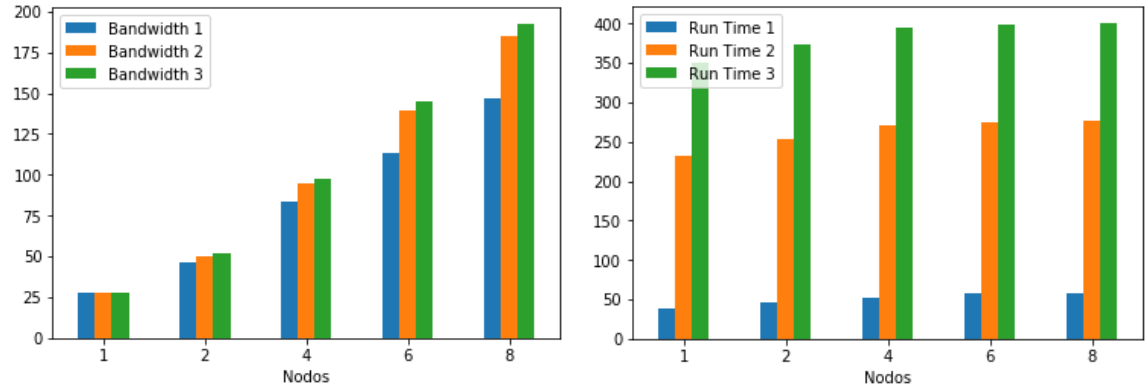
### 6.2 Jacobi Benchmark Results

Este *Benchmark* consiste en un problema de relajación (en cada iteración se obtiene un resultado más preciso) de una ecuación de calor con valores iniciales (límites en el área). Esto lo resuelven calculando el valor de un punto dependiendo de los 4 puntos circundantes. Por lo anterior, se puede observar un nivel de paralelismo, pero no tan amplio como el *HPL Benchmark*.

**Figura 16**

*Resultados de Jacobi Benchmark de 1 a 8 Nodos*






*Nota.* Se muestran los resultados del Benchmark con valores de dimensión de 4096 (azul), 10000 (naranja) y 12288 (verde).

La

Figura **16** muestra los resultados del *Jacobi Benchmark* del sistema de 1 a 8 Nodos y con valores de entrada de d=4096 (Azul), d=10000 (Naranja) y d=12288 (Verde). Éstos valores se usaron por las siguientes razones: la azul se refiere a los valores predeterminados del sistema, la naranja es un valor estándar de referencia (Reza, Tyler, & Sherief, 2017), y el verde es el valor más alto encontrado que no supera la memoria RAM para estos Nodos.

Se observa un crecimiento general según la cantidad de nodos y localmente (por nodos) según el nivel de la variable "d" (dimensión). También se aprecia un crecimiento no asintótico por el momento, indicando que aún puede crecer agregando más nodos. Sin embargo, se nota un tiempo de ejecución muy similar en casi todos los Nodos. Este problema se discutirá más adelante en este capítulo.

Se puede realizar un análisis del comportamiento de los valores especificados anteriormente por nodos y dimensiones de tabla X,Y. Teniendo en cuenta las dimensiones de la tabla de datos para la ejecución, se aprecia un comportamiento equivalente en cada nodo.

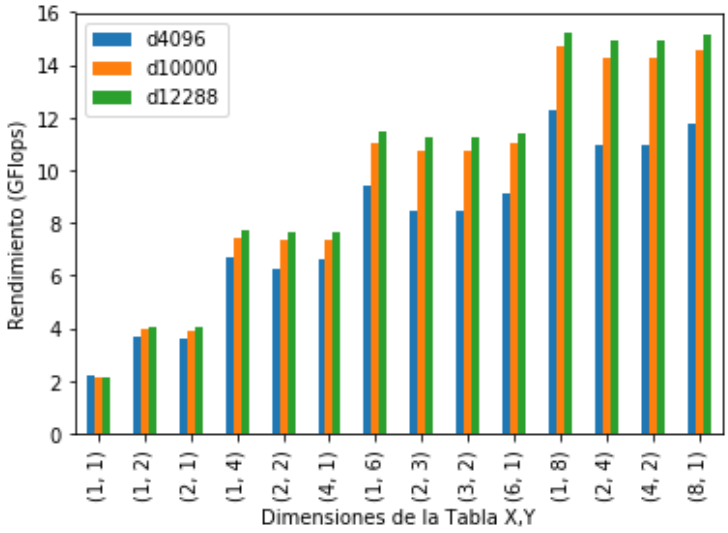
En la **¡Error! La autoreferencia al marcador no es válida.** se observa los resultados agrupados por grupos de Nodos. Cabe resaltar que esta agrupación tiene la condición de que la

multiplicación de los números debe ser igual a la cantidad de Nodos disponibles. Por ejemplo (1,6) representa un resultado con 6 Nodos mientras que (2,4) representa un resultado de 8 nodos.

De la **¡Error! La autoreferencia al marcador no es válida.** se puede analizar que los mejores resultados se encuentran en los extremos de cada grupo, esto significa que entre más separado estén estos valores será mejor el resultado, siendo el mejor de ellos 2 cuando el primer valor es el menor de los 2 (1,x; x es número de Nodos). Este resultado se ve claramente cuando el valor de la dimensión es muy pequeño, pero entre más nos acercamos a la dimensión ideal (cuando "d" llena perfectamente la RAM del Nodo), este resultado se nota menos.

**Figura 17**

*Resultados Detallados por Diferencia en Nodos en Jacobi Benchmark*




Cabe resaltar además que el resultado de los Nodos no aumenta al mismo ritmo de uso de Nodos, igualmente esto es válido sabiendo las implicaciones del paralelismo, uso de RAM y comunicación entre Nodos, aunque al observar la

Figura **16** y En la **¡Error! La autoreferencia al marcador no es válida.** se observa los resultados agrupados por grupos de Nodos. Cabe resaltar que esta agrupación tiene la condición de que la multiplicación de los números debe ser igual a la cantidad de Nodos disponibles. Por

ejemplo (1,6) representa un resultado con 6 Nodos mientras que (2,4) representa un resultado de 8 nodos.

De la **¡Error! La autoreferencia al marcador no es válida.** se puede analizar que los mejores resultados se encuentran en los extremos de cada grupo, esto significa que entre más separado estén estos valores será mejor el resultado, siendo el mejor de ellos 2 cuando el primer valor es el menor de los 2 (1,x; x es número de Nodos). Este resultado se ve claramente cuando el valor de la dimensión es muy pequeño, pero entre más nos acercamos a la dimensión ideal (cuando "d" llena perfectamente la RAM del Nodo), este resultado se nota menos.

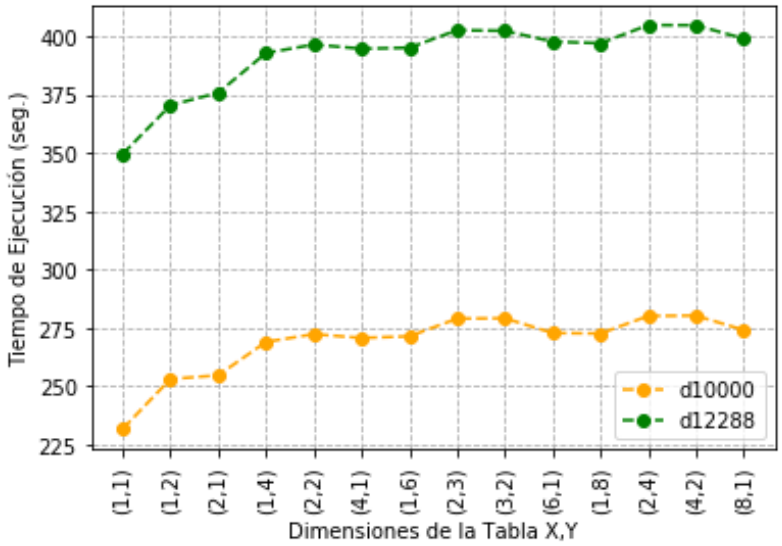
***Figura 17*** la pérdida se empieza a notar desde los 6 Nodos, sin embargo no es muy significativo aún.

Al revisar los resultados con respecto al tiempo total de ejecución, se observa como si los tiempos de ejecución no cambiaran o estaban muy cercanos entre sin importar la cantidad de nodos que se ejecuta el *Benchmark*. Sin embargo, haciendo una revisión más detallada se observa que estos valores sí se encuentran separados, solo que comparándolos con el tiempo de ejecución de la dimensión 4089 no se nota este cambio.

En la Figura 18 se muestra el comportamiento del tiempo de ejecución total del Jacobi Benchmark para la muestra de dimensión de 10000 y 12288. En ella se puede apreciar el crecimiento asintótico en tiempo de ejecución al aumentar los nodos (no olvidar que al multiplicar las dimensiones de Txy da la cantidad de nodos usados), lo que sugiere que llegará un punto en el que el problema se estanque a un número de nodos.

