INSTITUTO POLITÉCNICO NACIONAL
CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

Análisis y modelado del rendimiento de algoritmos paralelos en *clusters* de computadoras

Tesis que presenta
Ing. Luis Alberto Rivera Zamarripa

Para obtener el grado de
Maestro en Ciencias de la Computación

Directores de tesis
Dra. Nareli Cruz Cortés
Dr. Santiago Domínguez Domínguez

México D.F., julio 2012

Resumen

Hoy en día los *clusters* junto con las bibliotecas de paso de mensajes (tal como MPI) son una buena alternativa para ejecutar aplicaciones paralelas que requieren de mucho poder de cómputo. Esto es debido a su bajo costo y su cada vez mejor desempeño. Cuando una aplicación paralela es ejecutada en un *cluster* su código y los datos son distribuidos entre los procesadores para su procesamiento con el fin de obtener un buen desempeño. Sin embargo, para realizar esta distribución y obtener un buen desempeño de la aplicación es necesario elegir una buena configuración de sus recursos, tanto del *cluster* como de la aplicación. La elección de una buena configuración no es una tarea trivial, ya que requiere un amplio conocimiento en el área. Usualmente, la configuración es dada por un experto. Sin embargo, en ocasiones no es la más adecuada, ya que la decisión la toma con base en su experiencia y no siempre cuenta con los suficientes conocimientos para considerar los costos de los parámetros implicados para obtener un buen rendimiento del algoritmo en el *cluster*.

Para obtener el mejor rendimiento del algoritmo paralelo es necesario estimar el tiempo de ejecución de las posibles configuraciones considerando tanto el cómputo como la comunicación. Estimar el tiempo de ejecución sirve para crear estrategias de cómputo y comunicación que ayuden a hacer más eficiente el algoritmo a emplear y que aproveche los recursos disponibles en el *cluster*. Sin embargo, estimar el tiempo de ejecución normalmente se requiere expresar el algoritmo paralelo en un modelo matemático, lo cual es una tarea compleja y en ocasiones los resultados no son los esperados.

Esta tesis presenta el diseño e implementación de varios modelos para estimar el tiempo de cómputo y de comunicación de aplicaciones paralelas. En el modelo de cómputo se consideran el tipo y cantidad de operaciones realizadas por los nodos. En la comunicación se consideran varios patrones de transmisión de los datos, tanto de la aplicación como del *cluster* utilizado. Los modelos son implementados en una biblioteca en *octave* para varias de las funciones de MPI. De tal manera que facilita el análisis y el modelado del algoritmo paralelo porque el usuario no tiene que tener conocimientos de modelado y puede portar su código del programa paralelo en MPI al código en octave en forma simple y rápida. La biblioteca emplea los modelos propuestos para estimar el tiempo de comunicación basada en las condiciones de ejecución de la aplicación y realiza una buena estimación del tiempo de ejecución respecto al tiempo medido de ejecución de un algoritmo paralelo.

Abstract

Today, clusters together with the message passing libraries (like MPI) are a good choice to execute parallel applications, which need a high power computational, because of their low cost and ever better performance. When a parallel application is executed on a cluster its code and data are distributed among the processors for its processing with the purpose of obtaining a good performance. However, is necessary to configure correctly the cluster and application resources. To make a good configuration is necessary to have an extensive knowledge, usually the configuration is made by an expert, which in some cases, is not enough because his decision is influenced for his experience and does not always have enough knowledge to consider the cost of the involved parameters, to obtain a good performance of the algorithm in the cluster.

To obtain the best performance of the parallel algorithm is necessary to estimate the execution time of all possible configurations considering the computing and communication. The estimation of the execution time makes possible the development of computing and communication strategies that help to make more efficient the algorithm to use and can take advantage of the available resources in the cluster. However, to make this estimation we need to express the parallel algorithm in a mathematical model that is a difficult task and in some cases the results are not the expected.

This thesis presents the design and implementation of several models to estimate the execution time of the computing and communication of parallel applications. In the computing model are considered the type and amount of the made operations by the nodes. In the communication are considered several patterns of the data transmission corresponding to the application and the used cluster. The models are implemented in an octave library used for several MPI functions, which makes easy the analysis and modeling of the parallel algorithm because the user need not haveknowledge about modeling an can migrate the code from MPI to octave code in a fast and easy way. The library uses the proposed models to estimate the communication time that is based on execution conditions of the application and makes a good estimation of the execution time with regard to the execution time measured of a parallel algorithm.

Índice general

Índice de figuras

Capítulo 1

Introducción

Un *cluster* es un grupo de computadoras que trabajan en conjunto para resolver algún problema dado. El *cluster* está constituido por tres elementos básicos que son: un conjunto de computadoras personales (PCs), una red de comunicación que conecta a las computadoras por medio de un *switch* y un *software* que permite compartir el trabajo con otras computadoras via red [32]. Hoy en día construir un *cluster* con los elementos mencionados ofrece un buen rendimiento a bajo costo y facilita el desarrollo de aplicaciones paralelas. Actualmente existen *clusters* donde sus nodos son homogéneos, en los cuales las computadoras tienen las mismas características (por ejemplo, procesador memoria RAM, tarjeta de red, ancho de banda de la red, etc.) y clusters heterogéneos donde conjuntos de computadoras no comparten las mismas características.

Para desarrollar aplicaciones paralelas en un *cluster* existen numerosas herramientas de programación, generalmente siguen el modelo de un sólo programa con múltiples datos (SPMD, por sus siglas en el inglés). En este modelo, el mismo programa (código de la aplicación) corre en los diferentes procesadores, el programador debe de dividir la carga (datos) entre los procesadores (nodos) disponibles y especificar la comunicación entre ellos para compartir datos y coordinar su operación.

Al dividir la carga, el tiempo de ejecución de la aplicación paralela se puede fraccionar en cómputo y comunicación. El tiempo de cómputo es el periodo en el cual el procesador se encuentra procesando los datos (por ejemplo: ecuaciones matemáticas). El tiempo de comunicación se refiere a la transferencia de los datos entre un par o un conjunto de procesadores.

El tiempo de cómputo es función del tiempo que tarda en procesarse una operación en el procesador o núcleo, por lo tanto, cuando el número de operaciones se incrementa, mayor será el tiempo de cómputo. Si las operaciones se dividen en partes iguales entre los procesadores de un *cluster* homogéneo, los procesadores realizarán menos traba-

jo. Por consiguiente, cuando sea mayor el número de procesadores involucrados en el procesamiento, menor será el tiempo de cómputo.

Cuando el número de procesadores se incrementa, mayor será el tiempo de comunicación, debido a que influyen varios factores que llamaremos *parámetros* tales como: el tamaño del mensaje a transmitir, la distribución de los datos, el tamaño del bloque de datos a transmitir, la latencia de la red, la latencia del *switch*, el tiempo de transmisión y un *overhead* que incluye la pérdida de paquetes, la retransmisión de paquetes y la contención del canal de comunicación.

Al ejecutar una aplicación paralela en un *cluster* es necesario asignarle recursos (procesadores o núcleos) donde se distribuye el trabajo. La asignación de los recursos aún cuando sea dada por un experto con base en su experiencia no indica que sea la más adecuada, determinar que parámetros son los más adecuados es una tarea difícil, ya que es un problema de combinatoria y puede existir un gran espacio de búsqueda dependiendo de las características del *cluster* y de la aplicación. Para modelar el tiempo de ejecución de la aplicación paralela de todas las posibles combinaciones es un trabajo arduo. Existen modelos que ayudan a modelar el tiempo de cómputo, como la ley de Amdhal [2] y el tiempo de comunicación que ayudan a estimar el tiempo de ejecución de una combinación.

Los modelos existentes que estiman el tiempo de comunicación en un *cluster* son: LogP [7], LogGP [1], PLogP [19] y PLogPT [24]. El tiempo estimado por los modelos antes mencionados son cercano al tiempo medido de ejecución para ciertos casos específicos, ya que éstos se basan en los siguientes parámetros: el tamaño del problema, el número de procesadores, la latencia de la red, el tiempo mínimo entre mensajes consecutivos y un *overhead* de transmisión y recepción. Sin embargo, estos modelos no consideran la distribución de los datos, el tamaño del bloque de datos ni la latencia del *switch* que dependen de la aplicación paralela, de la biblioteca de comunicación usada tal como *Message Passing Interface* (MPI) y del elemento de conmutación de la red. Es decir, la aplicación depende de cómo el usuario hace la distribución del cómputo y de los datos al utilizar las operaciones de MPI, tales como operaciones de comunicación punto a punto y colectivas. Las operaciones colectivas básicas son *broadcast*, *scatter* y *gather*, donde MPI emplea varios algoritmos para hacer más eficiente la transmisión de los datos, además el tamaño del bloque de datos se transmite en cada operación. Sin embargo, al utilizarse varios algoritmos hace complejo estimar el tiempo de ejecución de estas operaciones colectivas debido a que sus patrones de comunicación varían en ejecución.

Para obtener los patrones de comunicación de las operaciones colectivas básicas, se experimentó con distintos tamaños de datos, así como con diferentes números de procesadores, en este caso se usó la herramienta *Wireshark* [30] para capturar los paquetes que viajan por la red y observar cual es su destino, así como qué nodos retransmiten el mensaje hasta llegar a su destino. Al analizar los patrones de comunicación de las operaciones colectivas se observó que MPI invoca diferentes algoritmos para realizar

la transmisión, con base en el número de procesadores y el tamaño de los datos a transmitir para cada operación y éstos son:

- En el *broadcast*, se obtuvieron el árbol binomial, el árbol binomial con intercambio y el *pipeline*.
- En el *scatter*, se obtuvieron el árbol secuencial, el árbol binomial y el *pipeline*.
- En el *gather*, se obtuvieron el árbol secuencial y el *pipeline*.

Por consiguiente, en esta investigación se proponen modelos que estiman el tiempo de cómputo y de comunicación. Los modelos de comunicación abarcan los diferentes tipos de comunicación (punto a punto y colectivas), así como los diferentes patrones con su respectivo algoritmo. Por lo tanto, la estimación de los tiempos de cómputo y comunicación, ayudan a estimar el tiempo total de ejecución de una aplicación paralela.

Para comparar los tiempos estimados con los tiempos reales, se tomó el problema de la multiplicación de matrices, donde se experimentó con operaciones de comunicación punto a punto, así como con colectivas. En los experimentos se varió el número de procesadores, la distribución de los datos, el tamaño del bloque de datos y el tamaño del problema. Donde los tiempos estimados fueron muy cercanos al tiempo medido de ejecución. Así mismo, el tiempo estimado de cómputo fue muy cercano respecto al tiempo medido de ejecución. Por lo que los modelos propuestos son factibles para estimar del tiempo de ejecución de una aplicación paralela.

1.1. Motivación

Cuando un problema requiere un alto poder computacional, usualmente se programa para una computadora (nodo) usando un solo procesador y no siempre se obtiene el desempeño deseado porque es imposible mejorar el rendimiento de la computadora al usar un procesador; el procesador consumiría más energía y generaría calor inaceptable, debido al material con el que es fabricado. Por lo tanto, es más eficiente usar varios procesadores o núcleos para obtener el desempeño deseado.

Si una tarea se ejecuta en determinado tiempo en un procesador o núcleo, se espera que una computadora con varios procesadores o núcleos, el tiempo de ejecución disminuya para la misma tarea. Sin embargo, no siempre se logra, a menos que se tome ventaja del procesamiento paralelo y una de las herramientas para lograrlo es usar un *cluster*.

Los *clusters* facilitan el cómputo paralelo y ofrecen un buen rendimiento a bajo costo. Un *cluster* es un conjunto de dos o más computadoras interconectadas entre sí para resolver un problema o sección del problema. Los beneficios del cómputo paralelo en

un *cluster* deben tomar en consideración el número de procesadores que se emplean, así como, el tiempo requerido para llevar a cabo el proceso de comunicación.

Cuando los usuarios utilizan un *cluster* como herramienta de trabajo para resolver un problema, es conveniente conocer el tiempo estimado de ejecución de la aplicación paralela, sin requerir la ejecución de algún programa paralelo. Estimar el tiempo de ejecución sirve para crear estrategias de cómputo y comunicación que ayuden a hacer más eficiente el algoritmo a emplear y que aproveche los recursos disponibles en el *cluster*. Además, el usuario sabrá el tiempo que tardará en recibir algún resultado.

En el presente trabajo se creó una aplicación para algoritmos paralelos que permite el estudio de los comportamientos de los tiempos de cómputo, comunicación, y *overhead*. La aplicación analiza el pseudo-algoritmo o el modelo del algoritmo, para extraer los parámetros que ayudan en la estimación del tiempo de ejecución. Los parámetros que obtiene son: el número de procesadores, la latencia de la red, la latencia del *switch*, la memoria en RAM, la velocidad de los procesadores, el tamaño del problema, la distribución de los datos y el tamaño del bloque de datos. Además, se propusieron varios modelos para estimar el tiempo de comunicación que involucra varios patrones de comunicación. Con base en los parámetros y los modelos propuestos, se estima el tiempo de ejecución de una aplicación paralela donde se obtuvieron resultados muy cercanos al tiempo medido de una aplicación.

1.2. Definición del problema

Actualmente, existen varias áreas que requieren de mucho poder de cómputo para la solución de problemas como: modelado en física y en química, cómputo científico, predicción del clima, entre otras. La mayoría de estos problemas son codificados de forma secuencial, lo que implica menor rendimiento y por tanto un tiempo de ejecución relativamente mayor que si fuese ejecutado en un *cluster*. Muchos de estos problemas pueden ser paralelizables, pero no se lleva acabo porque se tiene la incertidumbre en el tiempo de ejecución y qué recursos (por ejemplo, número de procesadores, distribución de los datos, ancho de banda, entre otros) son necesarios para un buen desempeño.

La elección de una buena configuración (recursos) no es una tarea trivial, ya que requiere un amplio conocimiento en el área. Usualmente, la configuración es dada por un experto. Sin embargo, en ocasiones no es la más adecuada, ya que la decisión la toma con base en su experiencia y no siempre cuenta con los suficientes conocimientos para considerar los costos de los parámetros implicados para obtener un buen rendimiento del algoritmo en el *cluster*.

1.3. Objetivos

A continuación se especifica tanto el objetivo general como los particulares de esta tesis.

1.3.1. General

Obtener la configuración de una aplicación paralela y de un *cluster*, tal que mejore el tiempo de ejecución de esta aplicación en el *cluster*.

1.3.2. Particulares

- Encontrar los parámetros de un *cluster* que más influyen en la ejecución de un algoritmo paralelo.
- Modelar el tiempo de ejecución de un algoritmo paralelo en un *cluster*.
- Modelar el tiempo de comunicación de las operaciones punto a punto utilizando MPI.
- Modelar las operaciones de comunicaciones colectivas de MPI (Broadcast, Scatter y Gather).

1.4. Alcances y limitaciones

En esta investigación se propusieron modelos para el tiempo de cómputo y comunicación, para estimar el tiempo ejecución de una aplicación paralela. Los modelos incluyen los parámetros más importantes, tanto en el tiempo de cómputo y comunicación. Además, se desarrolló una aplicación que hace uso de los modelos propuestos para determinar qué parámetros optimizan el tiempo de ejecución, así como los parámetros que hacen más eficiente el uso del *cluster*.

Debido a que los parámetros se obtuvieron de computadoras con las mismas características, tales como: velocidad del procesador, tamaño de la memoria RAM, sistema operativo, velocidad de transmisión de la tarjeta de red, entre otras. Para aplicar los modelos propuestos deben de cumplir las siguientes condiciones, que son: el *cluster* debe ser homogéneo, la carga de trabajo se distribuye equitativamente entre los nodos, la ejecución de los algoritmos paralelos debe ser uno a la vez en el *cluster*, no existirá pérdida de paquetes ni retransmisión de paquetes. Además, los algoritmos deben ser expresados en el lenguaje *octave* [8] usando la biblioteca desarrollada. La biblioteca

sólo cuenta con un conjunto pequeño de funciones de MPI para estimar el tiempo de ejecución de una aplicación paralela.

1.5. Contribuciones

Las principales contribuciones de esta investigación son las siguientes:

- Modelos para estimar el tiempo de comunicación de un algoritmo, para los diferentes patrones de comunicación de MPI, tales como punto a punto y colectivas.
- Un modelo para estimar el tiempo de cómputo de un algoritmo, con los factores que conlleva.
- Un análisis del rendimiento de algoritmos paralelos, con los diferentes parámetros involucrados en la ejecución del algoritmo en un *cluster*.
- Una aplicación que determina qué parámetros hacen más eficiente la ejecución de un algoritmo en un *cluster*.
- Una biblioteca de funciones de MPI para *octave* que ayuda a expresar algoritmo de MPI que se desea paralelizar en *octave*, por lo que el usuario puede implementar el algoritmo o expresar el modelo matemático del algoritmo en *octave*.

1.6. Estructura general del documento

Este documento de tesis consta de seis capítulos.

- En el capítulo uno se describe el planteamiento del problema, los motivos que nos llevaron a resolver el problema, los objetivos del proyecto de investigación, sus alcances y finalmente las contribuciones que deja el trabajo realizado en esta investigación.
- En el capítulo dos se define qué es cómputo paralelo en *clusters* y se describen los elementos que lo componen. Se comentan las arquitecturas paralelas existentes. También se explica el concepto de paso de mensajes, así como los lenguajes que ayudan a desarrollar aplicaciones paralelas.
- En el capítulo tres, se explica el estado del arte respecto a los modelos que estiman el tiempo de comunicación, de cómputo y se explica las comparaciones más representativas.

- En el capítulo cuatro se detallan los modelos propuestos para estimar el tiempo de comunicación y de cómputo. Además se explica el funcionamiento de la aplicación que se propone, que hace uso de los modelos.
- En el capítulo cinco se definen los experimentos y se muestran los resultados que se obtuvieron al implementar los modelos propuestos para obtener el tiempo de ejecución de una aplicación paralela. Además, se realiza un análisis de los mismos y se explica el significado de los valores obtenidos de los modelos propuestos.
- En el capítulo seis se detallan las conclusiones y el trabajo a futuro que se origina del análisis que se realiza a los resultados obtenidos.

Capítulo 2

Marco teórico: Conceptos de cómputo paralelo en *clusters* y su programación

En este capítulo se describen los conceptos básicos de computación paralela, así como las arquitecturas paralelas y su clasificación. Además, se muestran los elementos que componen un *cluster*, así como las herramientas de programación que facilitan el desarrollo de aplicaciones paralelas.

2.1. Introducción

Por mucho tiempo, la tendencia era resolver los problemas en una supercomputadora con un solo procesador, donde han alcanzado grandes velocidades. Sin embargo, gracias al avance tecnológico esta tendencia pronto llegará a su fin, porque existen límites tanto físicos como arquitectónicos que restringen el poder computacional que se puede lograr en un solo procesador.

Debido a los límites que existen en un solo procesador y la necesidad de resolver problemas más rápido, ha tomado mayor importancia el tema de cómputo paralelo. Para poder llevarlo a cabo es necesario un conjunto de computadoras con múltiples unidades de procesamiento, este conjunto es interconectado mediante una red de comunicación y con el *software* necesario para que las unidades de procesamiento trabajen juntas.

El principal objetivo de usar varios procesadores es crear computadoras poderosas simplemente conectando múltiples procesadores. Con varios procesadores se espera tener una velocidad de procesamiento mayor que el procesador con mayor velocidad en el mercado. Además, por su bajo costo y cada vez mejor desempeño, proporciona un

beneficio extraordinario en precio-rendimiento. Otra ventaja es, la tolerancia a fallas. Si un procesador falla, el resto de los procesadores seguirán prestando su servicio de forma continua, aunque con menor rendimiento [9].

La programación paralela es un aspecto importante en el cómputo paralelo. El primer paso en la programación paralela es el diseño del algoritmo paralelo o programa para un problema dado. El diseño empieza con la descomposición del cómputo independiente de una aplicación en varias partes, comúnmente llamadas tareas, las cuales pueden ser ejecutadas en paralelo ya se en núcleos o en procesadores. La descomposición en tareas puede ser complicada y laboriosa, debido a que existen diferentes posibilidades de descomponer el mismo algoritmo.

Las tareas de una aplicación son codificadas en un lenguaje de programación paralela y son asignadas a procesos o hilos para después ser asignadas a la unidad de procesamiento físico para su ejecución, a esto se le llama mapeo y es usualmente hecho en tiempo de ejecución o algunas veces influenciado por el programador. Las tareas de una aplicación pueden ser independientes pero también pueden depender cada una de otros resultados (Figura 2.1). Si una requiere resultados producidos por otra, la ejecución de la primera puede iniciar solo después de que la otra haya producido los resultados y entregado. Por lo tanto, los programas paralelos necesitan de sincronización y coordinación para una correcta ejecución. Los métodos de sincronización y coordinación en cómputo paralelo son fuertemente conectados con la forma en la cual la información es intercambiada entre procesos e hilos y esto depende en la organización de la memoria.

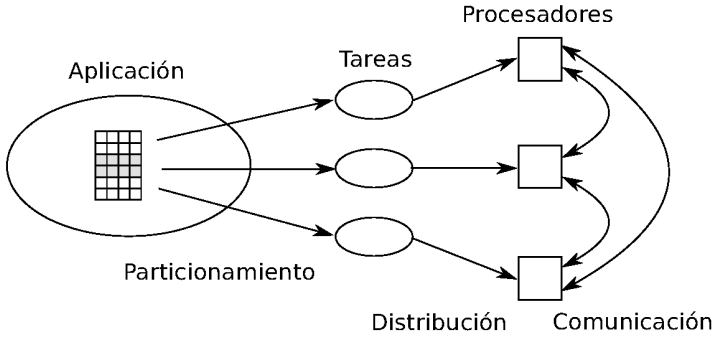

Figura 2.1: Paralelización de una aplicación

La organización de la memoria se puede clasificar en computadoras de memoria compartida y computadoras de memoria distribuida. A menudo, el término hilo puede ser enlazado con memoria compartida y el término proceso puede ser enlazado con memoria distribuida. Para las computadoras de memoria compartida, una memoria global almacena los datos de una aplicación y puede ser accedida por todos los procesadores o núcleos y sincronizados para su correcto acceso, el intercambio de información puede llevarse acabo mediante variables compartidas (Figura 2.2). Para las computadoras de memoria distribuida existe una memoria privada por cada procesador, la cual solo puede ser accedida por el mismo procesador y no se requiere alguna sincronización para

acceder a la memoria, y el intercambio de información puede llevarse acabo mediante el envío de datos de un procesador a otro por medio de una red, donde se debe de especificar la operación de comunicación explícitamente [29] (Figura 2.3).