**Figura 18**

*Resultados del Tiempo de Ejecución del Jacobi Benchmark*



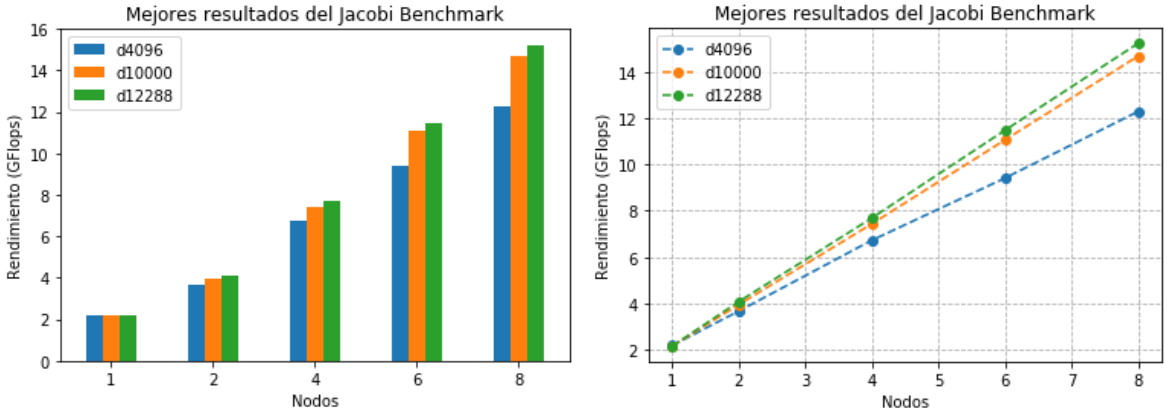

*Nota.* En estos resultados se aprecia un crecimiento que tiende a ser asintótico según el número de nodos que se usen.

En conclusión, el *Jacobi Benchmark* tiene un mejor comportamiento cuando los valores de los nodos están muy separados entre ellos, siendo el mejor de ellos con el valor nodo (1,n) siendo n el número de Nodos. De esta forma, se puede realizar una gráfica con solo estos valores para tener un conocimiento del rendimiento promedio que mejor se ajusta a este *Benchmark* usando este sistema, de la siguiente manera como se observa en la Figura 19.

Las gráficas presentadas en la Figura 19 muestra el mejor rendimiento del sistema para *Jacobi Benchmark*. De esta forma observamos cómo va mejorando por cada 2 nuevos nodos agregados. Del mismo modo se observa que la gráfica va a tender a disminuir el rendimiento, pero igual el rendimiento se comporta con una buena escalabilidad hasta esta cantidad de nodos.

**Figura 19**

*Resultados Jacobi Benchmark de la Forma (1,n) Siendo n el Número de Nodos Usados Para la Prueba*




## 6.3 Jacobi Benchmark (Usando Nix)

Una vez con los resultados del Jacobi Benchmark de forma tradicional, se debe ahora revisar el modelo del uso con el manejador de paquetes Nix. Recordando que la idea del modelo es proveerle al sistema levantado una herramienta que disminuya la capa del *Hypervisor* que tienen las máquinas virtuales en una capa más liviana y además cumpla con las condiciones del problema de la incompatibilidad de aplicaciones mencionadas en el Capítulo 1.

Por consiguiente, se deriva el Benchmark dentro del espacio de la herramienta Nix, creando las recetas necesarias y derivándolo para así obtener el paquete funcional. Una vez realizado esto, se procede a realizar las mismas pruebas con la finalidad de comparar su "gasto computacional extra" con respecto a la versión tradicional.

De la misma forma en que se ejecutó el *Jacobi Benchmark* de su forma tradicional, se ejecuta en el Manejador de Paquetes Nix. Para ello se requirió que todas las dependencias (aplicaciones) entren a la tienda Nix (*Nix Store*) y que se deriven para poder ejecutarlas en este ambiente. De lo anterior se da un comando de ejecución de la siguiente forma:

*/nix/store/<hash>/bin/mpirun -np <n> -hosts <nodos> /nix/store/<hash>*

*/bin/jacobi_cuda_normal_mpi -t <X Y> -d <dimensión>*

Teniendo en cuenta que:

- <hash>: El hash de la aplicación ya derivada con el nombre y versión de la misma.
- <n>: Cantidad de nodos (en este caso las Jetson TX1).
- <nodos>: Todos los nodos involucrados en el proceso (que tienen que ser igual al valor n de -np).
- <XY>: Tamaño de la dimensión del problema.
- <dimensión>: El tamaño del problema (ancho y alto de la matriz a calcular).

Después de ejecutar este *Benchmark*, al igual que en la versión tradicional, se observa unos valores que mejoran el rendimiento al aumentar el tamaño de la dimensión, además el mejor rendimiento lo tiene la división de tablas X,Y=(1,nodos).

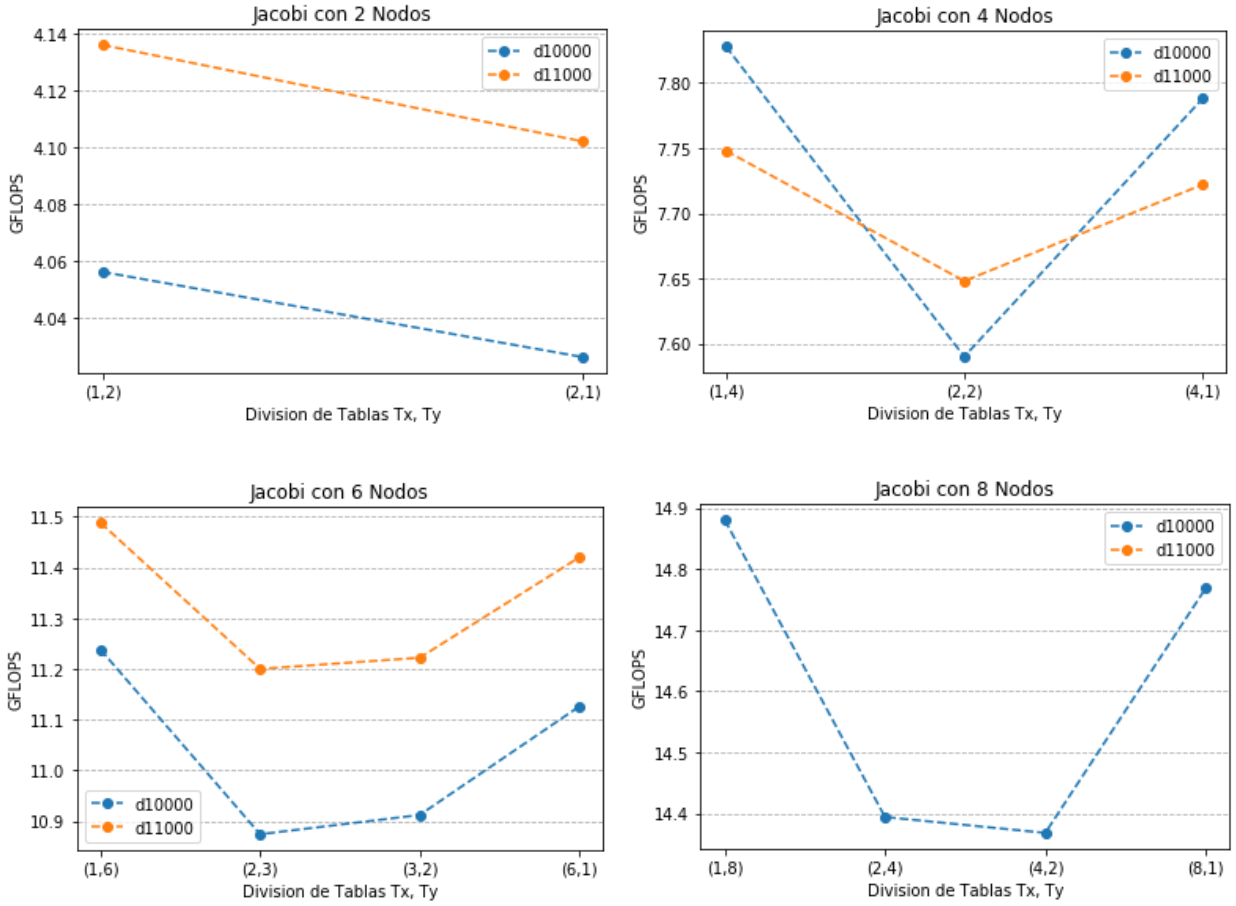
De la misma forma que sucedió en el Jacobi Benchmark tradicional, los resultados obtenidos cuando la dimensión del sistema es 4096 no son buenos. Por esta razón, se piensa que al mostrar los resultados es mejor mostrarlos con valores superiores a éstos para no mostrar un sub-uso de recursos por la dimensión predeterminada.

Por lo anterior se crean entonces gráficas con los mejores valores de rendimiento del sistema. Estos valores son con dimensiones 10.000 y 11.000. Los resultados con estos valores se muestran en la

Figura ***20***, donde se puede observar las condiciones que se tienen mejor resultado (Tx,Ty=1,nodos) y las diferencias entre mayor dimensión en el *benchmark*.

**Figura 20**

*Resultados de Jacobi Benchmark Usando Nix Según las Dimensiones de Mejor Rendimiento de la Prueba.*




Para especificar el valor de la dimensión se debe hacer un ensayo y error dentro de la arquitectura que se tenga. En nuestro caso se usó el ensayo sobre un nodo y luego se replicó a varios nodos. Debido a procesos externos en el sistema operativo, ya sea cualquier programa abierto o librerías de soporte de algún elemento de la Jetson, el Nodo puede llenar su RAM antes y terminar el proceso del *Benchmark* con error. En el caso específico del proyecto, se alcanzaba a llegar a valores de dimensión=12800 pero al hacer las pruebas en todos los Nodos algunos se caían. Por esta razón se decide utilizar solo la cantidad de dimensiones que abarque de mejor manera a

todos los Nodos sin que hayan estos problemas. De esta forma se estableció que el límite sería de 11.000.

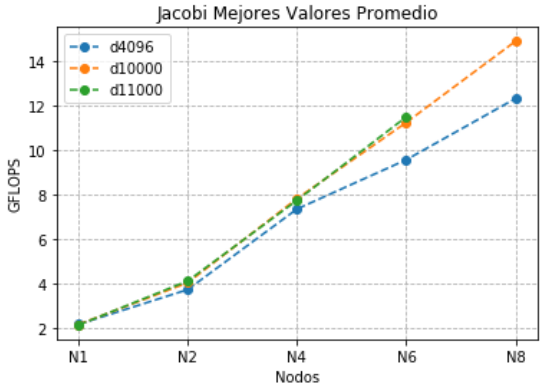
A pesar del cuidado que se tuvo para la realización de estas pruebas, en el momento de realizar el *benchmark* para 8 nodos, éste falló por llenarse la memoria RAM. La respuesta más válida a la razón de este problema se sitúa en uno de los Nodos que contiene la memoria externa donde se guardó y compiló el *Nix Store*. Además, otros Nodos aún tenían servicios que consumían demasiado recursos y además se reiniciaban cada vez que se cerraban manualmente, por lo que el límite usando en 8 Nodos fue de 10.000 únicamente.

Teniendo en cuenta los mejores promedios, se crea una gráfica que ilustra los mejores promedios por Nodos y por División de Tablas (X,Y) del rendimiento del sistema en GFLOPS, los resultados se observan en la

Figura ***21***:

**Figura 21**

*Jacobi Benchmark: Mejores Resultados del Promedio de GFLOPS por Número de Nodos y Dimensión*




A pesar de no tener los valores con dimensión 11.000, se puede observar una tendencia a ser un poco mayor a su versión de 10.000 a medida que crecen los nodos. No obstante, esta

tendencia igual da una separación mínima que a modo de análisis se puede descartar (en este caso permitir el desconocimiento exacto del resultado de esta prueba).

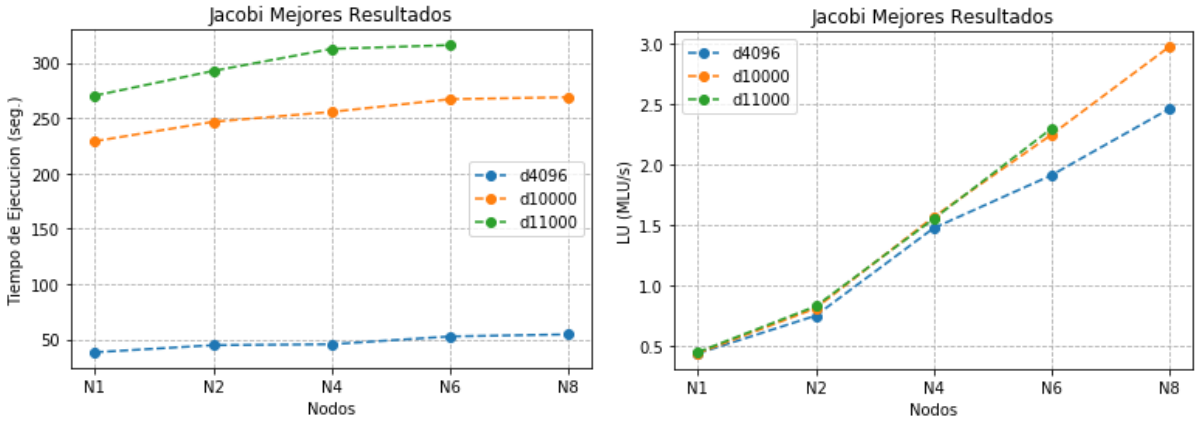
También es interesante observar que a medida que se aumentan la dimensión del problema por Nodos, los resultados de cada gráfica tiende a volverse plana, es decir, los resultados dejan de tener impacto por medio de la división de tablas Tx,Ty cuando la Dimensión crece.

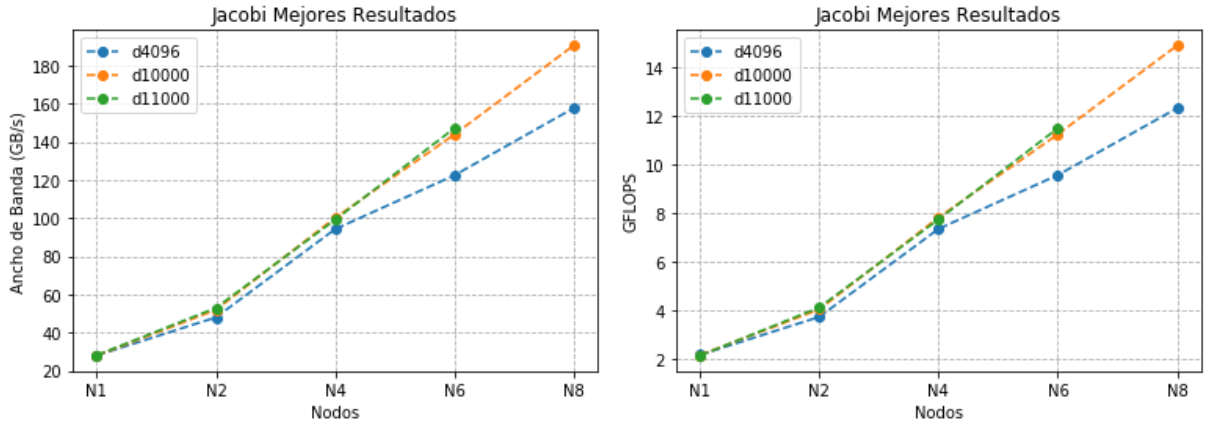
Los otros resultados (*Bandwidth*, *Lattice Updates* y *Tiempo de Ejecución*) tienen el mismo comportamiento entre ellos por Nodos, algo esperado considerando los resultados en su versión tradicional. La única diferencia radica en los resultados con 4 Nodos que tienen una diferencia de lo esperado en los resultados exceptuando el Tiempo de Ejecución, y esto puede deberse principalmente a problemas de comunicación, un programa corriendo en *background*, falta de memoria RAM o un problema interno del Nodo (por ejemplo, el control de temperatura).

Haciendo ya una comparación con los mejores resultados de cada Nodo se obtienen resultados interesantes, con tendencias en crecimiento y llegando a un límite según el número de Nodos:

**Figura 22**

*Jacobi Benchmark Mejores Resultados por Nodos*






*Nota.* Los mejores resultados suceden cuando cuando Tx,Ty= (1,nodos).

Como se puede observar en la

Figura ***22***, entre las dimensiones 10.000 y 11.000 no existe una mejora significativa de los resultados, salvo que por tendencia se hable de muchos nodos, en general se puede observar una tendencia de crecimiento, pero no hay que olvidar que el sistema está sujeto a un límite en RAM (que viene siendo la dimensión 11.000 aproximadamente) y la conexión entre los sistemas se hicieron con un *switch* tradicional y cables ethernet normales, así que la infraestructura puede mejorar para conseguir mejores resultados aún teniendo el máximo límite de cada Nodo a través de la comunicación entre nodos.

### 6.4 Jacobi Benchmark vs Jacobi Benchmark Nix

Se ha mostrado los resultados de los *Benchmarks* de la infraestructura de forma tradicional y usando el Manejador de Paquetes, graficado dichos resultados y concluyendo de ellos. Ahora en esta parte se comparará los resultados de dichos experimentos.