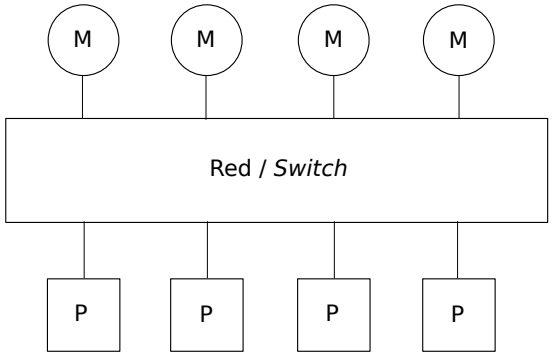

Figura 2.2: Memoria compartida

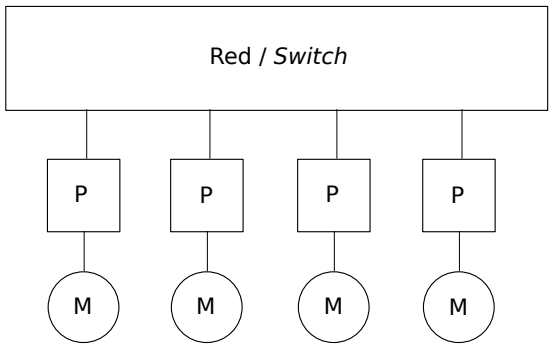

Figura 2.3: Memoria distribuida

2.2. Cómputo paralelo en clúster

El cómputo paralelo es la ejecución simultánea de alguna tarea en varios procesadores con el fin de reducir el tiempo de ejecución de una aplicación [11]. Para disminuir el tiempo de ejecución de la aplicación, el código y/o los datos de la aplicación son divididos y distribuidos entre los procesadores. Cada procesador ejecuta una o más tareas que realizan una parte del cómputo de la aplicación. Los procesadores son interconectados por un bus o una red de comunicación que permite a las tareas compartir datos o sus resultados.

El cómputo paralelo ha sido ampliamente utilizado para mejorar el desempeño de aplicaciones que demandan gran cantidad de cómputo. Algunas de estas aplicaciones

en general se caracterizan porque realizan gran cantidad de operaciones de cálculo en la solución de problemas de ciencias e ingeniería, por ejemplo: modelado del clima, simulaciones físicas, entre otras.

2.3. Elementos de un cluster

En los *clusters*, cada computadora está conformada por una tarjeta madre, unidad central de procesamiento (CPU) o procesador (P), memoria (M) y adaptador de red (AR) donde se procesarán los datos de una aplicación dada. Existen dos tipos de *clusters*: homogéneo y heterogéneo. En un homogéneo las computadoras tienen las mismas características, tales como, velocidad del procesador, tamaño de la memoria, velocidad del adaptador de red, entre otras; y en el heterogéneo algunas difieren. Otro elemento de *cluster* es el *software* especializado, que ayudará a distribuir el trabajo entre las computadoras. Para distribuir el trabajo entre las computadoras se requiere que se interconecten entre sí, para que se puedan comunicar unas con otras. Y así, brindar un correcto servicio (Figura 2.4).

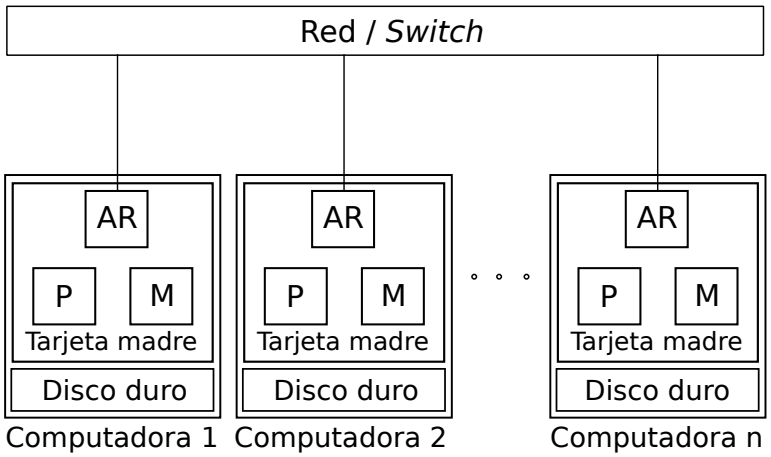

Figura 2.4: Arquitectura de un *Cluster*

2.3.1. CPU

La CPU o procesador de propósito general, realiza trabajos de oficina, casa o ingeniería, que cuenta con al menos uno núcleo. Los procesadores o núcleos contienen varias formas de paralelismo. La principal y más simple está en el uso de palabras de gran tamaño (algunos no lo consideran una forma de paralelismo). Para describirla por ejemplo en

la clase SIMD que pertenece a la arquitectura de Flynn [10] se puede mencionar MMX, 3DNow, SSE, AltiVec las cuales permiten realizar la misma operación en un grupo de números. Así que, el CPU usa paralelismo en la ejecución de una instrucción. La ejecución segmentada del estado de la instrucción (búsqueda, decodificación, ejecución, etc.) es otro ejemplo de paralelismo.

2.3.2. Memoria

El sistema de memoria es otro elemento en un sistema de cómputo paralelo que influye al tomar decisiones. Las limitaciones en el particionamiento de memoria proviene del hardware y el ambiente durante el tiempo de ejecución (ejemplo, sistema operativo, biblioteca de comunicación). Es importante determinar dónde y cómo los módulos de memoria son atendidos en el sistema de computadora.

2.3.3. Interconexión

Actualmente las redes de área local (llamadas LAN) ayudan a compartir recursos, facilitando la comunicación entre procesadores dentro de una institución, organización u hogar. Las redes LAN abarcan áreas relativamente pequeñas por lo general se emplean en edificios donde pueden conectarse con otras redes LAN mediante el internet. Por su gran demanda son cada vez más rápidas y más baratas, lo cual ha permitido que sean una excelente alternativa para ejecutar aplicaciones paralelas.

Las computadoras en una LAN son interconectadas por un medio de transmisión para comunicarse. Las redes LAN tradicionalmente operan con medios de transmisión tales como cable de par trenzado, cable coaxial, fibra óptica, portadora de rayos infrarojo o lasér, radio y microondas de frecuencia no comerciales. Las velocidades en la redes de área local van desde 10 Megabits por segundo (Mbps) hasta varios Gigabits por segundo (Gbps).

Las computadoras en una red LAN son conectadas internamente a enlaces o *switches*, que ayudan en la comunicación entre sí y evitan pérdida de datos así como colisiones entre los datos. Las interacciones entre las computadoras para coordinar, sincronizar o intercambiar datos se puede realizar a través de la comunicación de paso de mensajes sobre el enlace, cuando se emplea en un sistema multicomputadora.

La tarea principal de la interconección de la red es transmitir mensajes desde un nodo fuente a un nodo destino. El mensaje puede contener datos o una petición a memoria. El requerimiento en la red es realizar la transferencia correcta del mensaje lo mas rápido posible, aún si varios mensajes han sido transmitidos al mismo tiempo.

2.3.4. Software

Hasta ahora se ha definido la infraestructura o el *hardware* necesario para hacer cómputo paralelo. Como se mencionó anteriormente, los *clusters* ofrecen una excelente relación costo-rendimiento. Sin embargo, esto no es suficiente para hacer cómputo paralelo, es necesario además contar con cierto *software* como un sistema operativo que pueda manejar la computadora paralela y alguna metodología o modelo para desarrollar las aplicaciones paralela.

En cuanto a los sistemas operativos, es común que la arquitectura paralela tenga su propio sistema operativo como Solaris [37] o AIX [22], y ofrezcan sus propios programas para desarrollo. Algunos de estos programas son buenos, pero la restricción es que sólo pueden ser usados si se adquiere la costosa computadora paralela que ellos venden. Si estos programas se logran instalar en arquitecturas tipos *cluster*, éstos podrían ser incompatibles o son poco eficientes.

La otra opción es usar algún sistema operativo tipo Linux, que son de distribución libre y gratuitos. Algunos ejemplos de estos sistemas son: Debian [16] y Fedora [23], entre otros. Por ser de uso general, estos sistemas operativos resultan ser eficientes para máquinas paralelas como lo son los *clusters*. Adicionalmente existen compiladores y paquetería que proveen un mecanismo que permite coordinar el cómputo e intercambio de datos entre las tareas ejecutándose en el *cluster*. Las paqueterías más comunes son *Message Passing Interface* MPI [33] y *Parallel Virtual Machine* PVM [13].

2.4. Modelos de programación basado en el flujo de instrucciones y el número de datos

Las computadoras paralelas pueden ser divididas en dos categorías: flujo de instrucciones y datos. La primera está basada en los principios de la computadora de von Neumann [6], excepto que múltiples instrucciones pueden ser ejecutadas en algún tiempo dado. El flujo de datos no tiene un registro para activar las instrucciones o un lugar de control. El control es totalmente distribuido donde se lleva a cabo la activación de la instrucción por medio de un evento [25].

La clasificación más popular de arquitecturas de computadoras paralelas es la de Flynn publicada en 1966 [10]. Esta taxonomía de las arquitecturas está basada en la clasificación del flujo de datos e instrucciones en un sistema. Con estas consideraciones, Flynn clasifica los sistemas en cuatro categorías [29]:

- *Single Instruction, Single Data* (SISD). Conocidas como computadoras de series escalares, proveniente de la arquitectura de Von Neumann. En ella se ejecuta una sola instrucción y cuenta con un registro que se denomina contador de pro-

grama. Dicho contador se utiliza para la ejecución secuencial del programa. Las instrucciones se almacenan en la memoria y el contador de programa apunta a la siguiente instrucción para su ejecución (Figura 2.5).

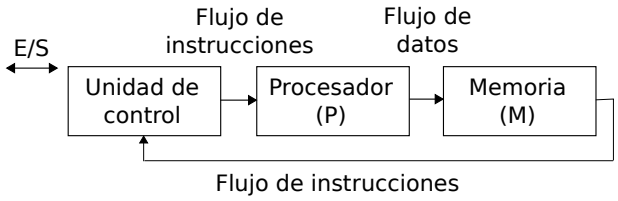

Figura 2.5: Arquitectura SISD

- *Multiple Instruction, Single Data* (MISD). Los procesadores vectoriales son clasificados con frecuencia en este modelo. Cada procesador tiene una memoria privada de programa, pero sólo un acceso a la memoria global de datos. En cada paso, cada unidad de procesamiento obtiene los mismo datos de la memoria de datos y carga una instrucción de la memoria del programa. Estas instrucciones posiblemente diferentes son ejecutadas en paralelo. Según Fayez [12] como ejemplo de este tipo de arquitectura paralela podemos mencionar a las redes neuronales y las máquinas de flujo de datos.

- *Single Instruction, Multiple Data* (SIMD). Esta arquitectura manipula una sola instrucción sobre un grupo de datos diferentes al mismo tiempo. Varios procesadores son manipulados por una única unidad de control que es la que distribuye el trabajo a los procesadores y espera, en todo caso el resultado de la ejecución. Al igual que las MISD, las SIMD soportan procesamiento vectorial (matricial) asignando cada elemento del vector a una unidad funcional para el procesamiento concurrente. Es considerado el método más simple de paralelismo y hoy en día es el más común. Las aplicaciones más comunes para esta arquitectura generalmente se utilizan en aplicaciones científicas y de ingeniería (Figura 2.6).

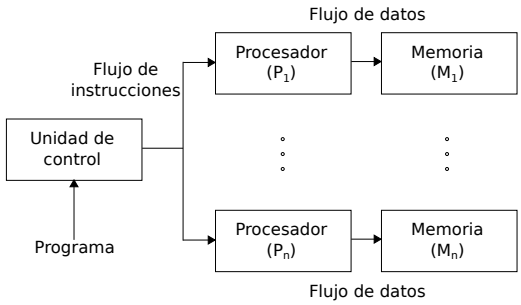

Figura 2.6: Arquitectura SIMD

- *Multiple Instruction, Multiple Data* (MIMD). Son consideradas arquitecturas complejas, sin embargo potencialmente ofrecen una mayor eficiencia en las operaciones concurrentes. El trabajo concurrente se realiza cuando se cuenta con varios procesadores para la ejecución operando simultáneamente y además hay varios programas (procesos) ejecutándose también al mismo tiempo (Figura 2.7).

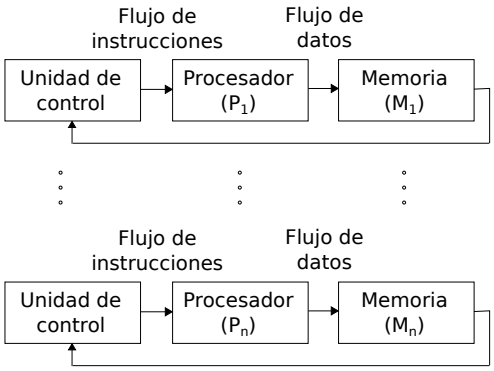

Figura 2.7: Arquitectura MIMD

El modelo SPMD (del inglés: *Simple Program Multiple Data*) es el más usado para desarrollar aplicaciones paralelas en *clusters* de computadoras. En este modelo el mismo código del programa de la aplicación es ejecutado en todos los procesadores. Los datos de la aplicación son divididos y distribuidos en los procesadores donde fue ejecutado el programa. Así, cada procesador ejecuta básicamente la misma pieza de código pero sobre una parte diferente de los datos. El particionamiento de los datos en este modelo también es conocido como paralelismo geométrico (debido a que los datos generalmente son divididos en bloques del mismo tamaño), descomposición en el dominio, o paralelismo en datos.

Los procesos de una aplicación paralela generalmente requieren compartir los datos entre ellos, por lo que deben comunicarse y sincronizarse para acceder a los datos. Para realizar la comunicación entre los procesos de un *cluster* comúnmente se usa el ambiente de programación de paso de mensajes. Este ambiente de programación está basando en una biblioteca que se ejecuta en el espacio de dirección del proceso. Además, la biblioteca provee un conjunto de funciones o primitivas que permiten al programador crear aplicaciones paralelas.

2.5. Programación con paso de mensajes

Actualmente, el modelo de programación predominante para el cómputo paralelo en *clusters* con memoria distribuida es el paso de mensajes. Como se recordará, en un *clus-*

ter los nodos tienen su memoria local y ninguno puede leer o escribir en la memoria de otro; no existe una memoria global accesible para todos los procesos que ejecutan la aplicación. El intercambio de los datos se realiza haciendo una copia (usando mensajes de envío y recepción), para transferir datos (Figura 2.8), por ejemplo: desde la memoria local del proceso A a la memoria local del proceso B, A debe enviar un mensaje conteniendo el dato de B, B debe recibir el dato en un *buffer* de su memoria local [29]. Un mensaje puede ser una instrucción, dato, sincronización o una señal de interrupción. Esta forma de comunicación es la más simple del envío de un mensaje en una aplicación paralela que comúnmente se conoce como punto a punto. Por otro lado, si una aplicación es divida en procesos; cada uno es ejecutado en procesadores o núcleos distintos. Si el número de procesos es más grande que el número de procesadores o núcleos entonces un proceso compartirá tiempo de ejecución con otro [9].

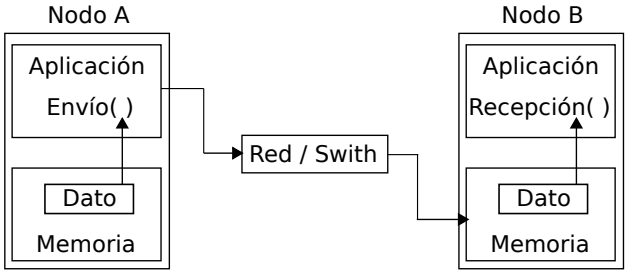

Figura 2.8: Ejemplo de un sistema con paso de mensajes

La comunicación punto a punto está compuesta de dos operaciones: una transmisión y una recepción. La operación de transmisión puede ser síncrona o asíncrona. La transmisión síncrona (envío bloqueante) termina solamente cuando el receptor emite un mensaje que indica que está listo para la recepción. El emisor termina la función de envío y continua con su ejecución, mientras los datos son transmitidos a bajo nivel. El proceso destino iniciará la recepción siempre y cuando tenga espacio suficiente en el *buffer* de recepción, si no tiene espacio suficiente no emitirá el mensaje que indica que está listo y por lo tanto permanecerá bloqueado (Figura 2.9). La transmisión asíncrona (envío no bloqueante) termina tan pronto como el mensaje correspondiente se haya entregado al sistema de comunicación. El emisor sigue su ejecución aún cuando no se haya emitido un mensaje por parte del receptor indicando que está listo. Además de este tipo de comunicación, existe la comunicación colectiva que facilita la transmisión entre procesos.

Muchos de los sistemas de paso de mensajes proporcionan operaciones colectivas que permiten que más de dos nodos se comuniquen entre sí. Las operaciones colectivas pueden ser escritas por el programador usando las operaciones básicas de punto a punto. Una transferencia colectiva involucra a más de un emisor o a más de un receptor. De esta manera, un mensaje puede ser transmitido desde un solo proceso a varios procesos, un proceso puede recolectar un conjunto de datos al recibir mensajes de varios procesos

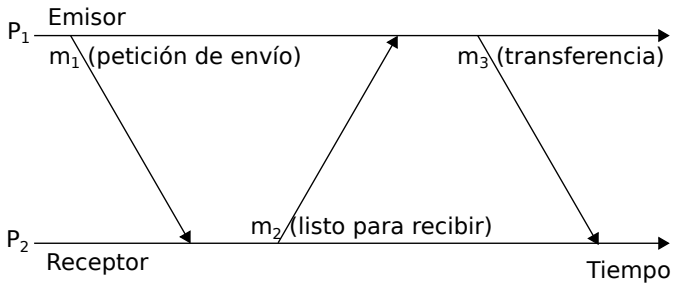

Figura 2.9: Protocolo de envío y recepción bloqueante

para formar uno sólo. A partir de las operaciones colectivas pueden formarse otras operaciones dependiendo de las necesidades que se tengan [31].

Las comunicaciones punto a punto y las colectivas puede emplearse en una gran variedad de ambientes de programación paralela. Los ambientes de programación paralela proporcionan una biblioteca de comunicación de paso de mensajes que manipulan componentes a bajo nivel de una forma muy transparente para el desarrollador. El desarrollador no se tiene que preocupar por el uso de sockets para transmitir mensajes entre los procesos ya que esta forma de transmisión no es muy adecuada debido a que los sockets son de un nivel de abstracción muy bajo.

2.6. Ambientes de programación paralela

2.6.1. PVM

PVM (*Parallel Virtual Machine*) fue el primer ambiente de paso de mensajes ampliamente aceptado que proporcionó portabilidad e interoperabilidad en un conjunto de nodos heterogéneos conectados por una red que aparenta un solo nodo. Los nodos en la red pueden ser uni-procesador, multinúcleo o *clusters* ejecutando *software* de PVM.

PVM tiene dos componentes: una biblioteca de rutinas y un demonio que reside en todos los nodos de la máquina virtual. También, ofrece una consola que permite iniciar, examinar o alterar la máquina virtual en cualquier momento durante la operación del sistema. Antes de ejecutar una aplicación PVM, el usuario necesita iniciar un demonio en cada nodo, de esta manera se creará la máquina virtual paralela. Las aplicaciones PVM necesitan ser ligadas con la biblioteca PVM, la cual contiene funciones de comunicación de punto a punto, comunicaciones colectivas, creación dinámica de tareas, coordinación de tareas y modificación de la máquina virtual [31]. Una aplicación en una máquina virtual paralela puede tomar diferentes estructuras. La estructura más común

es el grafo de estrella, donde el nodo intermedio en la estrella es llamado maestro y el resto de los nodos son llamados esclavos. En este modelo, el maestro inicializa la tarea de activar a todos los esclavos. Una estructura de árbol es otra forma de una aplicación PVM [9].

La principal desventajas de PVM es que su rendimiento no es tan bueno como los otros sistemas de paso de mensajes como MPI, debido a que PVM sacrifica rendimiento por flexibilidad al permitir heterogeneidad en la arquitectura de los nodos.

2.6.2. MPI

MPI (*Message Passing Interface*) es un estándar para paso de mensajes que ha sido desarrollado por un comité integrado por representantes de laboratorios de investigación, universidades e industriales. El estándar de MPI define la interfaz de usuario y la funcionalidad para un amplio rango de capacidades de paso de mensajes. Desde su primera versión MPI-1 en 1994 donde se definen las operaciones de comunicación estándar. MPI se convirtió en un estándar reconocido y el sistema de paso de mensajes la elección de muchos desarrolladores. Esto se debe tanto al rendimiento demostrado, como a su extensa colección de rutinas para soportar los patrones de comunicación ocurridos comúnmente [14]. La segunda versión MPI-2 en 1998 [15], agregan varias capacidades al estándar base, las más notables son: el control dinámico de procesos y E/S compartida.

El principal objetivo de MPI, como la mayoría de los estándares, es un grado de portabilidad a través de distintas computadoras. Esto significa que el mismo código fuente puede ser ejecutado en una gran variedad de computadoras siempre que la biblioteca de MPI esté disponible. Otro tipo de compatibilidad ofrecida por MPI, es la capacidad de ejecutarse transparentemente en sistemas heterogéneos, donde el usuario no necesita preocuparse si el programa está enviando mensajes entre procesadores de distintas o similares arquitecturas. La implementación de MPI tiene una amplia biblioteca con las funciones adecuadas que automáticamente realizará lo necesario para efectuar la conversión y utilizará el correcto protocolo de comunicación [33].

MPI consiste en una biblioteca que provee un conjunto de funciones para especificar el paso de mensajes entre procesos. Un requerimiento importante en todos los sistemas de paso de mensajes es garantizar una comunicación segura donde los mensajes no relacionados sean separados de otros. MPI define el concepto de comunicadores el cual combina contextos de mensajes y grupos de tareas para proporcionar mensajes seguros. Los comunicadores pueden ser clasificados en intra-comunicadores para operaciones dentro de un grupo de tareas (Figura 2.10) e inter-comunicadores para operaciones entre diferentes grupos de tareas. Donde se pueden realizar diferentes tipos de comunicaciones.