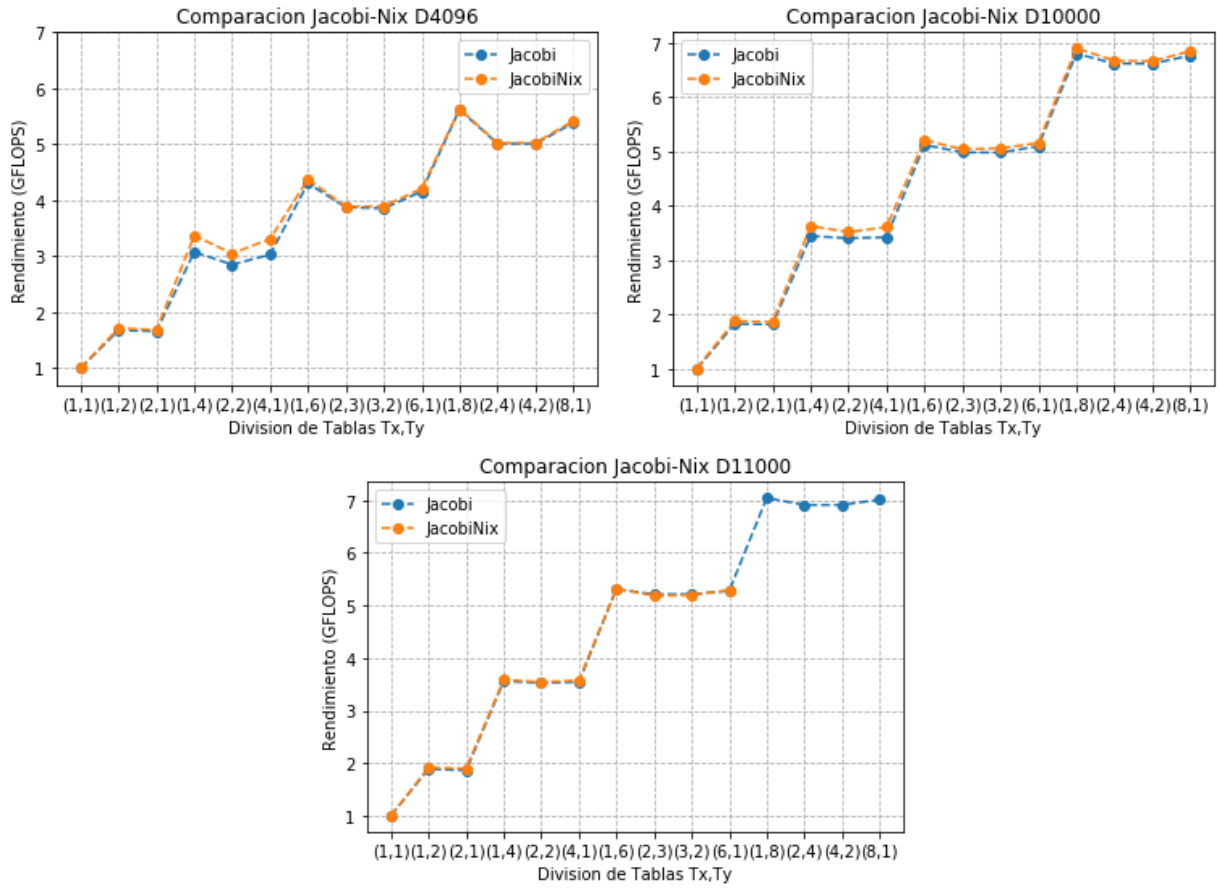
Como existe una similitud en el comportamiento de los resultados de *GFLOPS, Bandwidth* y *Lattice Updates (LU)*, se mostrará la comparación entre los valores obtenidos de la Velocidad de Cálculo y Tiempo de Ejecución.

Para tal análisis, se normaliza los datos según cada dimensión y cada tipo de *benchmark* (el tradicional y con Nix), midiendo cuánto mejora con respeto al valor inicial (1 nodo). Los resultados encontrados fueron interesantes, como se observa en la

Figura **23**:

**Figura 23**

*Comparación de Jacobi y Jacobi Nix Normalizados y Graficados por Cada Nodo.*




*Nota.* Puede observarse un crecimiento muy similar en ambos sistemas lo que sugiere que no hay una fuerte influencia en el uso del Manejador de Paquetes.

En la

Figura **23** se muestra la comparación del crecimiento del rendimiento en la prueba del Jacobi Benchmark usando la manera tradicional y con Nix. Éstos datos están normalizados para

observar el crecimiento en bruto usando los 2 tipos de *benchmarks*. Se puede observar que su comportamiento es muy similar en ambos casos, ya que crecen de la misma manera dependiendo de las tablas Tx,Ty, por lo que se puede concluir que con respecto al crecimiento normalizado, el usar Nix no sugiere una pérdida significativa en el rendimiento del sistema.

Lo anterior indica un comportamiento similar entre los dos métodos. Haciendo la misma revisión usando los valores reales se encuentran que los mismos tienen un comportamiento similar en cuanto a la homogeneidad en los resultados finales. Así, a partir de 6 nodos el comportamiento y resultados de cada método se aproximan a ser iguales sin cambios significativos, por lo que el experimento sugiere que el uso de Nix no cambia en nada el comportamiento general del sistema, es decir, no representa un impacto negativo en su rendimiento.

En cuanto a los valores mostrados en el tiempo de ejecución mostrados en la

De las gráficas expuestas en la Figura 24 se puede observar un cambio inicial entre los 2 métodos que van disminuyendo a medida que los nodos aumentan. Este comportamiento se debe principalmente a que el impacto de ejecutar el benchmark usando Nix disminuye con mayor nodos debido a que el costo de este proceso se reparte entre los nodos, a tal punto que con 6 y 8 nodos no se aprecia una diferencia que sea significativa.

Se esperaba una disminución del tiempo de ejecución por parte del método Nix debido por los accesos a memoria y lectura de binarios. Esta espera fue corroborada con el registro de los valores reales del benchmark que indicó una mejora sobre el método tradicional que se va perdiendo a medida que aumentan los nodos.

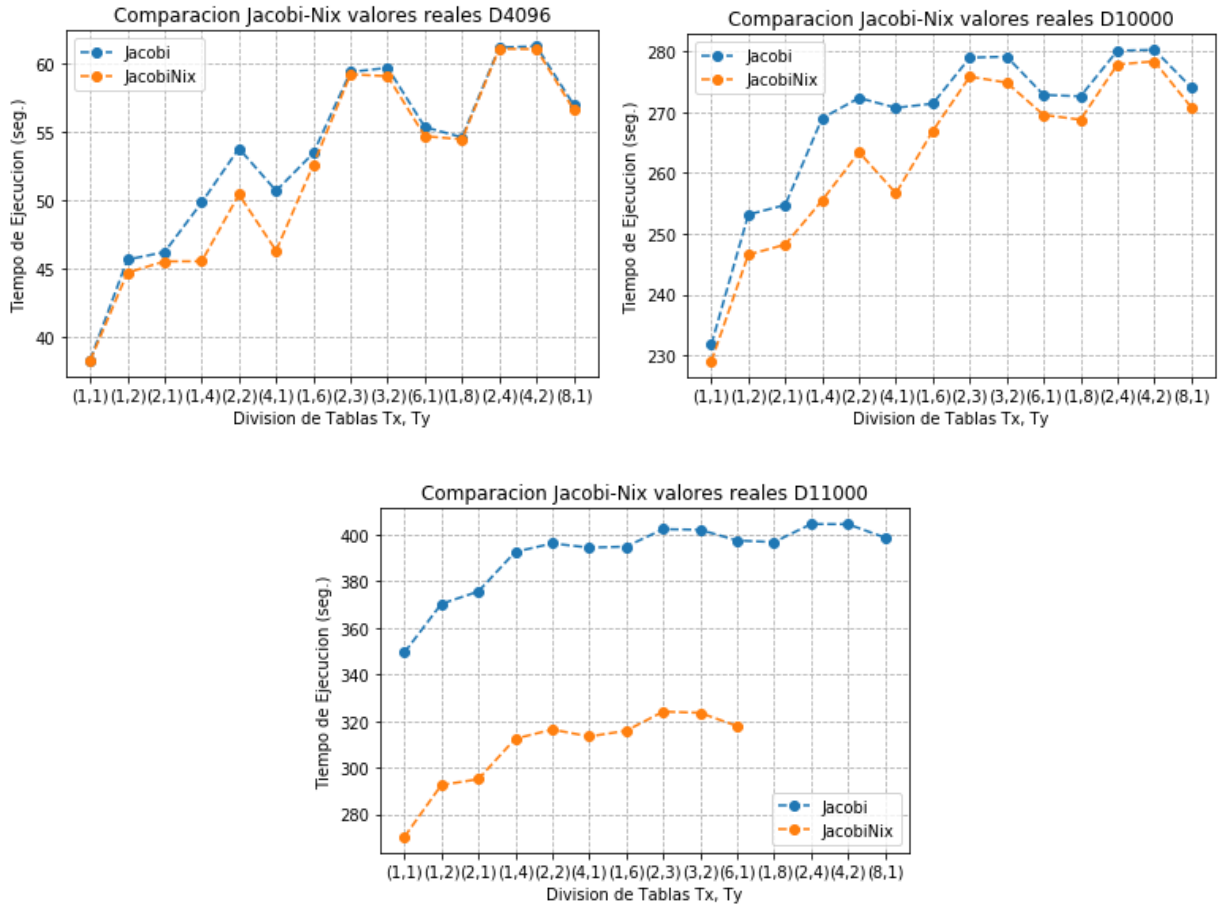
**Figura 24**, se puede observar que, aunque no están tan cercanos las gráficas de los 2 métodos, sí poseen un comportamiento muy similar. Además de lo anterior, entre más nodos se ejecute el *benchmark*, los valores de las gráficas estarán más cercanos, viéndose una similitud muy grande desde los 6 nodos.