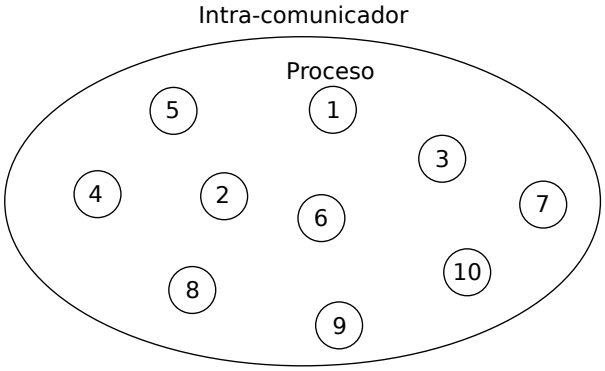

Figura 2.10: Esquema de intra-comunicador

2.6.3. Tipos de comunicaciones

MPI provee distintas primitivas de comunicación de punto a punto, bloqueantes y no bloqueantes y tipos de datos derivados. También provee diferentes rutinas de comunicación colectiva entre tareas pertenecientes a un grupo. Otras funciones incluyen manejo de topologías de procesos, funciones de chequeo y control de estado del medio ambiente [31].

2.6.3.1. Comunicación punto a punto

El mecanismo de comunicación básico de MPI es la transmisión de datos entre un par de procesadores, de un lado el emisor y del otro el receptor, comúnmente llamada comunicación punto a punto. En la comunicación punto a punto es posible transmitir valores escalares y arreglos continuos de valores para tipos de datos definidos por MPI. Además, MPI provee un método para el desarrollador, en el cual es posible definir un patrón de datos no continuos y usarlo en funciones de MPI. Estos patrones son llamados tipos *derivados* y pueden ser creados por una serie de funciones para formar virtualmente patrones arbitrarios de datos y combinación de tipos de datos. Además, la comunicación punto a punto es la base para la creación de otros tipos de comunicaciones que ofrece MPI.

La operación de envío en el emisor es realizada por la función MPI_Send donde se especifica el *buffer* que contiene los datos a transmitir, la cantidad de datos a transmitir, el tipo de dato, el proceso destino que recibirá los datos, una etiqueta para identificar cada mensaje y por último, un comunicador el cual pertenece a un grupo de procesos. La sintaxis es la siguiente:

MPI_Send(*buffer*, contador, tipo de dato, destino, etiqueta, comunicador)

La función de recepción cuenta con los siguientes parámetros, un *buffer* donde se copiarán los datos provenientes del emisor, la cantidad de datos a transmitir, el tipo de dato, un identificador de quien transmite, una etiqueta para identificar cada mensaje, el comunicador y por último información del estado de la transmisión. La sintaxis es la siguiente:

MPI_Recv(*buffer*, contador, tipo de dato, emisor, etiqueta, comunicador, estado)

Para transmitir mensajes entre dos procesadores en MPI, el proceso de transmisión interno consta de tres pasos [29]:

1. Los datos a ser transmitidos son copiados desde el *buffer* de envío especificado como parámetro a un *buffer* del sistema de MPI. El mensaje es ensamblado agregándole una cabecera de MPI con información en el proceso de envío, en el proceso de recepción la etiqueta y el comunicador es usado.

2. El mensaje se transmite por medio de la red desde el proceso emisor al proceso receptor.

3. En el receptor, los datos de entrada son copiados desde el *buffer* del sistema al *buffer* de recepción especificado en MPI_Recv.

Las funciones MPI_Send y MPI_Recv son operaciones asincrónicas bloqueantes. Esto significa que la operación MPI_Recv puede iniciar aún cuando la operación MPI_Send no ha iniciado. La ejecución de la operación MPI_Recv es bloqueada hasta que el *buffer* de recepción contenga los datos del emisor. Similarmente, la operación MPI_Send puede iniciar aún cuando la correspondiente operación de MPI_Recv no ha iniciado (Figura 2.11). La operación MPI_Send es bloqueada hasta que el *buffer* de envío pueda ser usado nuevamente. También existen operaciones no bloqueantes como MPI_Isend, MPI_Irecv, para más información de éstas y otras operaciones consultar [15]. El comportamiento exacto depende de la biblioteca de MPI usada. Los siguientes dos comportamientos pueden ser observados [33]:

1. Si el mensaje es enviado desde el *buffer* de envío sin usar un *buffer* del sistema interno, entonces la operación MPI_Send es bloqueada hasta que llegue un mensaje indicando que el receptor está listo. Esto requiere que el receptor haya iniciado la operación MPI_Recv.

2. Si el mensaje es copiado a un *buffer* del sistema interno en tiempo de ejecución, el emisor puede seguir operando tan pronto como el copiado de los datos haya finalizado. Por lo tanto, la operación MPI_Recv correspondiente no necesariamente tuvo que ser inicializada. Esta forma de operar tiene la ventaja de que

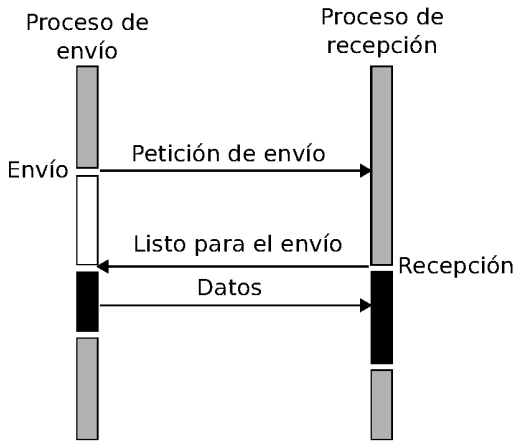

Figura 2.11: Comunicación punto a punto

el emisor no está bloqueado por mucho tiempo pero tiene la desventaja que el *buffer* del sistema requiere espacio adicional en memoria y el copiado hacia el *buffer* requiere tiempo adicional de ejecución.

Ejemplo: En la Figura 2.12 se muestra el código de un programa en MPI en el cual el proceso con identificador 0 (*rank*) usa la función MPI_Send para enviar un mensaje al proceso con identificador 1. Este proceso 1 usa la función MPI_Recv para recibir el mensaje. En el programa se muestra como todos los procesos participantes ejecutan el mismo programa pero diferentes procesos ejecutan diferentes partes del programa.

2.6.3.2. Comunicación colectiva

La comunicación colectiva es un conjunto de funciones que ayudan a distribuir los datos entre los procesos, una función colectiva es ejecutada por todos los procesos en el grupo con los argumentos correspondientes. Uno de los argumentos clave es un intra-comunicador que define el grupo de procesos participantes y provee un dominio de comunicación para la operación. En la ejecución de una función colectiva un proceso es definido como maestro (*root*), donde algunos argumentos son importantes sólo para el maestro y son ignorados por los otros procesos excepto el maestro y viceversa. Una de las actividades que realiza el proceso maestro es transmitir un bloque de datos a todos los procesos o dividir un bloque en partes iguales y transmitir cada parte a los procesos, dependiendo de la función que se use. También existen funciones colectivas donde varía la cantidad de datos a transmitir a cada proceso como se explica en [33].

MPI soporta una gran variedad de funciones colectivas. Las operaciones básicas soportadas son: *broadcast*, *scatter*, *gather*. En un *broadcast*, un proceso envía el mismo

```
#inclide <stdio.h>
#include <mpi.h>
#define root 0
#define cont 100
int main(int argc, char *argv[])
{
  int rank;
  int buffer[cont];
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if(rank == root)
    MPI_Send(buffer, cont, MPI_INT, 1, tag, MPI_COMM_WORLD);
  else
    MPI_Recv(buffer, cont, MPI_INT, 0, tag, MPI_COMM_WORLD, &
       status);
  MPI_Finalize();
  return 0;
}
```

Figura 2.12: Código en MPI para la transmisión de los datos entre dos nodos

mensaje a cada miembro del grupo. Una operación *scatter* permite que un proceso envíe un mensaje diferente a cada miembro. Una operación *gather*, establece que un proceso reciba un mensaje de cada miembro del grupo. Las operaciones básicas pueden ser combinadas para formar funciones más complejas.

Aunque las comunicaciones colectivas son descritas en términos de envío de mensajes directamente desde el emisor(es) al receptor(es), las implementaciones pueden usar un patrón de comunicación donde los datos son reenviado a través de nodos intermedios. Así, puede usarse un árbol de profundidad logarítmica para implementar el *broadcast*, en lugar de enviar datos directamente desde el maestro a cada proceso. Los mensajes pueden ser reenviados a procesos intermedios y dividir (para *scatter*) o concatenar (para *gather*). Una implementación óptima de comunicación colectiva tomará ventaja de las especificaciones de la red de comunicación y usará diferentes algoritmos, de acuerdo al número de procesos participantes y la cantidad de datos a transmitir [33].

Broadcast. MPI provee la siguiente función para transmitir un mensaje desde el proceso maestro a todos los procesos del grupo (Figura 2.13).

MPI_Bcast(mensaje, cont, tipo_dato, maestro, comunicador)

Donde el maestro denota el proceso que envía el bloque de datos. Este proceso provee el bloque de datos que será enviado en el parámetro mensaje. Los otros procesos especifican en mensaje su *buffer* de recepción. El parámetro "cont"denota el número de elementos en el bloque de datos.

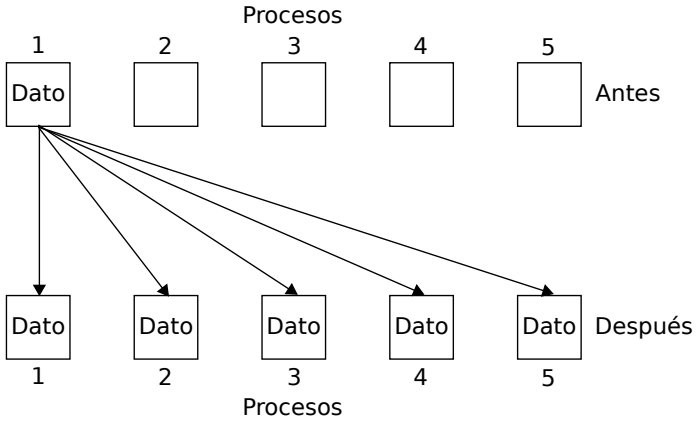

Figura 2.13: *Broadcast*

Scatter. En esta operación el proceso maestro provee un bloque de datos a cada proceso participante. Ejecutando la operación *scatter*, los bloques de datos son distribuidos a todos los procesos (Figura 2.14). La operación es realizada con la llamada a la función:

MPI_Scatter(sendbuf, sendcount, tipo_dato, recvbuf, recvcount, tipo_dato, maestro, comunicador)

Donde sendbuf es el *buffer* de envío que provee el proceso maestro el cual contiene un bloque de datos para cada proceso en el comunicador. Cada bloque contiene los elementos a transmitir así como el tipo de dato. Los bloques de datos son ordenados respecto al identificador del proceso en el comunicador. Los bloques se reciben en el *buffer* de recepción proveído por el correspondiente proceso. Cada proceso participante incluido el maestro debe proveer el *buffer* de recepción.

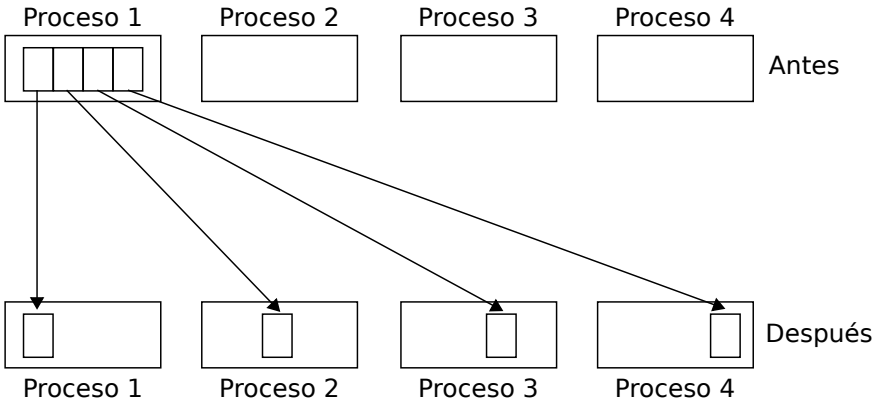

Figura 2.14: Ejecución de la función *Scatter* desde el proceso 1

Gather. Cada proceso provee un bloque de datos a un proceso maestro. El proceso maestro colecta cada bloque y va llenando su *buffer* de recepción con bloques de datos (Figura 2.15). La operación *gather* es realizada llamando la siguiente función de MPI:

MPI_Gather(sendbuf, sendcount, tipo_dato, recvbuf, recvcount, tipo_dato, maestro, comunicador)

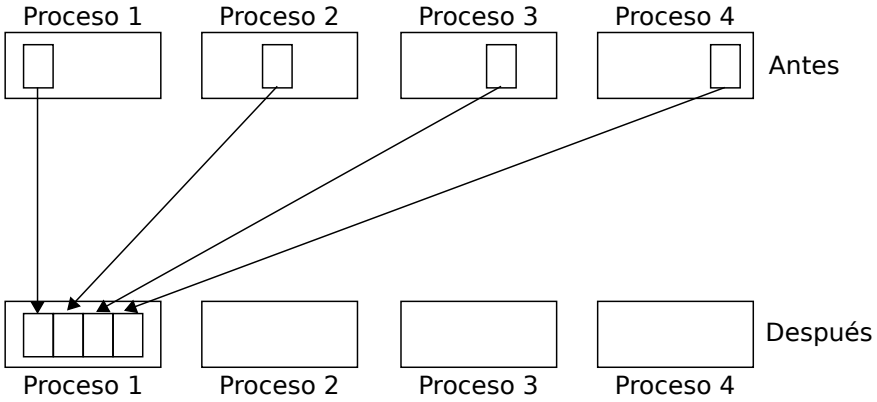

Figura 2.15: Ejecución de la función *gather* en el proceso 1

Aunque las funciones MPI_Scatter y MPI_Gather sólo trabajan para elementos divisibles, existen otras como MPI_Scatterv y MPI_Gatherv donde se trabaja con elementos indivisibles, esto significa que se pueden enviar bloques de datos a cada proceso de diferente tamaño. Para más detalle y otras operaciones colectivas consultar [33].

2.7. Resumen

Los *clusters* de computadoras hoy en día son una excelente alternativa para ejecutar aplicaciones paralelas porque ofrecen un beneficio extraordinario en costo-rendimiento. En las últimas décadas se investigó el uso de las LAN para explotar el paralelismo debido a su bajo costo y gran demanda. Estas redes permiten compartir recursos y facilitar la comunicación entre un gran número de procesadores. Estos beneficios han propiciado el desarrollo de los *clusters*. Un *cluster* consiste de una colección de computadoras o nodos independientes conectados conjuntamente en una LAN. Los nodos de un *cluster* pueden ser computadoras independientes o un sistema multiprocesador. Los ambientes de programación paralela utilizados en los *clusters* son herramientas portables y eficientes para el desarrollo de aplicaciones paralelas.

El tipo de aplicaciones que son comúnmente ejecutadas en los *clusters* se caracterizan por ejecutar el mismo programa en los procesadores, pero con diferentes datos. Este

modelo de programación es conocido como SPMD. Las aplicaciones desarrolladas con el modelo SPMD pueden ser muy eficientes si los datos son uniformemente distribuidos a los procesadores y el sistema es homogéneo.

Una de las herramientas que usualmente se usa para desarrollar aplicaciones paralelas es MPI, que consiste en una biblioteca que provee un conjunto de funciones para especificar el paso de mensajes entre los procesadores. MPI cuenta con funciones que hacen referencia al tipo de comunicación. Los tipos de comunicaciones que tiene MPI son: punto a punto y colectivas. Las comunicación punto a punto es cuando un par de procesadores se comunican e intercambian información. Las comunicación colectiva es la comunicación entre dos o más procesadores e intercambian información entre ellos. Una de las incógnitas que prevalece en el cómputo paralelo es ¿Cuál es el tiempo de ejecución de un algoritmo paralelo?, para responder a esta pregunta existen varios modelos que estiman el tiempo de cómputo y comunicación considerando varios factores que afectan el tiempo de ejecución.

Capítulo 3

Estado del arte: Modelos de estimación del tiempo de procesamiento en *clusters* de computadoras

En este capítulo se describen los trabajos previos que se han realizado para estimar los tiempos de comunicación y de cómputo. En el tiempo de comunicación se proponen varios parámetros que influyen en la transmisión de datos. Además, se presentan algunos trabajos que hacen comparación de las investigaciones previas para determinar cual tiene mejor desempeño. En el tiempo de cómputo se presenta la ley de Amdahl para logar una buena estimación.

3.1. Introducción

Estimar el tiempo de ejecución de una aplicación paralela en un *cluster*, ayuda a diseñar estrategias de paralelización para minimizar el tiempo de ejecución, además, se pueden reducir costos de procesamiento y de energía. El tiempo de ejecución involucra tanto el tiempo de cómputo como de comunicación. Es importante estimar el tiempo cómputo y comunicación, debido a que entre más procesadores involucrados en la solución de un problema, mejor será el tiempo de procesamiento, pero incrementará el tiempo de comunicación entre los procesadores. Encontrar un balance entre los dos tiempos no es una tarea trivial, por lo que se han desarrollado varios trabajos que estiman los tiempos. En el tiempo de cómputo Amdahl define los límites de aceleración que una aplicación puede alcanzar. En el tiempo de comunicación, se proponen varios modelos, donde algunos estiman el tiempo con un pequeño conjunto de parámetros y otros

proponen parámetros más detallistas, para los patrones de comunicación punto a punto y colectivas.

3.2. Tiempo de cómputo

En el modelo de cómputo se observa la duración del procesamiento de una aplicación paralela, donde el tiempo de cómputo se ve afectado por la velocidad del procesador de un nodo en el *cluster*, así como el número de operaciones que realiza dicho procesador. A continuación se presenta la descripción del modelo utilizado en un trabajo previo.

3.2.1. Ley de Amdahl

La propuesta presentada por Amdahl en [2] define los límites de la aceleración que puede ser alcanzada teóricamente por una aplicación dada, en función del tamaño de la fracción secuencial de la aplicación. La aceleración general depende de la aceleración de un componente en particular y de cuantos componentes participan. La ecuación para estimar la aceleración está dada por:

$$S = \frac{1}{(1-f) + f/P} \tag{3.1}$$

Donde f es la fracción de la porción paralelizada de la aplicación y P es el número de procesadores o núcleos disponibles. Es claro que de acuerdo a la ley de Amdahl cualquier aplicación con una fracción secuencial tendrá un límite superior, es decir, que tan rápido puede correr, independientemente de la cantidad procesadores o núcleos disponibles para su ejecución.

Intuitivamente, la ley de Amdahl es fácil de deducir. Teniendo P procesadores disponibles, el tiempo necesario para ejecutar la porción paralela es:

$$f/P \tag{3.2}$$

Y el tiempo total en P procesadores está dado por:

$$T(P) = (1-f) + f/P \tag{3.3}$$

Suponiendo que:

$$T(1) = 1 \tag{3.4}$$

3.3. Tiempo de comunicación

La comunicación es un factor que limita y degrada el rendimiento de las aplicaciones paralelas, por lo tanto, es importante contar con un modelo que estime el tiempo de comunicación. Hoy en día, existen varios trabajos que proponen modelos para estimar el tiempo de comunicación. A continuación se exponen los trabajos previos que incluyen los modelos básicos que se han propuestos, algunas mejoras de los modelos básicos, así como la comparación entre ellos y por último, algunos trabajos que predicen el tiempo de ejecución de una aplicación paralela.

3.3.1. Modelos básicos del rendimiento de las comunicaciones

En esta sección se presentan los trabajos previos donde proponen modelos para estimar el tiempo de comunicación. Además, se describen los parámetros que usa cada modelo y cómo han ido evolucionando para obtener una buena estimación. Al finalizar la sección se tendrá conocimiento de los conceptos básicos para las secciones siguientes.

3.3.1.1. LogP: Hacia un modelo realista de cómputo paralelo

El modelo LogP [7] expresa las características de la comunicación entre cada nodo con un pequeño número de parámetros. Los parámetros expresan las características de la comunicación entre cada nodo, los cuales son, latencia de cada mensaje *(L)*, el tiempo que un procesador consume en la transmisión o la recepción de un mensaje *(o)*, el tiempo mínimo entre mensajes consecutivos al transmitir o recibir *(g)* y el número de procesadores *(P)*. Los parámetros antes mencionados son medidos al realizar varias transmisiones o recepciones y toman un promedio de los tiempos obtenidos. En la Figura 3.1 se muestra los parámetros involucrados.

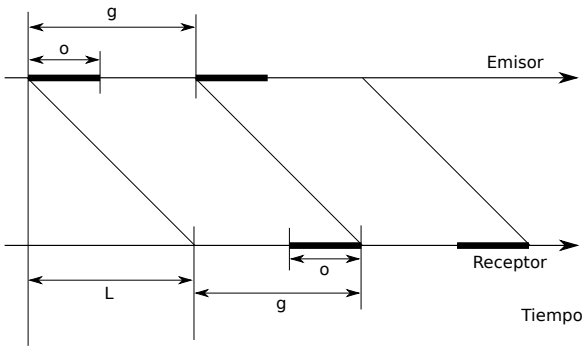

Figura 3.1: Parámetros del modelo LogP

3.3.1.2. LogGP: Incorporando mensajes grandes al modelo LogP para cómputo paralelo

El modelo LogGP [1] es una extensión del modelo LogP [7]. En LogGP se propone una solución a la baja precisión de LogP en la predicción del tiempo de transmisión cuando se transmiten mensajes grandes. La solución que proponen en [1] es dividir el parámetro *(g)* del modelo LogP en un elemento constante *(G)* y un elemento es proporcional al tamaño del mensaje. En el modelo que proponen asumen que el intervalo del mensaje es lineal al tamaño del mensaje, lo cual se cumple para mensajes grandes. En la Figura 3.2 se representa los parámetros del modelo que proponen.

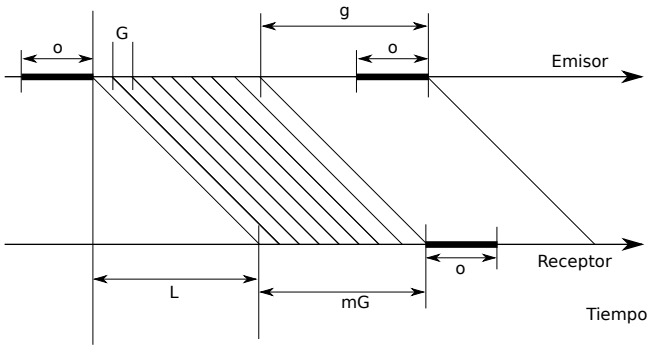

Figura 3.2: Parámetros del modelo LogGP

3.3.1.3. Ancho de banda eficiente en comunicaciones colectivas para sistemas de *clusters* de área amplia

El modelo de PLogP [19] es otra extensión del modelo LogP [7]. El modelo PLogP toma en consideración los mensajes de longitud arbitraria con la finalidad de obtener tiempos estimados de comunicaciones más precisos, considerando los siguientes parámetros: el *overhead* del envío (o_s), recepción (o_r) y el intervalo entre mensajes *(g)* que están en función al tamaño del mensaje. La relación entre los parámetros antes mencionados y el tiempo de comunicación se ilustra en la Figura 3.3.

Este modelo evita la dificultad involucrada en la descripción del intervalo del mensaje de acuerdo con el tamaño del mensaje. Por lo tanto, establece que el intervalo depende del tamaño del paquete máximo y mínimo de la red física y no puede ser formulado simplemente por una ecuación lineal. Además, los parámetros del modelo PLogP son medidos independientemente del tamaño del mensaje, por lo tanto el modelo puede expresar la correlación entre el intervalo de los mensajes y el tamaño del mensaje, tanto para mensajes grandes como para los pequeños.

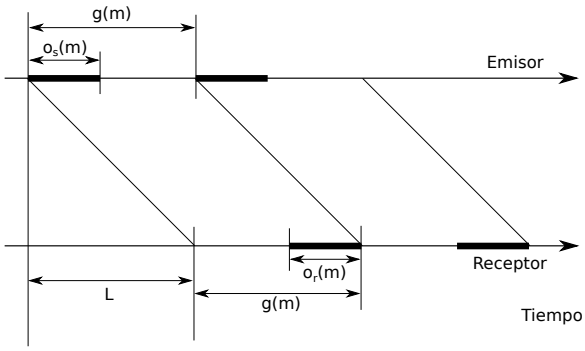

Figura 3.3: Parámetros del modelo PLogP

3.3.2. Modelo de comunicación realista para cómputo paralelo en *cluster*

En el artículo [35] proponen un modelo para cómputo paralelo en *cluster*. El modelo es etiquetado como una herramienta para el análisis del rendimiento y el diseño de algoritmos. Además, resumen los eventos de la comunicación para la transmisión de los datos en dos tipos: local y remota, por último exponen explícitamente los problemas de la contención, donde son capturados en los parámetros propuestos.

A continuación se listan los parámetros en la Tabla 3.1 que toman en consideración para la transferencia remota:

Tabla 3.1: Parámetros de la transferencia remota

Parámetro	Descripción
P	Número de procesadores
O_s	*Overhead* de transmisión
g_s	Intervalo de tiempo entre consecutivas transmisiones
L	Latencia de la red
g_r	Intervalo de tiempo mínimo entre consecutivas recepciones
B_L	*Buffer* de la red
O_r	*Overhead* de recepción asíncrona
U_r	*Overhead* de recepción en el destino

En la Tabla 3.2 se listan los parámetros que se consideran al realizar una transmisión local:

Una característica importante que tiene este modelo es que no usa valores constantes, los parámetros expuestos con anterioridad son capturados en tiempo ejecución. La captura de los parámetros tiene un costo adicional, que incluye la longitud del mensaje,

Tabla 3.2: Parámetros en la transferencia local

Parámetro	Descripción
M_{ctc}	De cache a cache
M_{ctm}	De cache a memoria
M_{mtm}	De memoria a memoria

la carga de tráfico y el factor de contención. Con base en lo anterior, plantean que el conjunto de parámetros propuesto permite realizar el análisis en alguna arquitectura dada.

Los experimentos fueron realizaron con la operación *broadcast* basada en árbol para demostrar el retardo causado por la contención durante simultáneas operaciones de envío y recepción. Para demostrar la eficiencia del modelo, desarrollaron un algoritmo *gather* óptimo con la capacidad de evitar pérdida de contención. Además, restringieron el número de emisores activos dentro de un límite superior e inferior, con base en lo anterior, su algoritmo redujo significativamente el costo de la comunicación.

3.3.3. Análisis y optimización de comunicaciones colectivas en un *cluster Beowulf*

En el trabajo presentado en [36] analizan el rendimiento de las operaciones colectivas tales como: *All-Gather, All-Reduce, Reduce-Scatter* en un *cluster beowulf*. El *cluster* consta de una red libre de contención con múltiples tarjetas de red por nodo, permitiendo el traslape de transmisiones de mensajes bajo ciertas circunstancias. Establecen que el tipo de red en un *cluster* puede afectar al modelo del rendimiento de las comunicaciones, por lo cual, desarrollaron una herramienta de simulación para entender el rendimiento de las operaciones de comunicación.

En la Tabla 3.3 se listan los parámetros usados.

El modelo que propusieron lo aplicaron en varios algoritmos y encontraron algoritmos que superaran el rendimiento de los implementados en LAM/MPI *(Local Area Multicomputer/Message Passing Interface)* por un factor de entre 2 y 3. Estos algoritmos son el *binary-exchange* y el *recursive halving*, donde obtuvieron buenos resultados muy cercanos al óptimo para mensajes pequeños, debido a su coeficiente α. Además, para mensajes grandes el algoritmo *ring* fue muy eficiente, dado que la red es libre de contención.

Además, investigaron los patrones de comunicación. En el caso, del patrón de comunicación *binary tree* el traslape de las comunicaciones ocurre al realizar consecutivos *broadcast*, resultando con un costo general significativamente menor que al tiempo esperado $p\ log_2\ p$ que de un mensaje punto a punto. En el patrón de comunicación

Tabla 3.3: Parámetros para estimar el tiempo de comunicación

Parámetro	Descripción
α	Costo de inicialización
β	Ancho de banda
p	Número de procesadores
n	Número de elementos para las operaciones *ag*, *ar* y *rs*
$S(n)$	Tiempo de envío para n elementos
$R(n)$	Tiempo de recepción de n elementos
t_{op}^{alg}	Tiempo de operación *(op)* usando algún algoritmo *(alg)*
ag	Operación *All-Gather*
ar	Operación*All-Reduce*
rs	Operación *Reduce-Scatter*

pipeline obtuvo mejores resultados y especialmente cuando se integra en aplicaciones donde algunas comunicaciones son por su naturaleza *pipeline*.

3.3.4. Caracterización del rendimiento de comunicaciones colectivas en *intra-cluster*

En el trabajo presentado por [3] no solo se enfocan en comunicaciones *inter-cluster* si no también en comunicaciones *intra-cluster* donde tienen como objetivo encontrar una buena estrategia de comunicación para los dos tipos de comunicación. Para encontrar la mejor estrategia de comunicación compararon distintas estrategias con los modelos de comunicaciones [7] [19], además, evalúan la precisión de los modelos y describen los cambios prácticos que pueden ser encontrados cuando se modelan comunicaciones colectivas.

En este trabajo presentan tres casos donde compararon la predicción del rendimiento de los modelos con los resultados reales para tres patrones de comunicaciones colectivas: uno a muchos *broadcast*, uno a muchos personalizado *scatter* y muchos a muchos *All-to-all*. Además, verificaron que los modelos desarrollados fueran lo suficientemente precisos para predecir el rendimiento de las comunicaciones colectivas, permitiendo la selección de la estrategia que mejor se adaptada a la red. A continuación se presentan las estrategias que usaron para cada patrón de comunicación: *Broadcast* (Tabla 3.4), *Scatter* (Tabla 3.5) y *All-to-all* (Tabla 3.6), y la descripción de los parámetros (Tabla 3.7).

Para modelar la operación "todos a todos", eligieron representar los efectos de la contención de la red como un factor linear. Los experimentos que realizaron demostraron que la suposición lineal que plantearon fue los suficientemente precisa para predecir el rendimiento de tal operación.

Tabla 3.4: Modelos de comunicación para el *Broadcast*

Estrategia	Modelo de comunicación
Flat tree	*(P - 1) * g(m) + L*
Flat tree rendezvous	*(P - 1) * g(m) + 2 * g(1) + 3 * L*
Segmented flat tree	*(P - 1) * (g(s) * k) + L*
Chain	*(P - 1) * (g(m) + L)*
Chain rendezvous	*(P - 1) * (g(m) + g(1) + 3 * L)*
Seg. chain (pipeline)	*(P - 1) * (g(s) + L) + (g(s) * (k - 1))*
Binary tree	$\leq \lceil log_2 P \rceil$ ** (2 * g(m) + L)*
Binomial tree	$\lfloor log_2 P \rfloor * g(m) + \lceil log_2 P \rceil * L$
Binomial tree rendezvous	$\lfloor log_2 P \rfloor * g(m) + \lceil log_2 P \rceil * (2 * g(1) + 3 * L)$
Seg. binomial tree	$\lfloor log_2 P \rfloor * g(s) * k + \lceil log_2 P \rceil * L$

Tabla 3.5: Modelos de comunicación para el *Scatter*

Estrategia	Modelo de comunicación
Flat tree	*(P - 1) * g(m) + L*
Chain	$\sum_{j=0}^{P-1} g(j * m) + (P - 1) * L$
Pipelined chain	*(P - 1) * (g(m) + L)*
Binomial tree	$\sum_{j=0}^{\lceil log_2 P \rceil - 1} g(2^j * m) + \lceil log_2 P \rceil * L$

Tabla 3.6: Modelos de comunicación para *All-to-all*

Estrategia	Modelo de comunicación
Límite superior	*(P - 1) * g(m) + (P - 1) * or(m) + L*
Límite inferior	*(P - 1) * os(m) + (P - 1) * or(m) + L*

Tabla 3.7: Parámetros que se consideran

Parámetro	Descripción
$g(m)$	Intervalo de un mensaje de tamaño m
L	Latencia de comunicación entre dos nodos
P	Número de nodos
s	Tamaño del segmento (en el caso de que se segmente el mensaje)
k	Número de segmentos
$g(s)$	Intervalo de un segmento con tamaño s
os	*Overhead* de envío (entrega del mensaje a la tarjeta de red)
or	*Overhead* de recepción

3.3.5. Un modelo de comunicación preciso de un *cluster* heterogéneo basado en una red ethernet

En el trabajo presentado por [20], exponen un modelo de comunicación de un conjunto de procesadores heterogéneos inter conectados por una red ethernet. El objetivo del modelo es realizar la predicción precisa de las contribuciones de las operaciones colectivas para el tiempo de ejecución de aplicaciones paralelas. El modelo que presentan toma en consideración el impacto de la heterogeneidad de los procesadores en el rendimiento de las operaciones colectivas. En el trabajo, se desarrollan los modelos analíticos para una comunicación punto a punto simple, múltiples comunicaciones punto a punto independientes, múltiples comunicaciones punto a punto de uno a muchos y por último una operación *broadcast*. Con base en el modelo de comunicación punto a punto crearon los otros modelos.

El tiempo de comunicación para una comunicación punto a punto en un entorno heterogéneo desde un nodo i a un j es dado por:

$$T_{ij}(M) = C_i + t_i M + C_j + t_j M + M/\beta_{ij} \tag{3.5}$$

Donde, C_i es el comunicador del *cluster*, $t_i M$ es el retardo por el procesamiento en el nodo fuente i, M/β_{ij} es el tiempo de transmisión del nodo i al j y M es el tamaño del mensaje.

El modelo que proponen determina el tiempo de comunicación con base en el tamaño del mensaje. Para estimar el tiempo de comunicación tomaron varios factores tales como, el retardo de la transmisión, la latencia del nodo fuente y la latencia del nodo destino. El retardo de la transmisión es el tiempo que transcurre entre el primer *bit* transmitido y el último *bit* capturado en el nodo destino. En este modelo el retardo de la propagación es ignorado por la naturaleza de la red. La latencia en el nodo fuente se debe al procesamiento fijo y al retardo en el procesamiento variable. El retardo del procesamiento fijo incluye la construcción de las cabeceras del mensaje mientras que el procesamiento variable se compone del copiado del mensaje. Por lo tanto, depende del tamaño del mensaje.

Con base en lo anterior realizaron los siguientes experimentos: comunicación punto a punto, la comparación es realizada con el modelo propuesto y el tiempo experimental. El par de nodos participantes son heterogéneos conectados por una red GigaEthernet, el tiempo de comunicación para mensajes pequeños es mínimo y la diferencia entre ellos es mínima. Al realizar una transmisión de un tamaño del mensaje de 1 KB, observaron un incremento en el tiempo de comunicación, debido a la comunicación estándar de MPI que requiere que el mensaje pase por un *buffer*. Esto puede ser un umbral donde el modelo está compuesto por dos ecuaciones. En múltiples comunicaciones punto a punto, *broadcast* y uno a muchos, fueron validadas con datos experimentales, donde los tiempos estimados son muy cercanos al tiempo experimental.

3.3.6. Análisis del rendimiento de las operaciones colectivas en MPI

En el artículo [26] analizan e intentan mejorar las comunicaciones colectivas *intra-cluster* en el contexto del paradigma de programación en MPI extendiendo los modelos de comunicación punto a punto, tales como: [17] [7] [1] [19]. El trabajo que realizaron fue comparar los modelos de comunicación paralela aplicados a operaciones colectivas en MPI *inter-cluster*, donde mostraron que aun cuando no se modela la congestión de la red directamente, todos los modelos proveen puntos de vista útiles de varios aspectos de los diferentes algoritmos y su rendimiento.

Los resultados que obtuvieron muestran que el modelo con menor precisión en la estimación del rendimiento de los algoritmos es [17], donde al transmitir mensajes de tamaño pequeños y medianos, el modelo hace la suposición que debe esperar el tiempo de la latencia antes de enviar el siguiente mensaje, el cual es demasiado pesimista. Para mensajes grandes, la suposición es que puede reenviar tan pronto como el primer byte del mensaje llega al receptor, lo cual es demasiado optimista. Por otro lado, la predicción de los modelos [1] [19] fue muy cercana, la cual puede ser usada para llegar a conclusiones similares. Sin embargo, el modelo [19] tiene parámetros más flexibles y su análisis sugiere que la predicción fue mucho más cercana a los resultados experimentales.

En este trabajo el propósito principal fue implementar y optimizar las operaciones colectivas, para ser usadas como una biblioteca para alguna distribución de MPI. Los experimentos y los análisis del rendimiento de algoritmos colectivos pueden ser usados para determinar el punto de intercambio entre los métodos disponibles. Por último, el tiempo de ejecución está basado en una tabla de valores estáticos, donde un método en particular puede ser seleccionado con base en el número de procesadores, el tamaño del mensaje y el proceso maestro.

3.3.7. Modelo del rendimiento de la red para TCP/IP basado en cómputo paralelo

En [24] proponen un nuevo modelo de comunicación llamado PlogPT, con el objetivo de predecir el rendimiento de las comunicaciones en un *cluster* donde los nodos se comunican por medio del protocolo TCP/IP sobre una red ethernet. El modelo que proponen es una extensión del modelo [19].

En el modelo PlogPT, la topología de la red es modelada como una conexión de árbol binaria donde una arista, un nodo intermedio y un nodo hoja son un enlace de red, un *switch* y una computadora, respectivamente. Los dos enlaces de los nodos intermedios tienen el mismo ancho de banda. El ancho de banda de los enlaces lo estiman ejecutando programas que realizan esa tarea. Además, sus experimentos para obtener el ancho de

banda, a diferencia de otros programas tradicionales, fue evaluada con patrones de comunicación bidireccional.

A diferencia del modelo PlogP [19], el intervalo en la red que expresa el tiempo de transferencia de un mensaje *(g)*, lo expresan como el intervalo $g_c(m, C)$ para cada comunicación *(C)* con un tamaño del mensaje *m*. $g_c(m, C)$ lo obtuvieron con base en el *throughput* $b_c(C, t)$ que expresa la comunicación *(C)* en un tiempo *t*, el *throughput* fue calculado con base en otras comunicaciones que comparten el mismo camino de comunicación. Además, otro factor que consideran es el tiempo de retransmisión T_{RTO}. El tiempo de retransmisión domina el tiempo de ejecución de las comunicaciones, donde la predicción se basa en una combinación *switches* intermedios y el mensaje en el *buffer* para todas las transmisiones desde una comunicación a otra. Como resultado, obtuvieron una mejora al estimar el tiempo de ejecución. En la figura 3.4 se ilustran todos los parámetros involucrados.

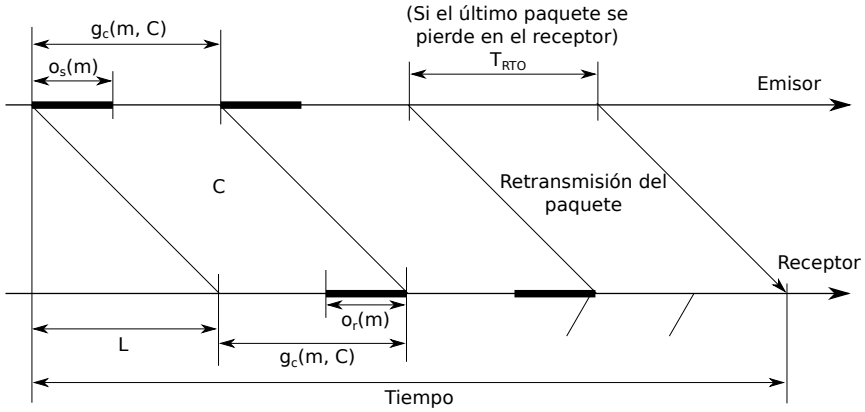

Figura 3.4: Parámetros del modelo PLogPT

El tiempo de ejecución de dos algoritmos de comunicación "todos a todos" han sido estimados y comparados con el tiempo de ejecución real. Los resultados muestran que todos los tiempos estimados son muy cercanos al tiempo real, mientras que el modelo PlogP no hace una estimación muy precisa.

3.3.8. Revisión de los modelos del rendimiento de la comunicación para cómputo en *clusters*

En el trabajo presentado en [21] analizan las restricciones de los modelos tradicionales, tales como: [17] [7] [1] [19] donde se ve afectada la predicción analítica del tiempo de ejecución de las operaciones de comunicación colectiva. En particular, dan a conocer que las contribuciones constantes y variables de los procesadores y la red no son separadas en los modelos antes mencionados. La separación de las contribuciones tienen diferente naturaleza y surgen de distintas fuentes que producen modelos más intuitivos y precisos,

pero los parámetros de tales modelos no se pueden obtener de sólo comunicaciones punto a punto, como usualmente se obtienen en los modelos tradicionales.

Los modelos tradicionales fueron estudiados detalladamente para analizar porque la predicción analítica tiene poca precisión. El problema común en estos modelos es que combinan contribuciones de diferente naturaleza y proveen el tiempo de ejecución de alguna operación de comunicación colectiva como una combinación de máximos y sumas de los parámetros de la operación punto a punto. En contraste, el modelo LMO que propusieron en [20], diseñado para *clusters* homogéneos y heterogéneos, demuestra que la separación de las contribuciones lleva a una predicción más intuitiva y precisa para el tiempo de ejecución de operaciones colectivas.

Por otra parte, el tiempo de ejecución de la comunicación fue obtenida con la ayuda del *benchmark* MPIBlib. Además, obtuvieron datos experimentales de las operaciones *scatter* y *gather*. Como se mencionó anteriormente, realizaron pruebas con varios modelos de comunicación, uno de estos modelos es el LMO propuesto por los mismos autores. En el modelo LMO provee dos diferentes fórmulas para el *scatter* y el *gather*, donde se ve reflejado una mayor pendiente en el tiempo de ejecución del *gather* para mensajes grandes. Por lo tanto, establecen que el modelo LMO refleja el comportamiento irregular del *gather*. Además, en un *cluster* de computadoras con el protocolo de comunicación TCP/IP, observaron un salto en el tiempo de ejecución del *scatter* en los 64 KB. En previas versiones del modelo LMO, incluyeron parámetros extras para reflejar el salto. Sin embargo, para mensajes grandes los saltos se repiten con regularidad, los cuales convergen a una forma más estable y con la misma pendiente. Por lo que incluyeron múltiples parámetros empíricos al modelo LMO y han presentado el tiempo de ejecución del *scatter* como una función lineal por tramos, pero debido a los valores significativos y por simplicidad, consideran solo el modelo lineal, donde obtienen aproximaciones satisfactorias del tiempo de ejecución observado de la operación nativa *scatter* de LAM. La predicción del modelo [19] para mensajes de tamaño medio es muy cercana al experimental y también refleja el salto en el tiempo de ejecución, después diverge de las observaciones. La estimación de otros modelos tradicionales son poco precisas.

Por lo tanto, en este trabajo describen una técnica más eficiente para una estimación más precisa de los parámetros de algún modelo, donde incluye un número pequeño de comunicaciones punto a punto y colectivas, además, una solución del sistema de ecuaciones lineales. La precisión de la estimación fue lograda por una selección cuidadosa del tamaño del mensaje y los valores promedios de los parámetros. La precisión del modelo intuitivo fue validado experimentalmente con las operaciones de comunicación *scatter* y *gather*.

3.3.9. Predicción del tiempo de ejecución para tareas de procesamiento de datos paralelos

En el trabajo presentado por [18] se enfocan en crear y hacer uso de modelos matemáticos en un dominio, es decir, aplicaciones que hacen procesamiento moderado en una gran cantidad de datos. Además, las posibilidades para predecir y minimizar el tiempo de ejecución fueron analizadas en un *cluster* de computadoras, donde la transferencia de datos genera el cuello de botella. Por lo tanto, los modelos genéricos son empleados en un algoritmo paralelo de ordenamiento, donde proponen ecuaciones para proveer el tiempo de ejecución esperado y para estimar el tamaño del *cluster* óptimo.