De las gráficas expuestas en la Figura 24 se puede observar un cambio inicial entre los 2 métodos que van disminuyendo a medida que los nodos aumentan. Este comportamiento se debe principalmente a que el impacto de ejecutar el benchmark usando Nix disminuye con mayor nodos debido a que el costo de este proceso se reparte entre los nodos, a tal punto que con 6 y 8 nodos no se aprecia una diferencia que sea significativa.

Se esperaba una disminución del tiempo de ejecución por parte del método Nix debido por los accesos a memoria y lectura de binarios. Esta espera fue corroborada con el registro de los valores reales del benchmark que indicó una mejora sobre el método tradicional que se va perdiendo a medida que aumentan los nodos.

**Figura 24**

*Comparación Jacobi y Jacobi Nix Según las Dimensiones, Normalizados al Tiempo de Ejecución de la Versión Tradicional a 1 Nodo*




*Nota.* Se aprecia una similitud en el comportamiento de las gráficas, pero entre más grande sea la dimensión, se aprecia mucho mejor la mejora con el uso de Nix con Jacobi.

La principal razón es el uso de memoria y la red de la infraestructura: con el método Nix, los binarios y librerías estaban disponibles de inmediato a través de los paquetes por lo que su ejecución era más rápida. No obstante, a medida que los nodos aumentan, se presenta problemas de comunicación debido al uso de la red por parte de los nodos, aumentando la latencia y perdiendo la ventaja que tenía éste método.

### 6.5 Consumo Energético de los NVIDIA Jetson TX1

Para analizar el consumo energético del sistema, se empleó las formulas dadas por (Woo & Lee, 2008). En este texto, el autor extiende las formulas de la ley de Amdahl para nuevos casos

de diseño de arquitecturas. Éstas arquitecturas son el uso de múltiples procesadores (denominado P*), múltiples procesadores eficientes (llamado c*) y 1 procesador tradicional y muchos eficientes (llamado P+c*).

De cada nuevo tipo de diseño de sistema mencionado arriba el autor analizó su comportamiento y basado en la ley de Amdahl extendió sus fórmulas, teniendo en cuenta varias suposiciones. Al final, para nuestro caso (P + c*) se obtienen estas fórmulas:

$$Rendimiento = \frac{1}{(1+f) + \frac{f}{(n-1)S_c}} \tag{1}$$

$$W = \left[(1-f)(1+(n-1)W_c K_c) + \frac{f}{S_c}\left(\frac{k}{n-1} + W_c\right)\right][Rendimiento] \tag{2}$$

$Donde,$

$n: Número\ de\ Nodos$

$f: Fracción\ del\ código\ paralelizable$

$k: Fracción\ consumo\ en\ idle\ del\ Procesador\ P$

$S_c: Fraccion\ de\ eficiencia\ del\ Procesador\ c\ con\ respecto\ al\ Procesador\ P$

$W_C: Fracción\ de\ consumo\ de\ energía\ del\ Procesador\ c\ con\ respecto\ al\ Procesador\ P$

$K_c: Fracción\ consumo\ idle\ del\ Procesador\ c\ con\ respecto\ a\ sí\ mismo.$

Para obtener los valores pedidos en estas funciones, se apoyó de varias fuentes donde habían hechos pruebas individuales del sistema. En (Everything you need to know About the NVIDIA Jetson TX1 Performance, s.f.) muestra un benchmark de la Jetson TX1 donde entregan valores del uso del CPU, uso de CUDA y consumo en Idle. En (ARM A53/A57/T760 investigated - Samsung Galaxy Note 4 Exynos Review, s.f.) muestra otras pruebas con los 4 *cores* activos de los ARM A57 que sirven para la ecuación pero ubicados en otro tipo de sistema (celular) y en (Roy

Longbottom's Android Benchmarks For 32 Bit and 64 Bit CPUs from ARM, Intel and MIPS, s.f.) menciona el rendimiento promedio del ARM A57 dentro de los sistemas Jetson TX1.

Utilizando los valores antes mencionados, se hace el despeje de las ecuaciones y se obtiene que Performance=9,4696 y W=1,5172, dado en valores normales, lo que da un valor de 6,49W de consumo promedio en toda la Jetson TX1.

### 6.6 Consumo Energético de la Jetson TX2 Usando Tegrastats

A manera de comparación y teniendo en cuenta el objetivo de analizar los sistemas embebidos para HPC, se logró obtener resultados de consumo energético en una Jetson TX2.

En el caso de esta propuesta, el laboratorio pudo obtener una Jetson TX2 para uso en pruebas HPC y otras áreas. Desafortunadamente solo se obtuvo una, por lo tanto no es posible realizar una comparación más detallada (en clúster) de estos sistemas, sin embargo se puede ofrecer una visión general del consumo promedio.

Se instala la última versión del Jetpack para la Jetson TX2 (ahora llamado SDK Manager) en una computadora con Ubuntu 14.04. Esta herramienta instala por defecto las últimas versiones del Linux for Tegra (sistema operativo para la Jetson), CUDA y otros paquetes para AI y Visión por Computador. De esta forma se obtiene una herramienta de monitoreo del sistema denominado Tegrastats.

Tegrastats es una herramienta que monitorea uso de recursos del sistema en general, así como uso del CPU, GPU, Temperatura y, al interés del proyecto, la Energía. Una vez que se ejecuta, dentro de la terminal le va entregando en tiempo real las estadísticas del sistema. Éstas estadísticas pueden copiarse a un archivo e incluso especificar el intervalo en el que analiza el uso del sistema (que por defecto es 1 segundo).

Usando como referencia una propuesta de monitoreo del consumo de Energía y Rendimiento del sistema llamado enerGyPU (Garcia H., Hernandez B., Montenegro, Navaux, & Barrios H., 2016), se realizó un script para obtener estos datos y luego graficarlos. Como el método que los autores usan no incluye para Jetson TX2 (salvo una rama que usa la operación Tegrastats pero lo envía a otra aplicación como nvprof para su uso) se decidió crear uno propio que lo pudiese usar y tenerlos afuera para graficarlos.

Afortunadamente, la última versión del Tegrastats (a la fecha la 32.1 del L4T) posee en sus estadísticas 6 diferentes tipos de monitoreo de energía para la TX2. De esta forma simplemente usamos un script que monitoree el sistema un poco antes de iniciar el *Benchmark* y acabe un poco después de que el *Benchmark* haya terminado.

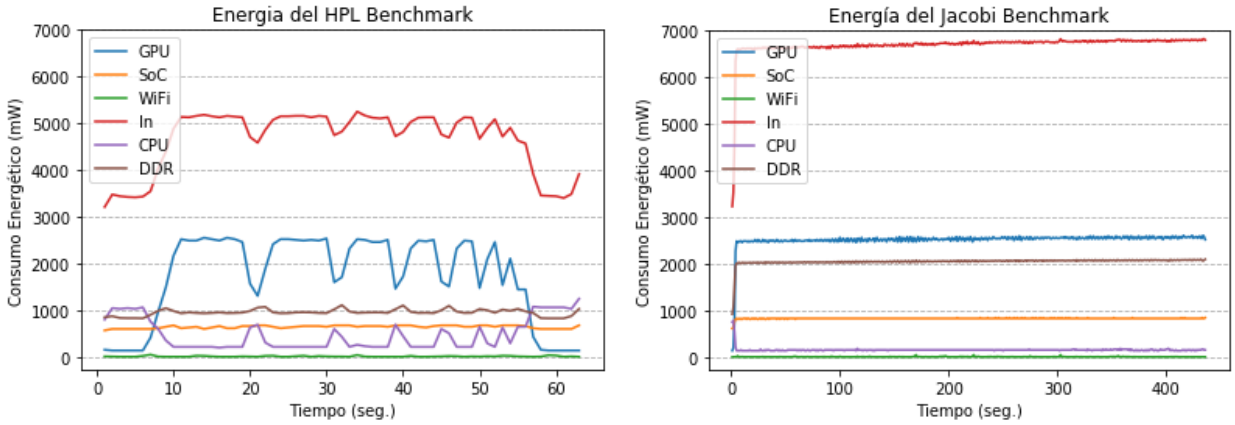
De esta forma obtenemos una gráfica como la mostrada en la En la **¡Error! La autoreferencia al marcador no es válida.** existen 2 gráficas, cada una de un *Benchmark* diferente. En la de HPL se aprecia unas caídas de energía cada cierto tiempo. Éstas caídas se deben a que la aplicación pasa del uso principal del GPU al uso del CPU, lo que hace que caigan los valores del GPU. Con esta gráfica se puede apreciar que no escala bien ya que ese intercambio entre CPU a GPU es repetitivo a lo largo de la aplicación.