Además, en esta investigación establecen con base en su experiencia, que cuando tratan con un volumen grande de datos, la comunicación y el rendimiento del almacenamiento local pueden ser modelados con el valor del ancho de banda, porque la transmisión puede ser ignorada comparada con el tiempo que lleva un estado estable (procesamiento). Por lo tanto, para tener una buena aproximación en el tiempo de ejecución, propusieron una metodología con los siguientes pasos:

- **Paso 1:** La entrada de datos es distribuida en P nodos arbitrarios.
- **Paso 2:** En un nodo la entrada es continuamente leída a través del canal de almacenamiento E/S, dividiéndola en n bloques de tamaño s, donde n es múltiplo de P.
- **Paso 3:** Cada nodo recibirá n/P bloques y los procesará en el orden que van llegando.
- **Paso 4:** Después de haber recibido y procesado todos los bloques n/P, resolverá las dependencias locales inter-bloque y determinará un resultado parcial de tamaño n/P.
- **Paso 5:** Como un nodo recibió la primera parte de todos los resultados parciales de los P nodos, el nodo resolverá las dependencias globales y reunirá la siguiente parte de los otros nodos.

Con esta metodología, obtienen una buena aproximación del tiempo de ejecución para problemas donde el procesamiento y las dependencias pueden ser estimadas con fórmulas matemáticas. Además, mencionan que requieren de pocos parámetros para ser procesados con métodos estadísticos. Aplicando los métodos estadísticos pueden obtener resultados que permiten soportar una decisión para asignar recursos o para problemas de calendarización de tareas.

Por lo tanto, observaron que los modelos matemáticos automáticamente dan la posibilidad de obtener el tamaño del *cluster* desde el punto de vista de la velocidad de ejecución, así como mover una gran cantidad de datos que generan mucho *overhead*, para la clase de problemas que plantearon, donde el rendimiento ganado disminuye con cada nuevo nodo agregado, además es claramente visible en los primeros nodos. Aunque en el modelo propuesto no lo incluye explícitamente, es importante que la cantidad de datos no excedan la memoria disponible por el nodo o por el *cluster*. En este trabajo suponen que los datos caben en memoria del *cluster*, de lo contrario se usará la memoria virtual del disco que puede causar una degradación en el rendimiento, donde el modelo que proponen toma en consideración estos factores.

3.4. Resumen

Estimar el tiempo de ejecución de una aplicación paralela en un *cluster*, ayuda a diseñar estrategias de paralelización para minimizar el tiempo de ejecución, que involucra tanto el tiempo de cómputo como de comunicación, debido a que entre más procesadores involucrados en la solución de un problema, mejor será el tiempo de procesamiento, pero incrementará el tiempo de comunicación entre los procesadores. Encontrar un balance entre los dos tiempos no es una tarea trivial, por lo que se han propuesto varios trabajos que estiman los tiempos, como: en el tiempo de cómputo Amdahl define los límites de aceleración que una aplicación puede alcanzar y en el tiempo de comunicación, se proponen varios modelos, donde algunos estiman el tiempo con un pequeño conjunto de parámetros y otros proponen parámetros más detallados, para las operaciones de comunicación punto a punto y colectivas.

Los modelos que proponen en trabajos previos no toman en consideración varios factores que pueden influir en el tiempo de ejecución total de una aplicación paralela, como: la distribución de los datos, el tamaño del bloque de datos, la latencia del *switch*, el tamaño de la memoria RAM y el tamaño del problema. Por lo tanto, se proponen varios modelos que toman en consideración estos factores para realizar una estimación cercana al tiempo medido de ejecución de una aplicación paralela.

Capítulo 4

Descripción de los modelos propuestos de estimación del tiempo de procesamiento en *clusters* de computadoras

En este capítulo se describe la metodología que se llevo acabo para estimar el tiempo de ejecución de una aplicación paralela, donde se detallan los modelos propuestos para estimar el tiempo de comunicación y de cómputo. Además, se describe el desarrollo y el uso de la aplicación que se propone para obtener los parámetros que minimicen el tiempo de ejecución de una aplicación paralela.

4.1. Introducción

Estimar el tiempo de ejecución de una aplicación paralela, así como los parámetros que optimicen dicho tiempo de ejecución, no es una tarea trivial, debido a que influyen varios factores, tales como el tiempo de cómputo, comunicación y *overhead* (sobrecarga). Por lo tanto, en la investigación se propone un modelo para estimar el tiempo de ejecución de una aplicación, que divide en tres tipos de tiempos que son: cómputo, comunicación y *overhead*. Se proponen modelos para la estimación de los tiempos de cómputo y comunicación. El tiempo de cómputo se refiere al periodo en el cual el procesador o núcleo está procesando un conjunto de datos. El tiempo de comunicación es el tiempo que se tarda en transmitir datos desde un nodo fuente a un nodo o varios nodos destinos. El tiempo de *overhead* abarca los retardos causados por la contención de la red, la pérdida de paquetes de datos y retransmisión de los paquetes de datos. La estimación del *overhead* está fuera del alcance de la investigación. A continuación se

da una descripción de los modelos que ayudaron a estimar el tiempo de ejecución de una aplicación paralela:

- El modelo del tiempo de cómputo de una aplicación paralela se basa en la ley de Amdahl [2], que define los límites de la aceleración que puede ser alcanzada teóricamente por una aplicación dada, por lo tanto, con base en la propuesta de Amdahl, se consideró el número de operaciones realizadas en un nodo y la velocidad de procesamiento del procesador o núcleo.

- El modelo del tiempo de comunicación que se propone abarca todos los factores que influyen en la transmisión de los datos, tales como: número de procesadores, tamaño de la memoria RAM, ancho de banda, latencia de la red, latencia del *switch* y los patrones de comunicación que emplea MPI, para obtener una estimación de tiempo de comunicación.

Además, en esta investigación se desarrolló una aplicación que hace uso de los modelos propuestos para determinar la mejor combinación de un conjunto de posibles combinaciones de los parámetros, tal como: número de procesadores, tamaño de la memoria RAM, ancho de banda de la red, latencia de la red, latencia del *switch*, distribución de lo datos y tamaño del bloque de datos, que mejore el tiempo de ejecución de una aplicación paralela. La aplicación acepta algoritmos paralelos para extraer los datos necesarios del *cluster* y de la aplicación. La extracción de los datos del algoritmo se realiza con la herramienta *octave* [8], para ello, el algoritmo debe ser codificado con la sintaxis de *octave* para analizar un conjunto de parámetros y estimar el tiempo de ejecución con los modelos propuestos.

En las siguientes secciones se detallan los modelos propuestos para estimar el tiempo de ejecución de una aplicación paralela. Además, se describe la aplicación que se propone para obtener los parámetros que minimicen el tiempo de ejecución, así como la herramienta usada para extraer los datos del algoritmo.

4.2. Condiciones necesarias para estimar el tiempo de ejecución

Los modelos propuestos que estiman el tiempo de ejecución de una aplicación paralela, se deben apegar a las siguientes condiciones para obtener un tiempo estimado cercano al tiempo medido. Estas condiciones son:

- La carga de trabajo se distribuye equitativamente, es decir a todos los nodos se les asigna la misma cantidad de datos a procesar.

- El *cluster* es homogéneo, es decir todos los nodos tienen las mismas características por ejemplo: velocidad de los procesadores, tarjeta de red, memoria RAM, entre otras.

- Sin pérdida de paquetes, es decir todos los paquetes de TCP/IP se entregarán al nodos destino.

- Sin retransmisión de paquetes, se refiere que un paquete no será retransmitido desde el nodo origen al nodo destino, es decir no se perderán paquetes ni contendrán errores.

- Ejecución de una aplicación a la vez en el *cluster*, es decir no deberán ejecutarse dos aplicaciones al mismo tiempo en el *cluster*.

- Los modelos propuestos sólo funcionan para redes con la topología de estrella, debido a que otras topologías involucrarían otras latencias, las cuales no están consideradas en los modelos propuestos.

- Solo se ejecuta un proceso en cada nodo y se utiliza el modelo SPMD.

4.3. Modelo del tiempo de ejecución

La ejecución de un algoritmo paralelo en entornos de *clusters* de computadoras puede ser definido como una secuencia de cómputo local intercalada con la comunicación de la red, donde los diferentes tipos de operaciones pueden traslaparse. Por lo que, para estimar el tiempo de ejecución de una aplicación paralela, se propone un modelo que abarca dichas operaciones. Los tiempos totales empleados en comunicación y *overhead* comparados con el tiempo empleado en cómputo, son indicativos de la eficiencia de una aplicación paralela. Por lo tanto, si se cuenta con un *cluster* formado por un conjunto de procesadores $P = \{P_1, P_2, P_3, ..., P_n\}$ que ejecutan procesos en paralelo. El tiempo total de ejecución de un procesador (P_i) puede expresarse como:

$$T_p^{Exec} = T_p^{Comp} + T_p^{Comm} + T_p^{Over} + T_p^{E/S} \tag{4.1}$$

Donde, los parámetros son (Tabla 4.1).

El tiempo de cómputo está inherentemente relacionado con el tamaño del problema a resolver. Un buen diseño puede hacer más eficiente el procesamiento de los datos.

Tabla 4.1: Parámetros del modelo de tiempo de ejecución

Parámetro	Descripción
T_p^{Exec}	Tiempo de ejecución paralela
T_p^{Comp}	Tiempo de cómputo paralelo
T_p^{Comm}	Tiempo de comunicación
T_p^{Over}	Tiempo de *overhead*
$T_p^{E/S}$	Tiempo de entrada y salida a disco

4.4. Modelo del tiempo de cómputo

La relación entre el tamaño del problema y el tiempo de cómputo que puede ser determinada en el número de operaciones a realizar. Si consideramos que la carga de trabajo se divide equitativamente entre los procesadores o núcleos y suponiendo que cada nodo tiene la misma configuración (*cluster* homogéneo), tenemos que el tiempo de cómputo por procesador es dado por la ecuación:

$$T_p^{Comp} \cong T_{p_i}^{Comp} = T_s/P \tag{4.2}$$

Donde $T_{p_i}^{Comp}$ es el tiempo de cómputo del p_i-iésimo procesador y es dado por el tiempo de ejecución secuencial T_s del programa ejecutando en un sólo procesador y, P es el número de procesadores del *cluster*. Esto es, T_s es igual al número de operaciones (sumas, multiplicaciones, etc.) de la aplicación por el tiempo que se lleva ejecutando cada una de ellas, es decir:

$$T_{p_i}^{Comp} = (N_p^{Oper} * T_{bas})/P \tag{4.3}$$

La descripción de los parámetros que usan los modelos se muestran en la Tabla 4.2.

Tabla 4.2: Parámetros del modelo de tiempo de cómputo

Parámetro	Descripción
T_s	Tiempo de ejecución secuencial
P	Número de procesadores
N_p^{Oper}	Total de operaciones básicas
T_{bas}	Tiempo de una operación básica

El tiempo de una operación básica T_{bas} que realiza el procesador es un valor fijo que se obtuvo experimentalmente al tomar el tiempo de procesamiento de una o varias operaciones de suma o multiplicación, entre otras o de un conjunto de las anteriores, donde se promedió las múltiples ejecuciones para obtener el tiempo mínimo. El T_{bas}

incluye implícitamente la velocidad de procesamiento del procesador, la transferencia de los datos local que representa el costo inducido por el movimiento de datos entre las diferentes jerarquías de la memoria, tales como, de cache a cache, de cache a memoria y de memoria a memoria.

4.5. Modelo de comunicación

La comunicación presente en programas paralelos no es tan aleatoria como la que se presenta en aplicaciones distribuidas. Normalmente, un programa paralelo tiene patrones de comunicación bien definidos. Estos van desde operaciones de comunicación punto a punto, entre uno o varios pares de procesadores hasta operaciones de comunicación colectiva las cuales involucran a más de un par de procesadores. En comunicaciones colectivas uno de los procesadores es el origen y los procesadores restantes son participantes. Destacan entre ellas, las operaciones tipo *broadcast*, *scatter* y *gather*. MPI es prolífico en variantes sobre operaciones colectivas. En una operación *broadcast* se envía un mensaje de un procesador a todos los participantes. Una operación *scatter* distribuye datos diferentes a cada uno de los participantes y la operación *gather* hace el trabajo opuesto, cada participante envía un mensaje diferente al procesador origen.

Para estimar el tiempo de comunicación de los diferentes patrones de transmisión se propone el siguiente modelo:

$$T_p^{Comm} = T_p^{Sync} + T_{comm_{p2p}} + T_{comm_{col}} + T_{O_{sw}} \tag{4.4}$$

En la ecuación 4.4, se estima el tiempo total de las operaciones de sincronización del procesador o núcleo T_p^{Sync} con los procesadores participantes en la transmisión de los datos. El tiempo de transmisión de las operaciones punto a punto $T_{comm_{p2p}}$ y las operaciones colectivas $T_{comm_{col}}$ son el tiempo total de transmisión de un procesador a los procesadores participantes sin retardos como, sincronización y *overhead* del *switch*. El tiempo total de *overhead* en el *switch* $T_{O_{sw}}$ es el tiempo que tarda en procesar el *switch* los paquetes de TCP/IP.

El tiempo de sincronización T_p^{Sync} involucra el factor de la latencia L. La latencia es un parámetro que representa el tiempo usado por la red en la entrega del dato desde la memoria del nodo emisor a la memoria del nodo receptor, como se observa en el Anexo C.1. Por ejemplo, desde la cola de transmisión del nodo emisor hasta la cola de recepción del nodo receptor. Este parámetro depende de la red y encapsula las velocidades de transmisión de diferentes componentes de la red, la topología de la red y el protocolo usado. El parámetro se obtuvo experimentalmente al realizar la comunicación entre un par de procesadores mediante la operación punto a punto, donde el nodo emisor transmitió cero bytes al nodo receptor y el nodo receptor transmitió cero

bytes al nodo emisor. Se tomó el tiempo de 32 transmisiones, donde se eliminó la mayor y la menor y se obtuvo el promedio de las restantes.

El tiempo de transmisión de las operaciones punto a punto y colectivas, consideran la distribución de los datos y el tamaño del bloque de datos a transmitir. La distribución de los datos influye al transmitir un bloque de datos a cada nodo destino. El tamaño del bloque establece el número de elementos que se transmiten a un nodo destino. Por lo tanto, los dos factores antes mencionados influyen en el aumento o disminución significativo de los datos totales a transmitir.

Los datos a transmitir se ven afectados por el protocolo de comunicación TCP/IP [34], el cual divide los datos a transmitir en pequeños paquetes de datos dentro de un rango como lo establece la unidad máxima de transferencia (MTU por sus siglas en inglés), en cada paquete el protocolo TCP/IP agrega varias cabeceras, tales son: aplicación H_{MPI}, protocolo de control de trasmisión (TCP por sus siglas en ingles) [28] H_{TCP}, protocolo de internet (IP por sus siglas en inglés) [27] H_{IP} y *Ethernet* H_{Eth}. Por lo tanto, la información que contienen las cabeceras es la dirección del nodo destino, el puerto del nodo destino, la longitud del paquete, entre otras. El tamaño de la cabera de MPI se obtuvo experimentalmente y las cabeceras de TCP/IP de la literatura [34]. Los tamaños de las cabeceras se muestran en la Tabla 4.3.

Tabla 4.3: Tamaño de las cabeceras

Cabecera	Tamaño (Bytes)
Aplicación (MPI)	22
TCP	32
IP	20
Ethernet	14

Para obtener el tamaño de las cabeceras en la red física, así como el patrón de distribución de las diferentes operaciones de comunicación en MPI, se empleó la herramienta Wireshark [30] para capturar el tráfico que circula en la red. Con la herramienta Wireshark se pudo observar el comportamiento de las operaciones de comunicación punto a punto y colectivas variando el tamaño de los datos a transmitir. Con la operación punto a punto se observaron los tamaños de las cabeceras que se transmitieron en el canal de comunicación. Y en las operaciones colectivas se observaron los distintos patrones de comunicación que realiza MPI, donde se identificaron los algoritmos que ejecuta MPI para llevar acabo la transmisión de los datos, los cuales se detallan en las siguientes secciones.

Por último el *overhead* del *switch* O_{sw} es el tiempo que tarda en procesar los paquetes de TCP/IP (latencia del *switch* L_{sw}), debido al dispositivo que usa la técnica *store-and-forward* para conmutar los paquetes, en esta técnica se verifica que el paquete no contenga errores de lo contrario será descartado. La latencia del *switch* es un parámetro que se obtuvo experimentalmente al obtener el tiempo de dos experimentos al transmitir

los datos con la operación punto a punto, en el primero se transmitió cero bytes entre un par de procesadores usando un *switch* de por medio, en el segundo la única diferencia respecto al primero es que no se uso un *switch* de por medio, se hizo la conexión directa entre los dos nodos, como se observa en el Anexo C.2. Y con base en los dos tiempos que se obtuvieron se obtuvo la diferencia entre los dos tiempos dando como resultado la latencia del *switch*. Para que la latencia del *switch* sea más realista se realizaron 32 transmisiones de las cuales se eliminaron la de mayor y la de menor tiempo y se obtuvo un promedio de las restantes.

4.6. Modelo para operaciones punto a punto

El tiempo de la comunicación punto a punto es dependiente de varios factores, como son el ancho de banda del canal, el tamaño del mensaje, la velocidad del procesador, la latencia de la red, la latencia del *switch*, la distribución de los datos, el tamaño del bloque de datos y otros factores que no son trascendentes en esta investigación. Para modelar el comportamiento de las operaciones punto a punto para un par de procesadores, se propone el siguiente modelo.

$$T_{comm_{p2p}} = \underbrace{L}_{T_p^{Sync}} + \underbrace{T_{trans}}_{T_p^{Trans}} + \underbrace{(N_{paq} * L_{sw})}_{T_{O_{sw}}} \tag{4.5}$$

Donde:

- L es el tiempo que le toma al procesador emisor sincronizarse con el procesador receptor.
- T_{trans} se refiere al tiempo que se tarda el procesador emisor en transmitir los datos al procesador receptor.
- En el *overhead* del *switch* es el número de paquetes N_{paq} de TCP/IP que pasan por el *switch* por el tiempo de procesamiento promedio del *switch* L_{sw}.

Para determinar la carga (datos) que le corresponden a cada procesador, está constituida por:

$$S_{p_i} = S_T/P \tag{4.6}$$

Donde:

- S_{p_i} se refiere a la porción de los datos para el p_i-iésimo procesador.
- S_T es el tamaño total de datos de la aplicación paralela.

Para determinar el número de bloques que se transmitirán, está constituido por:

$$N_B = N_{trans} = S_{p_i}/B \tag{4.7}$$

Donde:

- N_B es el número de bloques a transmitir.
- N_{trans} es el número de transmisiones que realiza el algoritmo paralelo.
- B es el tamaño del bloque (unidad mínima de transmisión del algoritmo paralelo).

Al transmitir un bloque de datos, el protocolo fragmenta los datos en paquetes y para estimar el número de paquetes que creará el protocolo TCP/IP se determina por:

$$N_{paq} = \lceil B/S_{TCP/IP}^{Datos} \rceil \tag{4.8}$$

Donde:

- N_{paq} es el número de paquetes que el protocolo TCP/IP debe crear para transmitir los datos.
- $S_{TCP/IP}^{Datos}$ es el tamaño del paquete de TCP/IP sin cabeceras.

Para determinar los datos que se transmiten desde el nodo origen al nodo destino, incluyendo las cabeceras del protocolo TCP/IP, está constituido por:

$$D_{p_i} = S_{p_i} + (N_{paq} * N_B) * (H_{MPI} + H_{TCP} + H_{IP} + H_{ETH}) \tag{4.9}$$

Donde:

D_{p_i} son los datos totales que se transmiten por la red. La transmisión de los datos se lleva acabo por medio del protocolo TCP/IP, que crea paquetes. A cada paquete el protocolo TCP/IP agrega información que será usada para que el paquete llegue a su destino. Esta información es agregada en cada paso del protocolo, cada paso agrega una cabecera con información, el protocolo cuenta con las cabeceras: H_{MPI}, H_{TCP}, H_{IP} y H_{ETH}, las cuales incrementan significativamente los datos a transmitir. Los factores que destacan son las variable $S_{p_i}^{Datos}$, B y N_B que dependen del problema, es decir cómo se distribuyan los datos y cuál es el tamaño del bloque a transmitir.

Para estimar el tiempo de transmisión T_{trans} está constituido por:

$$T_{trans} = D_{p_i}^{Datos}/\beta \tag{4.10}$$

Donde:

- T_{trans} se refiere al tiempo que tarda en transmitir los datos desde el nodo origen a los nodos destinos.
- β es el ancho de banda de la red, es decir la cantidad de información o de datos que se puede enviar a través de una conexión de red en un período dado.

Por otro lado, para estimar el tiempo de comunicación para varias transmisiones punto a punto, el modelo que se propone es el siguiente:

$$T_{comm_{p2p}} = \underbrace{L * P_{inter}}_{T_p^{Sync}} + \underbrace{T_{trans} * N_{trans}}_{T_p^{Trans}} + \underbrace{(N_{paq} * N_{trans}) * L_{sw}}_{T_{O_{sw}}} \tag{4.11}$$

Se puede observar que en el tiempo de sincronización T_p^{Sync}, influye el número de procesadores que intervienen P_{inter} en la transmisión de los datos. En el tiempo de transmisión T_p^{Trans} y en el *overhead* del *switch* $T_{O_{sw}}$, influye el número de transmisiones N_{trans} que se realicen.

Adicionalmente, es importante mencionar que se han considerado factores que son relevantes e igualmente importantes en la investigación y que normalmente no son considerados en los análisis de rendimiento, tales como la distribución de los datos, el tamaño del bloque de datos y la latencia del *switch*, en la distribución de los datos y el tamaño del bloque se refiere al número de envíos o recepciones de un bloque de datos. El número de envíos y recepciones varía respecto a la distribución que se elija, el tamaño del bloque varía para transmitir pequeños bloques, esto evita que los nodos restantes esperen mucho tiempo para recibir los datos asignados, lo cual implica más comunicación entre los nodos participantes o transmitir grandes bloques que aprovechan el ancho de banda de la red, lo que puede ocasionar saturación de la red [5]. Por último, la latencia del *switch*, como se mencionó anteriormente es el tiempo que le lleva al *switch* en procesar los paquetes de TCP/IP. Estos factores pueden interferir en gran medida en la aplicación y obtenerse respuestas no adecuadas; o en su caso establecer una relación entre ellos, donde se encuentre el tamaño adecuado del grano de distribución de la aplicación para que se mejoren los tiempos de ejecución totales.

4.7. Modelos para operaciones colectivas

Además de establecer comunicaciones simultáneas entre varios pares de procesadores, las aplicaciones paralelas presentan patrones de comunicación que involucran a grupos de procesadores. Los patrones más comunes se deben a operaciones de comunicación colectiva entre las cuales las más representativas son: *broadcast*, *scatter* y *gather*.

En la literatura mencionan que MPI puede usar distintos algoritmos para llevar acabo la distribución de los datos en una forma más eficiente [33], pero no se menciona que

algoritmos ejecuta MPI para realizar dicha tarea. Por lo que se indagó en las operaciones colectivas para averiguar los patrones de comunicación que realiza MPI. Para averiguar dichos patrones se ejecutaron las operaciones *broadcast*, *scatter* y *gather* y con la ayuda de la herramienta Wireshark se capturaron los paquetes de TCP/IP, una vez que se observaron los patrones de comunicación de las operaciones colectivas, se analizaron dichos patrones para deducir los algoritmos que usa MPI para la distribución de los datos, ya que se identificaron los algoritmos se procedió analizar en qué rangos de datos y procesadores intervienen los algoritmos. A continuación se describen los algoritmos de cada operación colectiva que se obtuvieron en esta investigación.

4.7.1. Broadcast

En la operación *broadcast* se observaron tres tipos de algoritmos los cuales son: árbol binomial, árbol binomial con intercambio y *pipeline*. A continuación se describen los tres algoritmos.

El primer algoritmo, el **árbol binomial**, se observó al realizar transmisiones de datos en los umbrales desde 1 Byte hasta 2047 bytes y desde 2 hasta 8 procesadores. El árbol binomial ofrece un mejoramiento substancial en el rendimiento para grandes *clusters*. Debido a la forma de operar, donde en el primer paso el nodo raíz $P0$ transmite datos al nodo $P1$. En el segundo paso el nodo raíz P0 transmite al P2 y en forma paralela P1 transmite a P3 (Figura 4.1). Con los nodos intermedios que transmiten y reciben, esto reduce el número de pasos para transmitir de $P-1$ de la forma secuencial a $log_2 P$ en un árbol binomial.

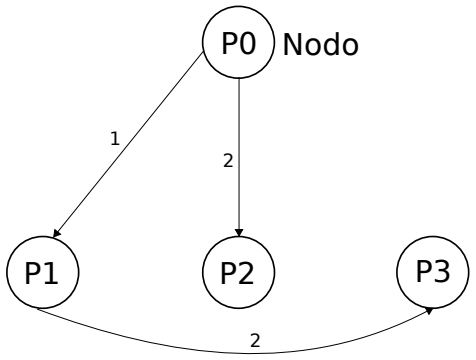

Figura 4.1: Transmisión con árbol binomial

El modelo que se propone para estimar el tiempo de comunicación es el siguiente:

$$T_{comm_{col}} = \underbrace{L * log_2 P}_{T_p^{Sync}} + \underbrace{T_{trans} * log_2 P}_{T_p^{Trans}} + \underbrace{(N_{paq} * N_{can}) * L_{sw}}_{T_{O_{sw}}} \tag{4.12}$$

En este modelo el tiempo de sincronización, de transmisión y el de *overhead* del *switch* se ven afectados por el algoritmo que invoca MPI para realizar la transmisión de los datos y el número de canales usados del *switch* (N_{can}), donde requiere sincronizarse con $log_2 P$ procesadores y distribuirá los datos en $log_2 P$ pasos a todos los procesadores, a través de los canales del *switch*.

Para determinar el número de canales usados en el *switch*, está dado por:

$$N_{can} = \lceil P/C_{sw} \rceil \tag{4.13}$$

Donde, C_{sw} es el número de canales que tiene el *switch* para llevar acabo la transmisión de los datos.

El segundo algoritmo, el **árbol binomial con intercambio**, se observó al transmitir datos en el umbral de 2048 bytes hasta 5 Megabytes y con los procesadores no se obtuvo debido a la compleja tarea de rastrear los paquetes que pasan por los nodos y observar hacia donde se dirigen dichos paquetes, esto se complicaba al incrementar el número de nodos. Este algoritmo es una mejora del anterior, el cual hace más eficiente la transmisión, esto es, al saturar la red. Una forma de lograr la saturación es al transmitir solo parte de los datos a nodos distintos, después, todos los nodos reciben alguna parte de los datos para dar lugar al intercambio entre los nodos por medio de la comunicación bidireccional. Esto es, el nodo raíz divide los datos en dos partes y trasmite la mitad a un nodo. A partir de aquí, los procesadores pares envían su mitad a los otros procesadores pares, lo mismo sucede con los procesadores impares, que transmiten su mitad a los otros procesadores impares, ambas transmisiones se realizan por el algoritmo de árbol binomial. Después, cada procesador tiene alguna parte de los datos, se realiza el intercambio de los datos con su vecino (Figura 4.2).

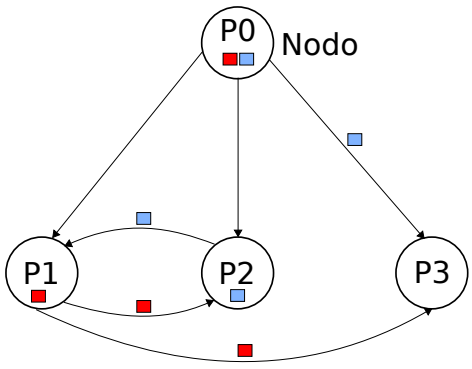

Figura 4.2: Transmisión árbol binomial con intercambio

El modelo propuesto para el algoritmo árbol binomial con intercambio es el mismo que usa en el árbol binomial que se expresa en la ecuación (4.12), debido a que sigue el mismo patrón de comunicación que el algoritmo de árbol binomial, pero con la

diferencia en el parámetro $S_{p_i}^{Datos}$, donde el nodo origen divide los datos a transmitir, es decir:

$$S_{p_i}^{Datos} = S_T/2 \tag{4.14}$$

Para transmitir datos pequeños, la latencia es a menudo un factor dominante, es decir, los algoritmos con pocos pasos (y pocos mensajes) son apropiados. Para datos muy grandes, los algoritmos antes descritos tienen bajo desempeño porque los datos son retransmitidos $\lceil log_2 P \rceil$ veces. Sin embargo, otro algoritmo que nos permite reducir el tiempo de transmisión es el *pipeline*.

El algoritmo ***pipeline*** se refiere cuando los nodos P son organizados como un arreglo lineal, un *broadcast* desde P_0 puede llevarse acabo al dividir los datos en segmentos de partes iguales. Específicamente, el nodo origen envía el primer segmento al segundo nodo en el arreglo. Al mismo tiempo el nodo origen envía el siguiente segmento, el segundo nodo reenvía el primer segmento al siguiente nodo, traslapando el tiempo de transmisión de los dos segmentos. El *broadcast* continua de esta manera, donde cada nodo recibe el siguiente segmento y reenvía los segmentos previos, hasta que todos los segmentos lleguen al último nodo [4]. El algoritmo *pipeline* se ilustra en la siguiente figura 4.3.

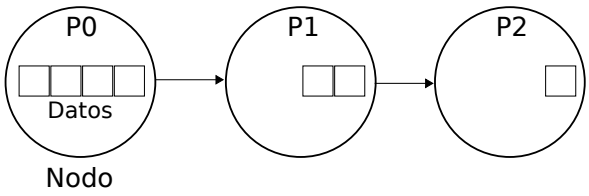

Figura 4.3: Transmisión con *pipeline*

El modelo que se propone para estimar el tiempo de comunicación para el algoritmo *pipeline* es el siguiente:

$$T_{comm_{col}} = \underbrace{L}_{T_p^{Sync}} + \underbrace{T_{trans} * N_{trans}}_{T_p^{Trans}} + \underbrace{(N_{paq} * N_{can}) * L_{sw}}_{T_{O_{sw}}} \tag{4.15}$$

Donde la latencia L se debe a la sincronización del nodo origen con un nodo destino. El tiempo de transmisión T_p^{Trans} está constituido por el tiempo que tarda al realizar una transmisión T_{trans} y por el número de pasos $(P-1)$ que realiza para que el último nodo tenga los datos que se transmiten, debido a la naturaleza del algoritmo, es decir, si el número de pasos para transmitir los datos al último nodo es $(P-1)$, al antepenúltimo nodo le llegará la información en $(P-2)$ pasos y así sucesivamente. Lo mismo sucede con el *overhead* del *switch* $T_{O_{sw}}$.

4.7.2. Scatter

En la operación *scatter* se distribuyen datos diferentes a cada uno de los nodos involucrados en la transmisión, como se mencionó al inicio de la sección. Los datos a transmitir a cada nodo está dado por:

$$S_{p_i}^{Datos} = S_T/P \tag{4.16}$$

Por lo tanto, al transmitir $S_{p_i}^{Datos}$ se observaron tres algoritmos que puede ejecutar MPI para llevar acabo la operación *scatter*, los cuales son: el árbol secuencial, el árbol binomial y el *pipeline*. A continuación se describe cada uno de los algoritmos mencionados.

En el primer algoritmo que se observó fue el **árbol secuencial** (Figura 4.4), este algoritmo es invocado en los siguientes rangos:

- Si los datos a transmitir $S_{p_i}^{Datos}$ es mayor o igual a 300 bytes y menor a 1 Megabyte.
- O si el número de procesadores P es menor o igual a 10.

Los rangos se obtuvieron experimentalmente al observar cuando ocurría el cambio de algoritmo en MPI, al transmitir diferentes tamaños de datos y al cambiar el número de procesadores. Y para modelar el tiempo de comunicación de este algoritmo se propone el siguiente modelo:

$$T_{comm_{col}} = \underbrace{L * N_{trans}}_{T_p^{Sync}} + \underbrace{T_{trans} * N_{trans}}_{T_p^{Trans}} + \underbrace{(N_{paq} * N_{can}) * L_{sw}}_{T_{O_{sw}}} \tag{4.17}$$

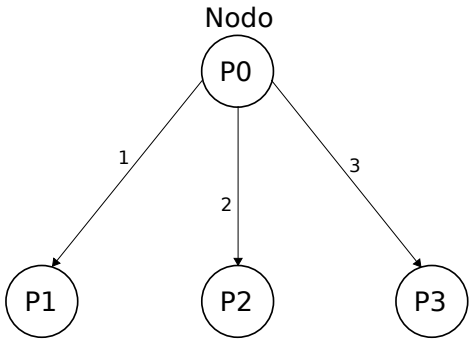

Figura 4.4: Transmisión secuencial

El siguiente algoritmo que se observó fue el **árbol binomial**, el cual se describió anteriormente. Este algoritmo es invocado por MPI en el siguiente rango, cuando la

transmisión de datos $S_{p_i}^{Datos}$ es menor a 300 bytes. Por lo que proponemos el siguiente modelo para estimar el tiempo de comunicación:

$$T_{comm_{col}} = \underbrace{L * log_2 P}_{T_p^{Sync}} + \underbrace{T_{trans} * log_2 P}_{T_p^{Trans}} + \underbrace{(N_{paq} * N_{can}) * L_{sw}}_{T_{O_{sw}}} \tag{4.18}$$

El último algoritmo que se observó en la operación *scatter* fue el ***pipeline***, el cual se describió anteriormente. El modelo que se propuso para este algoritmo es el mismo que en (4.15), pero la diferencia está en los datos a transmitir $S_{p_i}^{Datos}$.

4.7.3. Gather

La operación *gather* hace el trabajo opuesto respecto a la operación *scatter*, donde cada nodo participante envía un mensaje diferente al nodo origen, es decir, el total de datos que se recibirán es dado por $\sum_{i=1}^{P} S_{p_i}^{Datos}$. Donde se observaron los siguientes algoritmos:

El **árbol secuencial**, tiene el mismo comportamiento como en el algoritmo *scatter* pero el trabajo es opuesto, por lo tanto, se usó el mismo modelo (4.17). Este algoritmo es invocado por MPI, cuando el número de procesadores es menor a 11.

Por último, el algoritmo se observó para la operación *gather* fue el ***pipeline***, el cual es invocado por MPI cuando se transmite una gran cantidad de datos. El modelo que se propuso para este algoritmo es el mismo que en (4.15), pero la diferencia está en los datos a transmitir $S_{p_i}^{Datos}$.

4.8. Análisis y modelado de aplicaciones paralelas

El análisis y modelado del tiempo de ejecución de una aplicación paralela no es una tarea trivial debido a que existen varios factores que influyen en su ejecución, como se mencionó en secciones anteriores. Es decir, las operaciones de cómputo dependen del tamaño del problema a resolver, así como el número del procesadores involucrados y en las operaciones de comunicación influyen varios patrones de comunicación, más aun si están dentro de un ciclo se hace más complicado. Esto es, debido a que influye el tamaño de los datos a transmitir en cada interacción del ciclo, donde pueden intervenir un patrón de comunicación distinto.

Por lo tanto, debido a la complejidad que tiene el expresar el algoritmo de una aplicación paralela a un modelo matemático, se realizó una biblioteca de funciones de MPI para *octave* que facilita el análisis y modelado de algoritmos paralelos, donde es posible

expresar el algoritmo o el modelo matemático, facilitando la obtención del tiempo estimado de ejecución de una aplicación paralela sin importar cuál sea el algoritmo. La biblioteca fue posible gracias a la herramienta *octave* [8].

La herramienta *octave* es un lenguaje intérprete de alto nivel enfocado principalmente para cómputo numérico. Proporciona la capacidad de solucionar problemas numéricos lineales y no lineales, así como, realizar otro tipo de problemas numéricos. El lenguaje *octave* es muy similar a Matlab de manera que la mayoría de los programas son portables [8]. Por esta razón, se eligió esta herramienta por su fácil uso y la portabilidad en los programas.

Existen ciertos algoritmos que requieren un análisis más detallado para obtener los parámetros, como: la distribución de los datos, el tamaño del bloque de datos, que en sí dependen del algoritmo implementado. De esta manera, la biblioteca considera la distribución de los datos, así como el tamaño del bloque de los datos en el algoritmo paralelo.

En este trabajo de tesis, para estimar el tiempo total de ejecución de una aplicación paralela, el algoritmo puede ser expresado en tres formas distintas, que son:

1. Pseudocódigo en octave. El algoritmo es expresado de la misma forma que el código en MPI, es decir con los ciclos que este contiene en las operaciones de comunicación y cómputo. Las operaciones se estiman independientemente.

2. Pseudocódigo en octave reducido. El código del algoritmo puede ser reducido, es decir sin ciclos y sólo tomar en consideración el número de operaciones que efectúan los ciclos, ya que sólo se requiere estimar una operación de comunicación y cómputo para ser multiplicadas por el número de operaciones de comunicación y cómputo restantes.

3. Modelo matemático. El algoritmo es expresado mediante el modelo matemático que describe el número de operaciones realizadas, es decir, el número de envíos y recepciones que se realizan en el nodo origen para estimar el tiempo de comunicación y el número de operaciones de procesamiento para estimar el tiempo de cómputo.

Las operaciones de comunicación que se proponen son incluidas en una biblioteca de funciones de MPI para *octave*. Las funciones de MPI implementadas son: MPIO_Send, MPIO_Recv, MPIO_Bcast, MPIO_Scatter y MPIO_Gather (ver Tabla 4.4). Puede observarse que cada función de MPI implementada se nombra como MPIO para diferenciarla de MPI. Cada operación de la biblioteca implementa los modelos propuestos en las secciones anteriores, es decir en las operaciones MPIO_Send y MPIO_Recv se implementó el modelo de la sección 4.6 para operaciones punto a punto. En las operaciones colectivas, tales como: MPIO_Bcast, MPIO_Scatter y MPIO_Gather, se implementaron los modelos de las secciones 4.7.1, 4.7.2, 4.7.3, respectivamente. Por lo tanto, cada vez

que una función de la biblioteca es ejecutada, se realiza una estimación del tiempo de ejecución de la operación, la cual automáticamente se va sumando para obtener el tiempo total de la comunicación. Para observar el código implementado de estas operaciones de MPI se puede consultar el Anexo .

Tabla 4.4: Equivalencia de las funciones de MPI y de la biblioteca desarrollada

MPI	Biblioteca
MPI_Send	MPIO_Send
MPI_Recv	MPIO_Recv
MPI_Bcast	MPIO_Bcast
MPI_Barrier	MPIO_Barrier
MPI_Scatter	MPIO_Scatter
MPI_Gather	MPIO_Gather

Por ejemplo, en la Figura 2.12 de la sección 2.6.3.1 se tiene el código de MPI que transmite datos entre un par de procesadores, en donde se observa que a partir de la línea 1 a la 10 se tiene la inicialización del programa paralelo, desde la línea 11 a la 14 está la parte operacional del algoritmo, es decir en la línea 11 se tiene una condición para identificar el nodo que realiza la operación. Si es el nodo 0 (origen), entonces en la línea 12 se realiza la transmisión de los datos al nodo 1 (destino). Si no es el nodo origen, entonces en la línea 14 el nodo destino ejecuta la recepción de los datos y desde la línea 15 a la 17 se tiene la finalización del programa paralelo.

Si se desea migrar el código de MPI de la Figura 2.12 a *octave* usando la biblioteca desarrollada, se debe tomar en consideración que la parte de inicialización y de finalización del código en MPI no se requiere migrar a *octave*, debido a que este tiempo no es relevante para la biblioteca porque no influye de manera transcendental en el tiempo de ejecución del algoritmo, solo la parte operacional del algoritmo es considera. Sin embargo, en *octave* se estable un código de inicialización para su correcto funcionamiento, en este código como es mostrado en la Figura 4.5 se muestra lo que está establecido por omisión para cada cálculo del tiempo de ejecución de una aplicación paralela. Esto es, se observa que desde la línea 3 a la 35 son variables establecidas para el uso de la biblioteca, las cuales fueron obtenidas por el usuario y el proceso de caracterización. A partir de la línea 37 a la 39 es donde va el algoritmo de la aplicación paralela del usuario y éste es insertado cada vez que se requiere estimar su tiempo de ejecución. Finalmente en la línea 42 se obtiene la estimación el tiempo de ejecución.

Para migrar la parte operacional del algoritmo en MPI a *octave* usando la biblioteca desarrollada, se debe considerar que no se requiere hacer distinción de quién es el nodo origen o el nodo destino y sólo usar las funciones que contiene la biblioteca. Por lo

```
function [Texec, Tcomp, Tcomm]=fmaster(Datos,P,B,L,Lsw,Tbas)
  % Variables propias de la biblioteca
  global OCT_Datos; % Datos a transmitir
  global OCT_Procs; % Numero de procesadores
  global OCT_Bandw; % Ancho de banda
  global OCT_Latencia; % Latencia de la red
  global OCT_Lsw; % Latencia del switch
  global OCT_Tbas; % OCT_Tbas: Tiempo basico de la aplicacion
  global OCT_Ttrans; % Tiempo de transmision
  global OCT_Tcomm; % Tiempo de comunicacion

  % Tiempos
  global Texec; % Tiempo estimado de ejecucion
  global Tcomm; % Tiempo de comunicacion
  global Tcomp; % Tiempo de computo

  % Utileria
  global OCT_P_iters; % Numero de iteracciones

  % Variables de MPI
  global MPIO_CHAR;
  global MPIO_SHORT;
  global MPIO_INT;
  global MPIO_FLOAT;
  global MPIO_DOUBLE;
  global MPIO_LONG;

  % Inicializar valores de la biblioteca
  OCT_Init(Datos, P, B, L, Lsw, Tbas);

  Tcomm = 0;
  Tcomp = 0;
  Texec = 0;
  OCT_P_iters = 0;
  OCT_Tbas = Tbas;

  % ### Inicio del Algoritmo ###
  % AQUI: Algortimo del usuario
  % ### Fin del Algoritmo ###

  % Estimar el tiempo de ejecucion
  OCT_Estimacion();
endfunction
```

Figura 4.5: Código de la biblioteca desarrollada en *octave*

tanto, la parte operacional queda expresada con la biblioteca, como se ilustra en la Figura 4.6. Se puede observar que no se requiere de la parte receptora, debido a que la función MPIO_Send estima el tiempo de transmisión desde el nodo origen al nodo destino. El código generado tanto en MPI como en *octave* tienen una gran similitud entre las funciones de MPI y de la biblioteca, por lo que el usuario no tendrá dificultades en expresar su algoritmo.

```
1    MPIO_Send(buffer, cont, MPIO_INT, 1, tag, MPI_COMM_WORLD);
```

Figura 4.6: Código del usuario usando la biblioteca desarrollada

Ahora vamos analizar un ejemplo en MPI, donde se transmiten y reciben datos desde y hacia un nodo dentro de un ciclo *for*, como se ilustra en la Figura 4.7. Se observa que la sintaxis del ciclo *for* consta de tres partes, que son: inicialización, condición de parada e incremento. En este ejemplo de MPI usando el lenguaje C la inicialización de una variable puede ser inicializada a partir de cero. Sin embargo, la sintaxis del ciclo *for* expresada en *octave* difiere un poco a la expresada en MPI, donde el código de la Figura 4.7 puede expresarse en *octave* como se ilustra en la Figura 4.8.

```
1  #inclide <stdio.h>
2  #include <mpi.h>
3  #define root 0
4  #define cont 10
5  #define elems 100
6  int main(int argc, char *argv[])
7  {
8    int rank, i, tag = 1;
9    int buffer[cont];
10   MPI_Status status;
11   MPI_Init(&argc, &argv);
12   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13   for(i=0; i < cont; i++) {
14     if(rank == root)
15       MPI_Send(buffer, elems, MPI_INT, 1, tag, MPI_COMM_WORLD);
16     else
17       MPI_Recv(buffer, elems, MPI_INT, 0, tag, MPI_COMM_WORLD,
           &status);
18   }
19   MPI_Finalize();
20   return 0;
21 }
```

Figura 4.7: Código en MPI para la transmisión de los datos entre dos nodos dentro de un ciclo

```
MPIO_Init();
cont = 10;
elems = 100;
for i=1:cont
  MPIO_Send(buffer, elems, MPIO_INT, 1, tag, MPI_COMM_WORLD);
endfor
```

Figura 4.8: Código en *octave* para la transmisión de los datos entre dos nodos dentro de un ciclo

En la Figura 4.8 se puede observar la sintaxis del ciclo *for* en *octave* que consta de dos partes, que son: inicialización y condición de parada. En la parte de inicialización, una variable puede ser inicializada a partir de uno, esta variable se incrementa automáticamente dentro del ciclo hasta cumplir la condición de parada. Además, se puede observar que el código en *octave* es mucho más sencillo que el de MPI.