***Figura 25***, donde se puede apreciar el comportamiento del *Benchmark* en cuanto a consumo energético. Cada línea representa un elemento que la herramienta Tegrastats puede dar información acerca del consumo energético: GPU, SoC, WiFi, Sys-In, CPU, Sys-DDR. Los primeros segundos abarcan el momento del llamado y la carga de la aplicación, por lo que se aprecia ese crecimiento en la energía.

En la **¡Error! La autoreferencia al marcador no es válida.** existen 2 gráficas, cada una de un *Benchmark* diferente. En la de HPL se aprecia unas caídas de energía cada cierto tiempo. Éstas caídas se deben a que la aplicación pasa del uso principal del GPU al uso del CPU, lo que hace que caigan los valores del GPU. Con esta gráfica se puede apreciar que no escala bien ya que ese intercambio entre CPU a GPU es repetitivo a lo largo de la aplicación.

**Figura 25**

*Energía Promedio de la Jetson TX2 Mientras se Ejecutaba el HPL Benchmark (izquierda) y Jacobi Benchmark (derecha)*




Sin embargo, en la gráfica del *Jacobi Benchmark*, se observa que es muy homogéneo el uso de los recursos, por lo que su escalabilidad en cuanto a energía es igualmente homogénea y lineal. La gráfica sugiere que este algoritmo posee una escalabilidad directa con la cantidad de memoria, ya que el paso a CPU (aunque gaste más) es de la misma forma a lo largo de la aplicación.

En resumen, el consumo energético de la Jetson TX2 depende exclusivamente del tipo de aplicación de que tenga y va cambiando dinámicamente en el momento que la aplicación lo requiera. En el momento de ejecutar estos *scripts*, la Jetson TX2 estaba en el estado por defecto cuando se compila el sistema. Esto significa que posee todos los programas por defecto en

*background* ejecutándose, además de la interfaz de usuario gráfica (GUI), lo que hace que el consumo de energía típico sea un poco más de lo que sería si se pusiera en un clúster de Jetson TX2 ya que en este caso se usaría solo la terminal y los programas necesarios.

Los resultados promedio fueron en *HPL Benchmark* 8,5W y en *Jacobi Benchmark* 12W.

### 6.7 Resumen del Experimento

De una infraestructura de 8 NVIDIA Jetson TX1 se realizaron varios experimentos y pruebas usando 2 Benchmarks conocidos: HPL Benchmark y Jacobi Benchmark. Por motivos de tiempo de uso limitado a los Nodos (las Jetson TX1), se decidió probar con 1 solo Benchmark que pudiese comparase con métodos tradicionales (ejecutándolo de forma manual) y usando el Manejador de Paquetes Nix. El hacer este cambio no impactaría negativamente en el cumplimiento de los objetivos del proyecto ya que este se basa directamente en el análisis de la infraestructura, no del Benchmark en sí.

Los resultados del Jacobi Benchmark (el Benchmark escogido para el experimento), muestran un crecimiento en el rendimiento del sistema a medida de que aumenta los nodos. Además de lo anterior, se concluye que para este Benchmark los mejores resultados se tienen al especificar la tabla Tx,Tx como (1, nodos) y que el máximo valor de la dimensión del sistema es 11.000 ya que a valores más altos la RAM de los Nodos individuales se llena y genera que todo el proceso se detenga.

Usando Nix, obteniendo sus datos y graficando se observa un mismo comportamiento con respecto a su par tradicional. Revisando si aparte del comportamiento los valores que se obtienen compiten con los valores tradicionales se compararon ambos datos, dado por sorpresa que, no solo

los valores se acercan con respecto a la forma tradicional, sino que también mejora de manera muy pequeña (no significativa) a este método cuando los nodos son muy pocos (Nodos < 6).

Finalmente, el tiempo de ejecución en Nix es mejor que del método tradicional como se esperaba, ya que la ejecución es directa con el binario. Sin embargo, a medida de que los Nodos del sistema crecen, se pierde esta ventaja y es debido a que la comunicación entre nodos se satura para obtener este binario y por lo tanto la latencia aumenta.

### 6.8 Resumen del Análisis de Energía

En el caso del consumo energético, se buscó 2 maneras de realizar una aproximación de este valor: a través de cálculos matemáticos basados en un proyecto previo y benchmarks realizados por otras personas, y a través del uso de la herramienta de uso del sistema de NVIDIA, que en su nueva versión permite la obtención de datos de consumo energético de ciertas componentes individuales del sistema.

Con un valor normalizado de 1,5172 Watts en la Tegra TX1 (siendo 6,49 Watts en promedio), y de 12W y 8,5W para el Benchmark de HPL y Jacobi respectivamente para la Jetson TX2, se observa un rendimiento favorable con respecto al consumo energético en estos sistemas.

En este capítulo se mencionó las pruebas realizadas con el *Benchmark* y sus resultados usando el método tradicional y con un Manejador de Paquetes. De estas pruebas se extrajeron los datos de los resultados de ambas pruebas, se graficaron y analizaron encontrando algunos resultados no esperados como otros esperados dependiendo del cómo se había levantado la infraestructura.

En la Tabla 6 se resume los mejores resultados promedio de la aplicación de algunos benchmarks sobre el sistema levantado. En esta tabla se observa la pérdida de rendimiento por el uso de red entre los nodos y además se observa un buen rendimiento en la Jetson TX2.

La Tabla 6 se observan resultados interesantes con respecto a los mejores valores de las pruebas de Benchmarks. Podemos apreciar que con 1 solo Nodo se obtienen los mejores resultados, más existe como referencia base y no como prueba en clúster. A medida que crecen los nodos se pierden una linealidad de eficiencia, esto es debido a las múltiples comunicaciones externas al algoritmo, lo que debe ser un punto de consideración en futuros proyectos.

**Tabla 6**

*Mejores resultados promedio de los benchmarks sobre la Jetson TX1 y TX2*

| Arquitectura | Rendimiento (GFlops) | Consumo Energético (Watts/Nodo) | Eficiencia Energética (GFlops/Watts) | Eficiencia por Core (Watts/Core) |
|---|---|---|---|---|
| **TX1 HPL 1 Nodo** | 7,22 | 9, 89 | 0,73 | 0,033 |
| **TX1 HPL 8 Nodos** | 23,98 | 9, 89 | 0,303 | 0,033 |
| **TX2 HPL 1 Nodo** | 14,31 | 4,647 | 3,08 | 0,018 |
| **TX1 Jacobi 1 Nodo** | 2,16 | 9, 89 | 0,22 | 0,033 |
| **TX1 Jacobi 8 Nodos** | 15,22 | 9, 89 | 0,192 | 0,033 |
| **TX1 Jacobi-Nix 8 Nodos** | 14,76 | 9, 89 | 0,186 | 0,033 |

En la misma tabla se compara el Benchmark Jacobi con el Jacobi-Nix y se puede apreciar una ligera mejora en su eficiencia energética, más no es crítica, por lo que se puede concluir que dicho método usando Nix puede mantener y en algunos casos mejorar ligeramente la eficiencia del código gracias a su aislamiento del proceso en el sistema.

En el siguiente capítulo se recogerá toda la información obtenida a lo largo de los capítulos, para poderla analizar y dar las conclusiones obtenidas en este proyecto.

# 7. Conclusiones

El presente proyecto se adentró en el uso de sistemas embebidos para la computación de alto rendimiento (HPC), utilizando de igual manera una herramienta para la gestión de dichos programas. Después de una búsqueda, selección, diseño, levantamiento y pruebas, se ha obtenido un mejor panorama en el uso de dichos sistemas para HPC.

Iniciando con los manejadores de paquetes modernos, estas herramientas son actualmente indispensables para poder gestionar de la mejor forma las aplicaciones e usuarios de un sistema. Si bien es cierto que cada herramienta posee ventajas y características que lo hacen única en la gestión de dichos programas, es Nix, el manejador de paquetes funcional el escogido para la realización de las pruebas en el sistema.

Las propiedades más destacables de Nix son su facilidad de instalación y ejecución de los comandos dentro de su entorno, no requiere permisos de instalación de usuario elevado, el concepto de paquetes en el que se encuentran los programas son elementos aislados, por lo que se controla qué librerías se usan para cada programa. Estos paquetes se ejecutan directamente en binario, creados a partir de una "receta", además son escalables y portables por su naturaleza de aislamiento. Aunque sigue siendo una herramienta en desarrollo y todavía más para sistemas embebidos, posee una comunidad relativamente grande donde se esmeran por mejorar su desarrollo, comunidad en la que se incluye el laboratorio hermano en Grenoble, Francia donde se realizó las pruebas de los sistemas embebidos.