Ahora vamos a analizar la parte operacional de un algoritmo de la multiplicación de matrices, el cual consta de dos matrices cuadráticas A y B, y el resultado de la multiplicación es almacenado en una tercera matriz C. El código del algoritmo se ilustra en la Figura 4.9. En el código de la Figura 4.9 existen varios ciclos *for* que recorren las matrices para su procesamiento, en donde en la línea 9 del código se realiza el procesamiento de las matrices del algoritmo.

```
1  filas = N / P; //Se determina el numero de filas
2  offset = fncOffset(filas); // Se determina el offset
3  for(i=0; i < filas; i++) // Filas a procesar
4  {
5    r = i + offset;
6    for(j = 0; j < NF; j++) // Filas de A
7    {
8      for(k=0; k < NC; k ++) // Columnas de B
9        *((C+r)+j) += *((A+r)+j) * *((B+r)+k);
10   }
11 }
```

Figura 4.9: Código en el lenguaje C para la multiplicación de matrices

Para migrar el código de la Figura 4.9 a la biblioteca desarrollada, se debe identificar la operación básica del algoritmo dado. En este caso, en la línea 9 del código se tiene la operación básica para la multiplicación de matrices. Una vez que se identificó la operación básica, el siguiente paso es realizar su caracterización, es decir obtener el tiempo de procesamiento de esta operación básica. Al obtener el tiempo de procesamiento del algoritmo dado, el código puede ser expresado con la biblioteca desarrollada como se ilustra en la Figura 4.10. Donde el tiempo de ejecución de la operación básica va implícito en

la función OCT_Procesamiento. Así cada vez que la función OCT_Procesamiento es invocada se incrementa el tiempo de cómputo total con base en el tiempo de ejecución de la operación básica. Además, se puede tener más de una operación básica en el algoritmo y pueden ser agregadas fácilmente a la biblioteca. De esta manera, la biblioteca desarrollada facilita la programación del usuario y la hace más versátil y sencilla.

```
filas = N / P; % Se determina el numero de filas
for i=1:filas % Filas a procesar
  for j=1:NF % Filas de A
    for k=1:NC % Columnas de B
      OCT_Procesamiento();
    endfor
  endfor
endfor
```

Figura 4.10: Código en *octave* para la multiplicación de matrices

A continuación se presenta una comparación entre MPI y la biblioteca, donde se expresa un algoritmo de la multiplicación de matrices en un esquema paralelo usando comunicaciones punto a punto. Este algoritmo consiste en multiplicar dos matrices cuadráticas A y B, y el resultado es almacenado en una tercera matriz cuadrática C con un tamaño igual a N x N. En el primer paso, primero se distribuyen de una manera equitativa las filas de datos que le corresponde de la matriz A a los nodos de cómputo y después la matriz B es replicada en todos los nodos desde el nodo cero conocido como nodo maestro. Una vez que todos los datos de ambas matrices son distribuidos a los nodos de cómputo, se inicia el proceso de la multiplicación de la matriz A y B, y los resultados parciales de C son obtenidos en cada nodo. Por último los nodos de cómputo transmiten su porción de la matriz C al nodo origen.

En el código del algoritmo de la multiplicación de matrices expresado en MPI se puede observar lo siguiente: de la línea 8 a la 21 se realiza la transmisión de las matrices A y B desde el nodo maestro a los nodos de cómputo y en cada transmisión se envía una fila de las matrices, de la línea 25 a la 36 se realiza la recepción de los datos enviados por el nodo maestro. De la linea 40 a la 48 se realiza la operación de cómputo donde se lleva acabo la multiplicación de las matrices en cada nodo de cómputo. La recepción de los datos de la matriz C en el nodo maestro está desde la línea 62 a la 69, donde recibe los resultados de los nodos de cómputo. Por último de la línea 53 a la 57 se realiza el envío de los resultados desde los nodos de cómputo al nodo maestro.

```
telems = N * N; // Total de elementos
filas = N / nprocs; // Porcion de elementos

MPI_Barrier(MPI_COMM_WORLD);

```

```
6  if(rank == root) // Si es el nodo maestro
7  { // Transmision de los datos
8    for(i=1; i < nprocs; i++) // A cada procesador
9    { // Matriz A
10     for(j=0; j < filas; j++) // Elementos
11     {
12       MPI_Send(&A[elems], N, MPI_FLOAT, i, 1, MPI_COMM_WORLD);
13       elems += N;
14     }
15     elems = 0;
16     for(j=0; j < N; j++) // Matriz B
17     {
18       MPI_Send(&B[elems], N, MPI_FLOAT, i, 1, MPI_COMM_WORLD);
19       elems += N;
20     }
21   }
22 }
23 else // Nodos de computo
24 {
25   elems = 0;
26   for(i=0; i<filas; i++)//Recepcion de A en los nodos de
        computo
27   {
28     MPI_Recv(&A[elems], N, MPI_FLOAT, root, 1, MPI_COMM_WORLD,
          &status);
29     elems += N;
30   }
31   elems = 0;
32   for(j=0; j<n; j++)//Recepcion de B en los nodos de computo
33   {
34     MPI_Recv(&B[elems], N, MPI_FLOAT, root, 1, MPI_COMM_WORLD,
          &status);
35     elems += N;
36   }
37 }
38 // Procesamiento de los datos
39 elems = 0;
40 for(i=0; i < filas; i++)
41 {
42   for(j = elems, l=0; j < elems + N; j++, l++)
43   {
44     for(k=l, p = elems; k < total_elems; k += N, p++)
45     C[j] += A[p] * B[k];
46   }
47   elemsx += N;
48 }
```

```
49 //Recepcion de los resultados de C en el nodo maestro
50 if(rank != root)
51 {
52   elems = 0;
53   for(i=0; i < filas; i++)
54   {
55     MPI_Send(&C[elemsx],N, MPI_FLOAT, root, 1, MPI_COMM_WORLD);
56     elemsx += N;
57   }
58 }
59 else // Transmision de los resultados
60 {
61   elems = inicio;
62   for(i=1; i < nprocs; i++)
63   {
64     for(j=0; j < filas; j++)
65     {
66       MPI_Recv(&C[elems], N, MPI_FLOAT, i, 1, MPI_COMM_WORLD, &
             status);
67       elemsx += N;
68     }
69   }
70 }
```

Es importante mencionar que la parte de inicio, tanto de MPI como del algoritmo no se toman en cuenta en este caso. El código completo del algoritmo se puede observar en el Anexo B.3.

A continuación mostramos el código del algoritmo expresado usando la biblioteca desarrollada, el cuál es equivalente al de MPI. En éste código, a partir de la línea 2 a la 10 se estima el tiempo de transmisión de las matrices, donde se observa la estimación del tiempo de comunicación respecto a la distribución de los datos y el tamaño del bloque de datos a transmitir desde el nodo maestro a los nodos de cómputo. El tiempo de cómputo se estima de la línea 12 a la 18, donde se observa que se estima el tiempo con base en las operaciones que realiza cada nodo de cómputo mediante la función *OCT_Procesamiento*(). La recepción de los resultados producto de la multiplicación de las matrices se realiza de la línea 20 a la 23, donde se estima el tiempo de comunicación desde los nodos de cómputo a el nodo maestro.

```
1 MPIO_Barrier(MPI_COMM_WORLD);
2 for i=2:OCT_Procs
3   for j=1:elems_proc(i)
4     MPIO_Send(buf, OCT_Datos, MPI_FLOAT, i, 1, MPI_COMM_WORLD);
5   endfor
6
```

```
7     for j=1:OCT_Datos
8       MPIO_Send(buf, OCT_Datos, MPI_FLOAT, i, 1, MPI_COMM_WORLD);
9     endfor
10  endfor
11
12  for i=1:elems_proc(1) # Filas a procesar
13    for j=1:OCT_Datos # Columnas
14      for k=1:OCT_Datos # Elemento
15        OCT_Procesamiento();
16      endfor
17    endfor
18  endfor
19
20  for i=2:OCT_Procs
21    for j=1:elems_proc(i)
22      MPIO_Recv(C, OCT_Datos, MPI_FLOAT, i, 1, MPI_COMM_WORLD,
          status);
23    endfor
24  endfor
```

Como se puede observar tanto la sintaxis de MPI como de la biblioteca son muy similares, esto es con el fin de facilitar el uso de la biblioteca, donde el usuario solo tiene que aprender lo básico de la sintaxis de *octave*. Además, el usuario puede hacer cambios en su algoritmo sin ningún inconveniente, por lo tanto se puede expresar cualquier algoritmo paralelo para obtener el tiempo estimado de su ejecución.

A continuación se presenta el mismo algoritmo de la multiplicación de matrices en un esquema paralelo con comunicaciones colectivas. En este código del algoritmo no se presenta el proceso de inicialización ni finalización. El código completo del algoritmo tanto en MPI como en *octave* se puede observar en el Anexo B.1.

```
1  N = atoi(argv[1]); // Obtiene el valor dado por el usuario
2
3  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
5
6  telems = N * N; // Elementos totales
7  segmento = telems / nprocs; // Segmento de datos
8
9  elems = N / nprocs; // Filas a procesar
10
11 // Comunicacion
12 MPI_Barrier(MPI_COMM_WORLD);
13 MPI_Scatter(A, segmento, MPI_FLOAT, FA, segmento, MPI_FLOAT,
     root, MPI_COMM_WORLD);
14 MPI_Bcast(B, telems, MPI_FLOAT, root, MPI_COMM_WORLD);
```

```
15
16  // Computo
17  for(i=0; i < elems; i++)
18  {
19    for(j = seg, p=0; j < seg + N; j++, p++)
20      {
21        for(k = p, q = seg; k < telems; k += N, q++)
22          FC[j] += FA[q] * B[k];
23      }
24      seg += N;
25  }
26
27  // Comunicacion
28  MPI_Gather(FC, segmento, MPI_FLOAT, C, segmento, MPI_FLOAT,
       root, MPI_COMM_WORLD);
```

Algoritmo usando la biblioteca desarrollada.

```
1  MPIO_Barrier(MPI_COMM_WORLD);
2  MPIO_Scatter(send_A, segmento, MPI_FLOAT, recv_a, segmento,
       MPI_FLOAT, root, MPI_COMM_WORLD);
3  MPIO_Bcast(bufB, total_elems, MPI_FLOAT, root, MPI_COMM_WORLD);
4
5  for i=1:elems_proc(1) # Filas a procesar
6    for j=1:TP # Columnas
7      for k=1:TP # Elemento
8        OCT_Procesamiento();
9      endfor
10   endfor
11 endfor
12
13 MPIO_Gather(send_c, segmento, MPI_FLOAT, recv_C, segmento,
       MPI_FLOAT, root, MPI_COMM_WORLD);
```

Los algoritmos expresados tanto en MPI como en la biblioteca se pueden observar que tienen una gran similitud. En la biblioteca sólo es necesario expresar la operación colectiva e internamente se estima el tiempo de comunicación que ésta conlleva, así como todos los factores que involucra. Por lo tanto, la biblioteca es muy versátil donde se puede cambiar de algoritmo o partes del mismo sin grandes cambios, siempre que se respete la sintaxis de las operaciones de comunicación.

4.9. Resumen

Determinar qué parámetros mejoran el tiempo de ejecución no es una tarea trivial y estimar el tiempo de ejecución de una aplicación paralela es complejo, porque influyen varios factores, tales como el tiempo de cómputo, comunicación y *overhead*. Para estimar el tiempo de cómputo y comunicación propusimos modelos que consideran varios factores. En el modelo del tiempo de cómputo, se consideraron los factores, como: el número de operaciones realizadas en un nodo y la velocidad de procesamiento del procesador o núcleo. Los modelos del tiempo de comunicación que se proponen abarcan los factores que influyen en la transmisión de los datos, tales como: número de procesadores, tamaño de la memoria RAM, ancho de banda, latencia de la red, latencia del *switch*, distribución de los datos, tamaño del bloque de datos y los patrones de comunicación que emplea MPI, son usados para estimar el tiempo de comunicación de las operaciones punto a punto y colectivas.

Estos modelos fueron implementados en una biblioteca en *octave*. De tal manera que es:

- Fácil de usar.
- No se requiere tener conocimiento de análisis matemático ni tener un modelo matemático del algoritmo paralelo.
- No se requiere aprender un nuevo lenguaje.
- Fácil la migración de un algoritmo en MPI a *octave*.
- Permite obtener el tiempo de ejecución de una aplicación paralela al variar los diferentes parámetros de la aplicación paralela sin realizar grandes cambios al código de la aplicación.

Capítulo 5

Validación de la propuesta: Experimentos y resultados

En este capítulo se presentan los experimentos que se realizaron para obtener el tiempo de cómputo, de comunicación y de un caso de estudio. Se presentan los resultados que se obtuvieron al estimar el tiempo de cómputo, de comunicación y del caso de estudio con los modelos propuestos, el tiempo estimado es comparado con el tiempo medido.

5.1. Experimentos

La multiplicación de matrices en un esquema paralelo, requiere el uso continuo del procesador y la transferencia de los datos a través de la red entre los nodos. El problema de multiplicación de matrices es flexible para realizar cambios en el tamaño del problema, la distribución de los datos y el tamaño del bloque de los datos a transmitir. Por lo tanto, se eligió para experimentar el tiempo de comunicación al transmitir diferentes bloques de datos. También, para experimentar el tiempo de cómputo al cambiar el tamaño del problema.

En este trabajo, se obtuvieron dos tiempos de ejecución, que son:

- *Tiempo estimado* de ejecución, se refiere al tiempo de comunicación y de cómputo que se obtuvieron al aplicar los modelos que se propusieron en esta investigación.
- *Tiempo medido* de ejecución, se refiere al tiempo que efectivamente tarda en finalizar la ejecución del problema en un esquema paralelo, que involucra el tiempo de comunicación y de cómputo.

Los tiempos estimados y medidos son comparados para determinar que tan precisos

son los modelos propuestos, respecto al tiempo medido para dar una buena estimación del tiempo de ejecución.

El tiempo de ejecución se puede estimar cuando:

- Se conocen las características del problema y del *cluster*, es decir se conoce el tamaño del problema, el ancho de banda de la red, la latencia de la red, la latencia del *switch* y el número de procesadores disponibles. Sin embargo, existen algunos parámetros que permanecen sin variar (ver Tabla 5.1) y los que varían (ver Tabla 5.2).

- Se conoce sólo el problema, es decir sólo se conoce el tamaño del problema, y los parámetros que permanecen sin variar (ver Tabla 5.3) y los que varían (ver Tabla 5.4).

Tabla 5.1: Parámetros fijos, cuando se conoce el problema y el *cluster*

Tamaño de la memoria RAM
Ancho de banda de la red
Latencia de la red
Latencia del *switch*
Tamaño del problema

Tabla 5.2: Parámetros variables, cuando se conoce el problema y el *cluster*

Número de procesadores
Distribución de los datos
Tamaño del bloque de datos

Tabla 5.3: Parámetros fijos, cuando sólo se conoce el problema

Tamaño del problema

En esta investigación, se trabajó cuando el problema y el *cluster* se conocen. Donde para estimar el tiempo de ejecución se estimó el tiempo de comunicación y cómputo, con una combinación de parámetros. Una combinación es un conjunto de parámetros, donde algunos permanecen fijos y otros variables, por ejemplo (ver Tabla 5.5). Se trabajó variando el número de procesadores, el tamaño del problema, la distribución de los datos y el tamaño del bloque de datos.

Tabla 5.4: Parámetros variables, cuando sólo se conoce el problema

Número de procesadores
Tamaño de la memoria RAM
Ancho de banda de la red
Latencia de la red
Latencia del *switch*
Distribución de los datos
Tamaño del bloque de datos

Tabla 5.5: Ejemplo de una combinación de parámetros

Número de procesadores	2	3	...	30
Tamaño de la memoria RAM	4 GB	...	...	4 GB
Ancho de banda de la red	1 Gbps	...	...	1 Gbps
Latencia de la red	4 μseg	...	...	4 μseg
Latencia del *switch*	75 μseg	...	...	4 μseg
Tamaño del problema	5000 por 5000	...	...	5000 por 5000
Distribución de los datos	fila-columna	segmento-segmento	segmento-todo	fila-todo
Tamaño del bloque de datos	5000 elementos	10000 elementos	15000 elementos	20000 elementos

5.1.1. Estimación del tiempo de ejecución

Para realizar las posibles combinaciones, se desarrolló una aplicación que combina todos los parámetros que pueden variar. Los parámetros que recibe la aplicación para ser procesados, los obtiene de un conjunto de archivos. Los archivos son divididos en dos categorías respecto a la información que contienen, las cuales son: del *cluster* y del problema que se desea procesar. Los archivos que se refieren al *cluster*, contienen información de las características, tales como: el número de procesadores disponibles en el *cluster*, la memoria en RAM disponible en los nodos, el ancho de banda de la red, la latencia de la red y la latencia del *switch*. Las características del problema se obtienen respecto al algoritmo que se deseé ejecutar y son: el tamaño del problema, la distribución de los datos y el tamaño del bloque de los datos a transmitir.

Después de que la aplicación desarrollada realiza todas las posibles combinaciones, cada combinación es enviada desde la aplicación desarrollada a la herramienta *octave*, donde se estima el tiempo de ejecución de dicha combinación, la estimación se realiza con los modelos propuestos en esta investigación. Cuando son procesadas todas las posibles combinaciones en la herramienta *octave*, son enviadas de nueva cuenta a la aplicación desarrollada, donde son ordenas con base en el tiempo de ejecución estimado en forma

ascendente por medio del algoritmo *quicksort*, de esta forma es más fácil encontrar los parámetros que minimizan el tiempo de ejecución. En la Figura 5.1 se ilustra el proceso que lleva acabo la aplicación desarrollada.

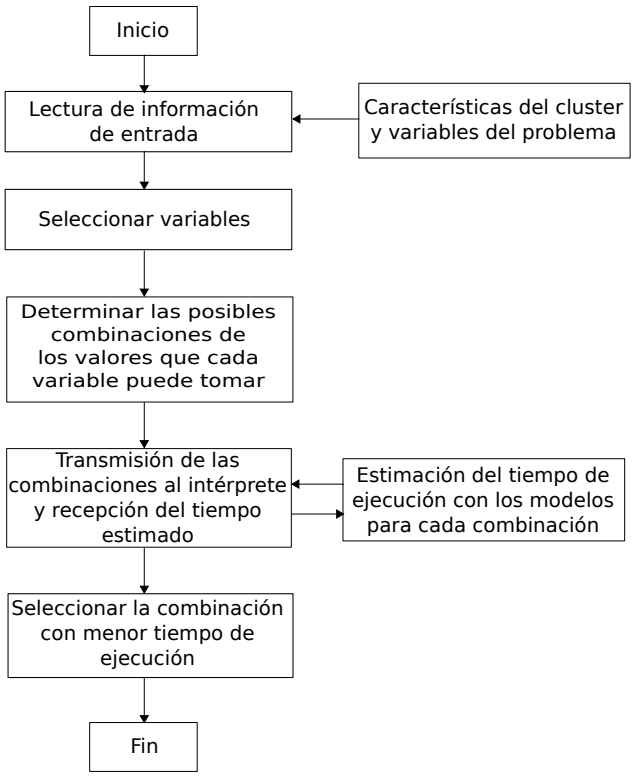

Figura 5.1: Diseño de la aplicación

Para obtener el tiempo medido de ejecución del algoritmo, se experimentó combinando la distribución, el tamaño del bloque de datos, el número de procesadores y el tamaño del problema, para observar el comportamiento del tiempo de comunicación y de cómputo.

5.1.2. Validación de los modelos propuestos

Para estimar el tiempo de ejecución con los modelos propuestos, es necesario expresar el número de operaciones de comunicación y de cómputo del algoritmo, es decir, se requiere el modelo de cada algoritmo dado. Los algoritmos que se emplearon usan comunicaciones punto a punto y colectivas, a continuación se expresan los modelos de cada algoritmo con su correspondiente distribución.

En el algoritmo que emplea comunicaciones punto a punto para transmitir una matriz usando la distribución fila-columna, es representada por la ecuación 5.1, que representa

el número de comunicaciones que involucra las transmisiones y recepciones de los datos.

$$N^{oper} = \underbrace{N_B * P_{inter}}_{M_A} + \underbrace{N * P_{inter}}_{M_B} + \underbrace{N_B * P_{inter}}_{M_C} \tag{5.1}$$

En la Tabla 5.6 pueden observarse las descripciones de los parámetros de la ecuación 5.1.

Tabla 5.6: Parámetros del algoritmo con comunicaciones punto a punto usando una distribución fila-columna

Parámetro	Descripción
N^{oper}	Número de operaciones de comunicación
N_B	El número de bloques para cada procesador
P_{inter}	Número de procesadores que intervienen en la comunicación
N	Tamaño del problema
M_A	Número de transmisiones de la matriz A
M_B	Número de transmisiones de la matriz B
M_C	Número de transmisiones de la matriz C

Los experimentos que se hicieron en cada operación colectiva (*broadcast, scatter y gather*) fueron los siguientes:

- Se realizaron transmisiones desde el nodo origen a un número fijo de procesadores y variando el tamaño del problema.
- Se realizaron transmisiones desde el nodo origen a un número diferente de procesadores y con un mismo tamaños del problema.

Por último, para representar el número de operaciones de cómputo de todos los algoritmos, utilizamos la siguiente ecuación:

$$N_p^{Opers} = N * N \tag{5.2}$$

Donde N_p^{Opers} es el total de operaciones de cómputo que realiza el algoritmo. Una operación puede ser vista por ejemplo, al multiplicar una fila y una columna de las matrices A y B, y da como resultado un elemento de la matriz C (Figura 5.2).