En la parte de la infraestructura, se exploraron algunos sistemas embebidos y se concluyó que dichos sistemas para ser usados en HPC, requerían como mínimo tener soporte para uso como nodo esclavo, poseer cores de procesamiento en paralelo, principalmente los ARM (o alguno de la

línea de procesadores con bajo consumo energético), tener GPU Cores que soporten las librerías de CUDA (por el momento) y que soporte el uso de manejadores de paquetes modernos.

De esta manera, se levantó una infraestructura de 8 NVIDIA Jetson TX1 conectados por cable ethernet tradicional a un switch, y usando un computador portátil como nodo maestro (sistema operativo Debian) y almacenamiento, se integró el sistema con el manejador de paquetes funcional Nix.

Teniendo esta estructura de sistema creado, se dio cuenta que a pesar de que la configuración de los Nodos con Nix es compleja por su desconocimiento en la tecnología y que el aprendizaje de ella es relativamente alta, una vez que se obtenga la receta para su derivación es prácticamente invisible la ejecución de este paquete para futuros usuarios.

Utilizando este sistema se realizaron pruebas de rendimiento y eficiencia energética teniendo en cuenta 2 benchmarks: HPL y Jacobi. Se probó HPL con todos los nodos y Jacobi con y sin Nix. Analizando los resultados se aprecia una pequeña (pero no concluyente) mejora usando Nix frente a sus valores tradicionales. No obstante, esta mejora nos permite concluir que el método de uso con Nix no interfiere con el rendimiento del sistema. Sin embargo, la herramienta sí interfiere con el uso en memoria, evidenciado en la imposibilidad de hacer condiciones de pruebas en varios nodos iguales a las que tenían con 1 nodo.

Además de los resultados anteriormente descritos, se calculó teóricamente el consumo energético de estos sistemas basados en un trabajo previo sobre consumo energético de nuevos tipos de procesadores. A pesar de ser un cálculo teórico con muchas suposiciones, los valores se acercan a lo mencionado por los fabricantes. No obstante, se pudo obtener y probar 1 Nodo de la nueva generación de los nodos usados en este proyecto, y usando una herramienta nueva para el control de energía en estos nodos, se pudo tener una nueva comparación en el consumo energético

de estas tarjetas, concluyendo que va por buen camino en las mejoras de desarrollo de estos sistemas.

El mejor escenario para estos sistemas (y en general para cualquier sistema basado en ejecución en GPU Cores), es tener el algoritmo que no use casi nada la comunicación entre nodos y que dentro de este tenga en gran parte de su código un paralelismo en ejecución.

Un buen uso de este modelo de arquitectura de sistemas embebidos con un manejador de paquetes es en el área de cálculo científico. Aquellos laboratorios que requieran algún tipo de procesamiento o cálculo de alto rendimiento pero que no sea viable gastar en un alquiler de recursos de algún supercomputador, puede ser una opción viable el invertir en estos sistemas, además, por su portabilidad y escalabilidad, este modelo permite crecer a medida que se tengan los recursos para hacerlo y gracias a su manejador de paquetes no es complejo ejecutar dichos programas en entornos creados específicamente para ellos.

## 8. Trabajo Futuro

En este proyecto se hizo una revisión de forma general del rendimiento de una arquitectura poco convencional denominada sistemas embebidos basados en benchmarks. Cada uno de estos benchmarks aportaron alguna información sobre el comportamiento del sistema y el proceso general de poder hacer estas pruebas de igual manera aporta una información valiosa a la hora de concluir sobre el mismo sistema.

No obstante, cabe resaltar que existen muchos otros caminos que se pueden bifurcar o derivar de esta propuesta. Por ejemplo, la forma de obtener los resultados de los experimentos. En este esquema se usó dos benchmarks conocidos para la obtención de resultados. Sin embargo, podría buscarse otros benchmarks que abarquen otras áreas de las arquitecturas probadas, como

solo uso de CPU, pruebas de estrés de la GPU, pruebas de comunicación entre otros. También aplicaciones como inteligencia artificial, sabiendo que ahora NVIDIA se está enfocando en esta área para las Jetson, muy seguramente ofrecerán resultados interesantes para la Jetson TX1 y TX2.

Desde el punto de vista de la arquitectura, futuros proyectos se evaluarían la nueva Jetson TX2 (la más actual a la fecha), además de usarlo junto con la TX1 como una arquitectura distribuida heterogénea entre nodos (ya es heterogénea en sí misma), para observar su comportamiento de escalabilidad y rendimiento en el sistema.

Finalmente, otra línea de evaluación sería frente a la energía sin necesidad de usar programas alternos, usando herramientas de medición eléctricas para poder observar el comportamiento con menos ruido y más precisión. Desde luego todo lo expuesto en este apartado se puede agregar la mejora en la misma red, como conexiones gigabit y almacenamiento SSD, aumentando el número de nodos hasta llegar a su máxima escalabilidad viable de rendimiento para observar el mejor comportamiento de la arquitectura expuesta.

## Bibliografía

(UC-Lab), U. C. (s.f.). *Hadoop cluster running on Intel Galileo Gen 2*. Recuperado el Abril de 2019, de https://uc-lab.in.htwg-konstanz.de/blogging/hadoop-cluster-running-on-intel-galileo-gen-2.html

Ai, Y., Peng, M., & Zhang, K. (2017). *Edge Computing Technologies for Internet of Things: A Primer.* Elsevier.

*Amazon Web Services*. (2017). Obtenido de https://aws.amazon.com/es/

*ARM A53/A57/T760 investigated - Samsung Galaxy Note 4 Exynos Review*. (s.f.). Recuperado el Abril de 2019, de https://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-review/6

Bashari Rad, B., John Bhatti, H., & Ahmadi, M. (2017). *An Introduction to Docker and Analysis of its Performance.* IJCSNS International Journal of Computer Science and Network Security.

Bonomi, F., Milito, R., Zhu, J., & Addepalli, S. (2012). *Fog Computing and Its Role in the Internet of Things.* ACM.

Breitbart, J., Pickartz, S., Weidendorfer, J., & Monti, A. (2017). *Viability of Virtual Machines in HPC.* Springer.

Bzeznik, B., Henriot, O., Reis, V., Richard, O., & Tavard, L. (2017). *Nix as HPC Package Management System.* ACM.

*Devtalk nvidia (foro nvidia)*. (s.f.). Recuperado el Abril de 2019, de https://devtalk.nvidia.com/default/topic/982848/jetson-tx1/tx1-specific-arm64-deb-repo-for-cuda-8/post/5063053/#5063053

*Docker Get Started*. (s.f.). Recuperado el Febrero de 2019, de https://docs.docker.com/get-started/

*Docker Web Page*. (s.f.). Obtenido de https://www.docker.com/what-container#/virtual_machines

Dolstra, E. (2006). *The Purely Functional Software Deployment Model.* Utrecht.

Dongarra, J., Luszczek, P., & Petitet, A. (2003). *The LINPACK Benchmark: past, present and future.* Concurrency and Computation: Practice and Experience.

*Easy Build Web Page*. (s.f.). Recuperado el Febrero de 2019, de https://easybuild.readthedocs.io/en/latest/

*Everything you need to know About the NVIDIA Jetson TX1 Performance*. (s.f.). Recuperado el Abril de 2019, de https://www.phoronix.com/scan.php?page=article&item=nvidia-jtx1-perf&num=3

Gamblin, T., LeGendre, M., R. Collette, M., L. Lee, G., Moody, A., R. de Supinski, B., & Futral, S. (2015). *The Spack Package Manager: Bringing Order to HPC Software Chaos.* ACM.

Garcia H., J., Hernandez B., E., Montenegro, C., Navaux, P., & Barrios H., C. (2016). enerGyPU and enerGyPhi Monitor for Power Consumption and Performance Evaluation on Nvidia Tesla GPU and Intel Xeon Phi. *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. doi:10.1109/CCGrid.2016.100

Geimer, M., Hoste, K., & McLay, R. (2014). *Modern Scientific Software Management Using EasyBuild and Lmod.* New Orleans: First International Workshop on HPC User Support Tools.

Geveler, M., Ribbrock, D., Donner, D., Ruelmann, H., Hoppke, C., Schneider, D., . . . Turek, S. (2017). *The ICARUS White Paper: A Scalable Energy-Efficient, Solar-Powered HPC Center Based on Low Power GPUs.* Springer.

Guerreiro, J., Illic, A., Roma, N., & Tomás, P. (2017). *Performance and Power-Aware Classification for Frequency Scaling of GPGPU Applications.* Springer.

*Intel Galileo (Arduino web page)*. (s.f.). Recuperado el Marzo de 2019, de https://www.arduino.cc/en/ArduinoCertified/IntelGalileo

*Intel Galileo Gen 2 Board (Specs)*. (s.f.). Recuperado el Marzo de 2019, de https://ark.intel.com/content/www/us/en/ark/products/83137/intel-galileo-gen-2-board.html

*Jetson TK1*. (s.f.). Recuperado el Marzo de 2019, de https://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html

Lucas, P., Ballay, J., & McManus, M. (2012). *Trillions: Thriving in the Emerging Information Ecology*.

McCalpin, J. (1995). *Memory Bandwidth and Machine Balance in Current High.* IEEE computer society technical committee on computer architecture (TCCA) newsletter.

Miosga, M. (s.f.). *Ubiquitous Computing Laboratory Blogs*. Recuperado el Abril de 2019, de https://uc-lab.in.htwg-konstanz.de/blogging/hadoop-cluster-running-on-intel-galileo-gen-2.html

*Montblanc-Project*. (s.f.). Obtenido de http://www.montblanc-project.eu/

*MPI Tutorial*. (s.f.). Recuperado el Marzo de 2019, de http://mpitutorial.com/

*MPI Tutorial: Running an MPI Cluster within a LAN*. (s.f.). Recuperado el Abril de 2019, de http://mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan/

*Nix Package Manager*. (s.f.). Obtenido de https://nixos.org/nix/about.html

*Nix Package Manager Guide*. (s.f.). Recuperado el Abril de 2019, de https://nixos.org/nix/manual/

*NixOs*. (s.f.). Obtenido de https://nixos.org/

*Nvidia Jetson Systems*. (s.f.). Recuperado el Marzo de 2019, de https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/

*Package Manager*. (s.f.). Recuperado el 2017, de https://en.wikipedia.org/wiki/Package_manager

Plauth, M., & Polze, A. (2017). *Are Low-Power SoCs Feasible for Heterogeneous HPC Workloads?* Springer.

Rajovic, N., Rico, A., Puzovic, N., & Adeniyi-Jones, C. (2013). *Tibidabo: Making the case for an ARM-based HPC system.* Elsevier.

Reza, A., Tyler, F., & Sherief, R. (2017). *Understanding the Role of GPGPU-accelerated SoC-based ARM Clusters.* Honolulu: IEEE International Conference on Cluster Computing (CLUSTER). doi:10.1109/CLUSTER.2017.86

*RISC vs CISC*. (2000). Recuperado el 2017, de https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/index.html

*Roy Longbottom's Android Benchmarks For 32 Bit and 64 Bit CPUs from ARM, Intel and MIPS*. (s.f.). Recuperado el Abril de 2019, de http://www.roylongbottom.org.uk/android%20benchmarks.htm#anchor5

Saffran, J., Garcia, G., Souza, M., Penna, P., Castro, M., Góes, L., & Freitas, H. (2017). *A Low-Cost Energy-Efficient Raspberry Pi Cluster for Data Mining Algortihms.* Springer.

Saurabh, N., Kimovski, D., Ostermann, S., & Prodan, R. (2017). *VM Image Repository and Distribution Models for Federated Clouds: State of the Art, Possible Directions and Open Issues.* Springer.

Spack. (2018). *Spack: A flexible package manager that supports multiple versions, configurations, platforms, and compilers.* Recuperado el Febrero de 2019, de https://spack.io/

*STREAM benchmark (web page)*. (s.f.). Recuperado el Abril de 2019, de http://www.cs.virginia.edu/stream/ref.html

Tamuli, M., Debnath, S., Majumdar, S., & Ray, A. (2015). *A Review on Jacobi Iterative Solver and its Hardware based performance analysis.* Arunachal Pradesh: IEEE conference ICPDEN 2015. doi:10.13140/RG.2.1.4323.3444

Team, S. (s.f.). *Spack GitHub*. Recuperado el Mayo de 2019, de https://github.com/spack/spack

*The Parallella Board*. (s.f.). Recuperado el Marzo de 2019, de https://www.parallella.org/

Tso, F. P., White, D., Jouet, S., Singer, J., & Pezaros, D. (s.f.). *The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infraestructures.*

*What is edge computing and how it's changing the network*. (s.f.). Obtenido de https://www.networkworld.com/article/3224893/internet-of-things/what-is-edge-computing-and-how-it-s-changing-the-network.html

*What is Heterogeneous Computing?* (2017). Obtenido de http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-computing/

Woo, D. H., & Lee, H.-H. (2008). *Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era.* IEEE Computer. doi:10.1109/MC.2008.494

Yi , S., Hao, Z., Qin, Z., & Li, Q. (2015). *Fog Computing: Platform and Applications.* IEEE.

Yi, S., Li, C., & Li, Q. (2015). *A Survey of Fog Computing: Concepts, Applications and Issues.*

# Apéndices

En esta sección se expondrá cómo se llegaron a ciertos resultados expuestos en algunos capítulos de este proyecto.

## A. Consumo Energético de la Jetson TX1

Para lograr obtener un valor del consumo energético de la NVIDIA Jetson TX1 se utilizó los valores dados por un Hyuk et al. (Woo & Lee, 2008) acerca del consumo energético basado por una arquitectura en la que involucra un procesador y muchos GPU (llamados por el autor como procesadores eficientes).

$$Rendimiento = \frac{1}{(1+f) + \frac{f}{(n-1)S_c}} \tag{3}$$

$$W = \left[(1-f)(1+(n-1)W_cK_c) + \frac{f}{S_c}\left(\frac{k}{n-1} + W_c\right)\right][Rendimiento] \tag{4}$$

$Donde,$

$n: Número\ de\ Nodos$

$f: Fracción\ del\ código\ paralelizable$

$k: Fracción\ consumo\ en\ idle\ del\ Procesador\ P$

$S_c: Fraccion\ de\ eficiencia\ del\ Procesador\ c\ con\ respecto\ al\ Procesador\ P$

$W_C: Fracción\ de\ consumo\ de\ energía\ del\ Procesador\ c\ con\ respecto\ al\ Procesador\ P$

$K_c: Fracción\ consumo\ idle\ del\ Procesador\ c\ con\ respecto\ a\ sí\ mismo.$

Teniendo esta fórmula se obtiene el problema de que se requiere la energía en estado idle de los dos tipos de procesadores que tiene el sistema. Valores que claramente no se tienen ni se pueden obtener de forma específica (de lo contrario no se usaría este método), así que la única solución es usar métodos alternos para obtener estos valores y así obtener el valor total del consumo energético.

Se conoce que la energía total del procesador tradicional es la suma de su estado en espera y activo, al igual que la del procesador eficiente, por lo que se pueden obtener dos fórmulas:

$$W_I = W_{PI} + W_{CI} \quad (5)$$

$$W_A = W_{PA} + W_{CA} \quad (6)$$

$donde$:

$P: Procesador\ Tradicional$

$C: Procesador\ Eficiente$

$I: Estado\ Pasivo\ (Idle)$

$A: Estado\ Activo$

Buscando benchmarks realizados sobre esta arquitectura desde otras fuentes, se encuentran unas pruebas que arrojan resultados interesantes sobre la arquitectura:

$$OpenGL = 16.2W$$

$$CPUTest = 10\ W; 4\ cores = 6{,}52$$

$$CUDA\ Test = 13\ W$$

$$Idle = 5\ W$$

De estos valores se desprenden 4 funciones con 4 incógnitas, que se despejan y se obtienen los valores de cada consumo energético por procesador:

$$5 = W_{PI} + W_{CI}$$

$$13 = W_{PA} + W_{CA}$$

$$10 = W_{PA} + W_{CI}$$

$$W_{PA} = 6{,}52$$

Y de esta forma los resultados son:

$$W_{PA} \cong 6{,}52$$

$$W_{CI} \cong 3{,}48$$

$$W_{CA} \cong 6{,}48$$

$$W_{PI} \cong 1{,}52$$

Teniendo en cuenta que los procesadores tradicionales (P) se toman como 1 solo (según la suposición del autor), y los procesadores eficientes (C) se toman como 256 (los CUDA Cores), entonces las variables normalizadas de algunas constantes de la formula son:

$$K \cong 0{,}2331$$

$$K_C \cong 0{,}5369$$

$$W_C \cong 0{,}003882$$

Para hallar la eficiencia entre los procesadores se usa unos resultados de benchmarking a CPU y notas del GPU, que dice que el CPU resuelve en 3,54 resultados por ciclo por 64 bits. Como el sistema está a 1,8GHz entonces se obtiene 6,372 GFlops para la ARM A57. Del mismo modo, bajo la sección de notas de la arquitectura, indica que las GPU pueden llegar a 1TFlops como pico máximo teórico, dando un promedio de 4Gfops por core, por tanto, la relación entre ambos cores son:

$$S_C \cong 0{,}6277$$

Con lo anterior se tienen todas las constantes de la fórmula original, por lo despejando las variables se obtiene que:

$$Performance \cong 9{,}4696 \quad (7)$$

$$W \cong 1{,}5172 \quad (8)$$

Al ser un valor normal, se debe reemplazar por el consumo de energía al cuál se está normalizando, y este es el procesador tradicional en estado activo, por tanto:

$$W \cong 1{,}5172 * 6{,}52\ Watts \cong 9{,}8915\ Watts \quad (9)$$