Para obtener el tiempo de una operación del problema, se tomó el tiempo de procesamiento de una operación 30 veces, para promediar el tiempo que tarda un procesador en computar dicha operación y así obtener el tiempo de una operación básica del problema T_{bas}. Por lo tanto, para estimar el tiempo de cómputo es dado por:

$$T_{p_i}^{Comp} = (N_p^{Opers} * T_{bas})/P \tag{5.3}$$

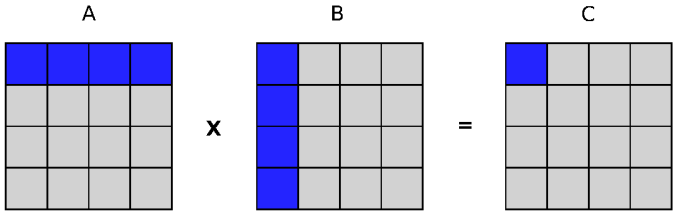

Figura 5.2: Ejemplo de una operación de una multiplicación de matrices

Por otra parte, en cada algoritmo se realizaron experimentos con diferente número de procesadores y con distintos tamaños de problemas, donde se observó el comportamiento del tiempo de cómputo, del tiempo de comunicación y del tiempo total de ejecución, y se compararon con el tiempo estimado por los modelos propuestos. Los experimentos se realizaron en el *cluster* XENA del Departamento de Computación del Centro de Investigación y de Estudios Avanzados del IPN (CINVESTAV). El *cluster* cuenta con las características mostradas en la Tabla 5.7.

Tabla 5.7: Características del *cluster* xena

Componentes
32 nodos (computadoras)
Una red de 1 Gbit Ethernet
Un *switch* con 48 puertos GbE
Procesadores quad core hyperthreading a 2.7 GHz
8 MB de cache
4 GB en memoria RAM
Fedora 11 como sistema operativo
openMPI versión 1.4.1

En la siguiente sección se muestran los resultados que se obtuvieron al estimar el tiempo de ejecución de los algoritmos vistos.

5.2. Resultados

Para obtener una muestra del tiempo medido de ejecución de cada algoritmo en el *cluster* mencionado en la sección anterior, se promediaron los tiempos medidos de 32 muestras tomadas, donde se eliminó el tiempo máximo y el mínimo, para obtener un tiempo de ejecución promedio. A continuación se muestran los tiempos medidos de las comunicaciones, del cómputo y de los tiempos totales de ejecución, los cuales son

comparados con los tiempos estimados de comunicaciones, de cómputo y de ejecución.

5.2.1. Estimación de la comunicaciones punto a punto

En primer lugar se experimentó con un par de procesadores, el experimento consiste en variar el número de bytes que se transmiten desde el nodo origen al nodo destino. Los tiempos que se obtuvieron al cambiar el número de bytes, se ilustra en la Figura 5.3. Donde se puede observar que existen determinados saltos, esto se debe al tamaño del *buffer* de TCP/IP implementa al transmitir cierta cantidad de bytes. Estos saltos son a los 9 KB y 64 KB. Por otro lado, el tiempo estimado es muy cercano al tiempo medido, debido a que se consideran los factores de mayor relevancia, como son: la latencia del switch, la latencia de la red, el tipo de comunicación y el tamaño del bloque de datos.

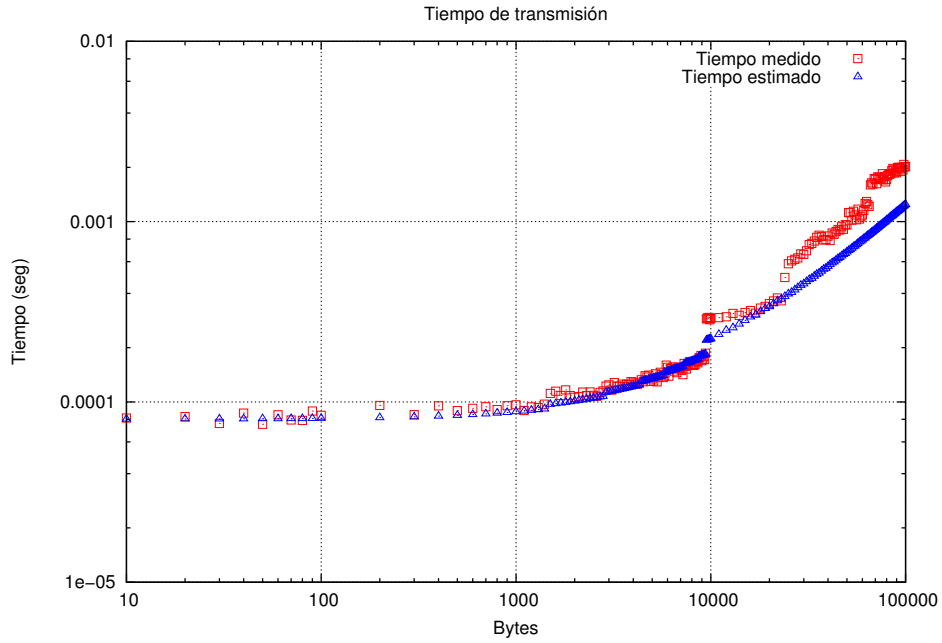

Figura 5.3: Comparación del tiempo de comunicación medido con el estimado al variar el tamaño del bloque a transmitir con operaciones punto a punto

A continuación se experimentó con dos matrices, una de tamaño $N = 3000$ y otra de tamaño $N = 4000$. Cada matriz se transmite desde 2 hasta 30 procesadores. Los tiempos obtenidos para $N = 3000$ se ilustran en Figura 5.4, y para $N = 4000$ se ilustran en la Figura 5.5. En general los tiempos estimados son muy cercanos a los tiempos medidos. Note que mientras mayor sea el número de nodos involucrados en la comunicación, menor es la diferencia entre el tiempo medido y el estimado, debido a factores como el tiempo de copiado a la cache que deja de influir en la transmisión y toma mayor relevancia el tiempo de transmisión, así como el tiempo de espera de cada

nodo para que se le envíen los datos que le corresponden, debido a la transmisión desde el nodo origen a los nodos destinos es secuencial. Además, otro factor que influye en las comunicaciones es la retransmisión de los paquetes de TCP/IP, esto es si un paquete en el nodo destino no llega o está erróneo, hace la petición al nodo origen para que se le retransmita dicho paquete.

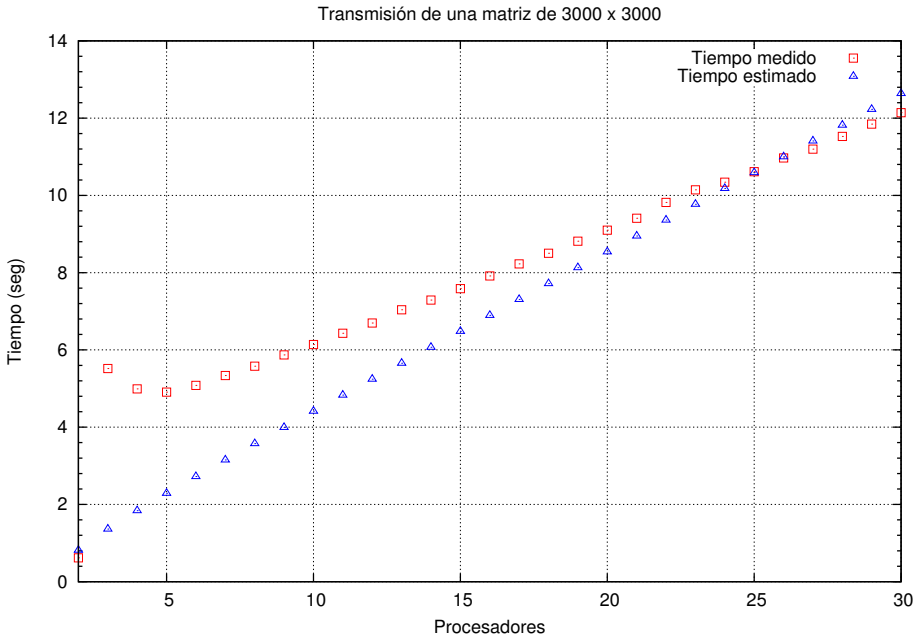

Figura 5.4: Comparación del tiempo de comunicación medido con el estimado al transmitir una matriz de 3000 x 3000 con operaciones punto a punto

Con todos los factores que involucran las comunicaciones, el tiempo estimado con los modelos propuestos para comunicaciones punto a punto es muy cercano al tiempo medido, debido a que los modelos propuestos engloban todos los factores que conlleva la comunicación entre los nodos. Por lo tanto, se establece que el modelo propuesto ayuda a estimar el tiempo de comunicación para operaciones punto a punto de manera muy cercana. Además es importante tener en mente que este tipo de operaciones son la base para las demás operaciones que se realizan en MPI.

Además, se observa en las Figuras 5.4 y 5.5 que tienen el mismo comportamiento pero con una diferencia mayor en los primeros procesadores, debido a que entre menor sea el número de procesadores influye más el tiempo de acceso a la jerarquía de la memoria, es decir acceso a la cache, a la memoria RAM, el tamaño del *buffer* de TCP/IP, entre otros aspectos.

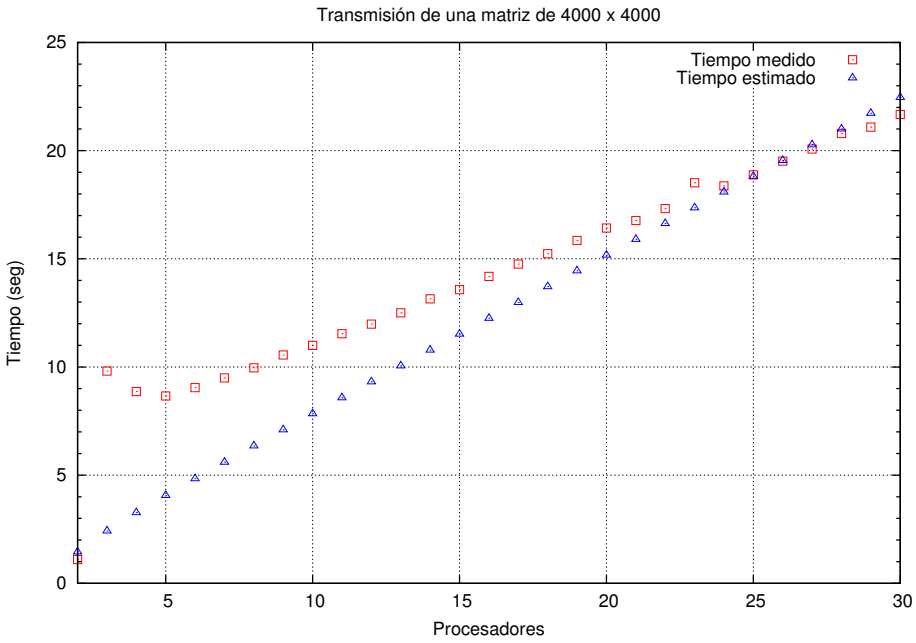

Figura 5.5: Comparación del tiempo de comunicación medido con el estimado al transmitir una matriz de 4000 x 4000 con operaciones punto a punto

5.2.2. Estimación de las comunicaciones colectivas

En las comunicaciones colectivas se manejan las operaciones, tales como *broadcast*, *scatter* y *gather*, sin embargo las operaciones colectivas tienen diferentes patrones de comunicación que dependen del tamaño de datos a transmitir, así como del número de procesadores y del número de canales del *switch*. A continuación se presentan los resultados que se obtuvieron de dichas operaciones.

5.2.2.1. Broadcast

Se realizaron experimentos donde se cambió el tamaño del problema, para observar el comportamiento con diferentes casos. Aquí es importante mencionar que la distribución que emplea esta operación depende del tamaño de datos a transmitir y el número de procesadores que son involucrados, debido a que MPI trata de optimizar la transmisión para hacerla más eficiente. Los tiempos medidos de ejecución de la operación y los tiempos estimados se presentan para 16 procesadores, cambiando el tamaño de los datos a transmitir en la Figura 5.6 y en la Figura 5.7 se transmite una matriz de 5000 x 5000, cambiando el número de procesadores. En la Figura 5.6 se observa que el tiempo estimado es muy cercano al tiempo medido, cuando son problemas relativamente pequeños pero, a excepción de los problemas muy grandes, el tiempo estimado se va

alejando del tiempo medido, debido a que influyen los parámetros que se obtuvieron experimentalmente, los cuales tienen un porcentaje de error y cuando es acumulado este porcentaje afecta en el tiempo estimado de transmisión. En la Figura 5.7 se observa que el tiempo estimado es muy cercano, cuando el número de procesadores se incrementa y cuando son pocos procesadores el tiempo estimado es significativamente mayor al tiempo medido, debido a los parámetros que se obtuvieron experimentalmente, así como los criterios para el cambio de algoritmo de transmisión.

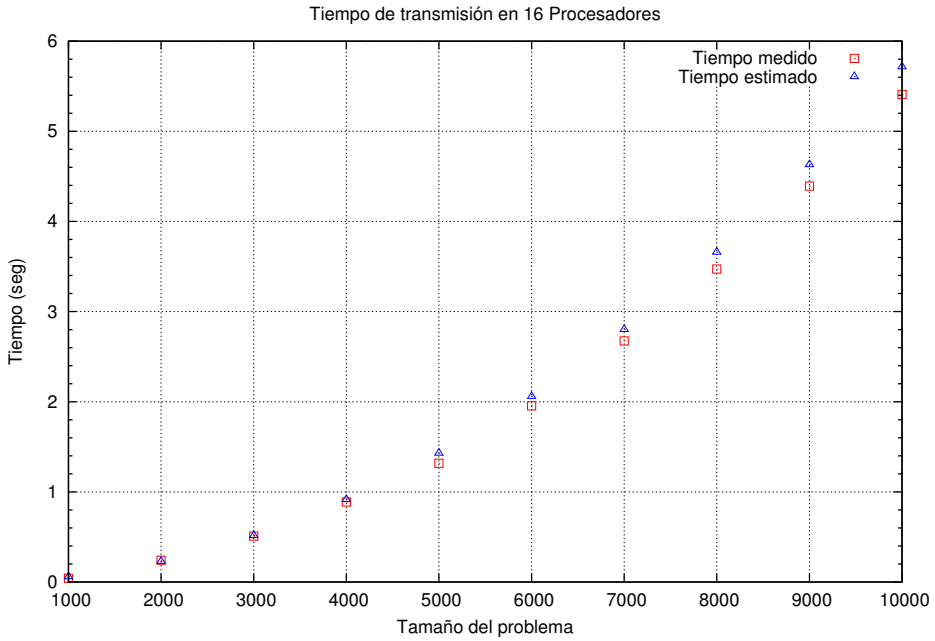

Figura 5.6: Comparación del tiempo de comunicación medido de la operación *broadcast* con el estimado cambiando el tamaño del problema

5.2.2.2. Scatter

En esta operación se realizaron experimentos donde se cambió el número de procesadores y el tamaño del problema. Observamos el comportamiento de la comunicaciones para ambos casos, al cambiar el número de procesadores y el tamaño del problema fijo (ver Figura 5.8), y con diferentes tamaños de problemas y el número de procesadores fijo (ver Figura 5.9).

En las Figuras 5.8 y 5.9, se puede observar como el tiempo estimado es muy cercano al tiempo medido, cuanto mayor es el número de procesadores, para un problema fijo. Sin embargo, cuando el tamaño del problema crece, el tiempo estimado se aleja gradualmente pero sigue manteniendo el mismo comportamiento. Como se mencionó anteriormente, el tiempo de comunicación se ve influenciado por la pérdida de paquetes,

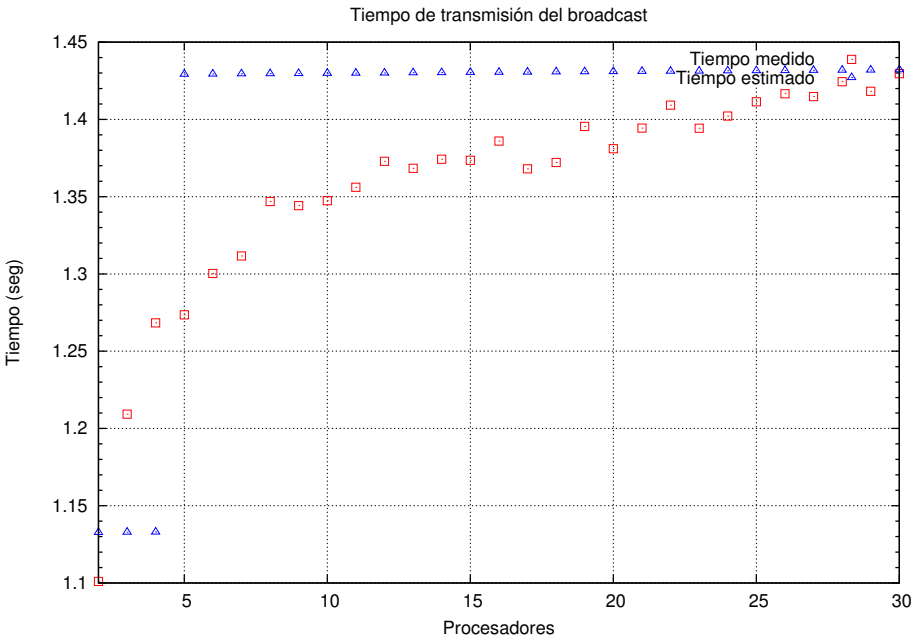

Figura 5.7: Comparación del tiempo de comunicación medido de la operación *broadcast* con el estimado al transmitir una matriz de 5000 x 5000, cambiando el número de procesadores

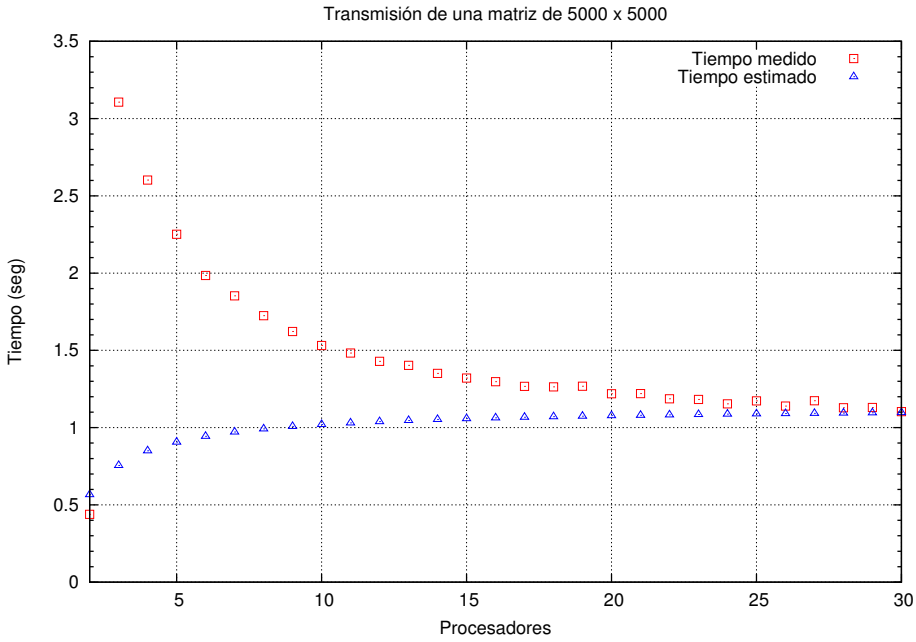

Figura 5.8: Comparación del tiempo de comunicación medido de la operación *scatter* con el estimado al transmitir una matriz de 5000 x 5000

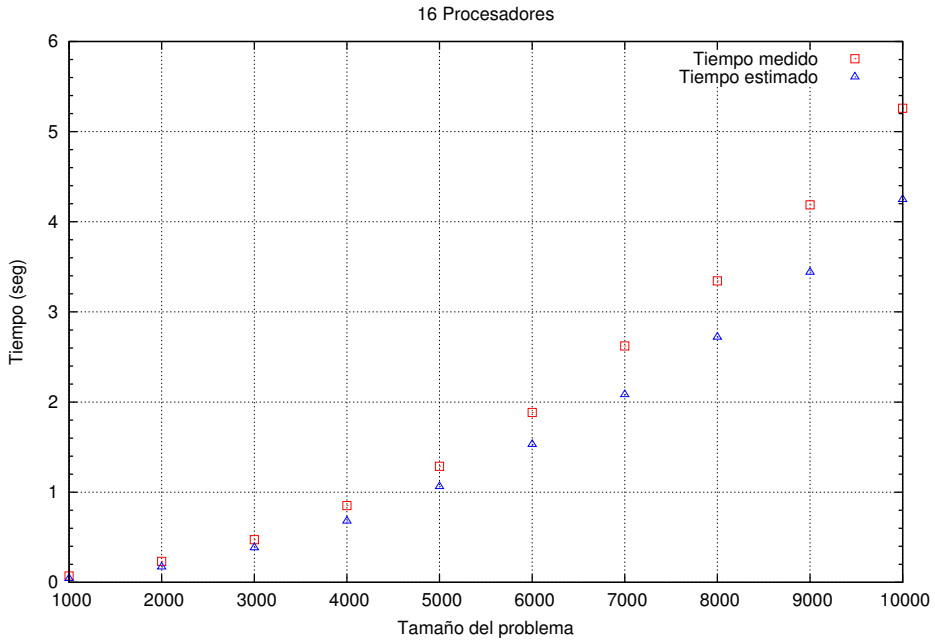

Figura 5.9: Comparación del tiempo de comunicación medido de la operación *scatter* con el estimado al cambiar el tamaño del problema

la retransmisión de los paquetes, la contención del canal de comunicación, entre otros, por consiguiente, es posible que se requiera de un modelo más detallado. Sin embargo, los modelos propuestos para esta operación son factibles para realizar la estimación para la operación *scatter*, con sus diferentes patrones de comunicación.

5.2.2.3. Gather

En esta operación de comunicación se realizaron experimentos donde se cambió el número de procesadores (ver Figura 5.10) y el tamaño del problema (ver Figura 5.11).

En las Figuras 5.10 y 5.11 se puede observar que el comportamiento de los tiempos estimados es muy cercano al tiempo medido, donde en algunos puntos, el tiempo estimado es muy aproximado respecto al tiempo medido. Esto hace que los modelos propuestos sean factibles para realizar una buena predicción del tiempo de comunicación para la operación *gather*.

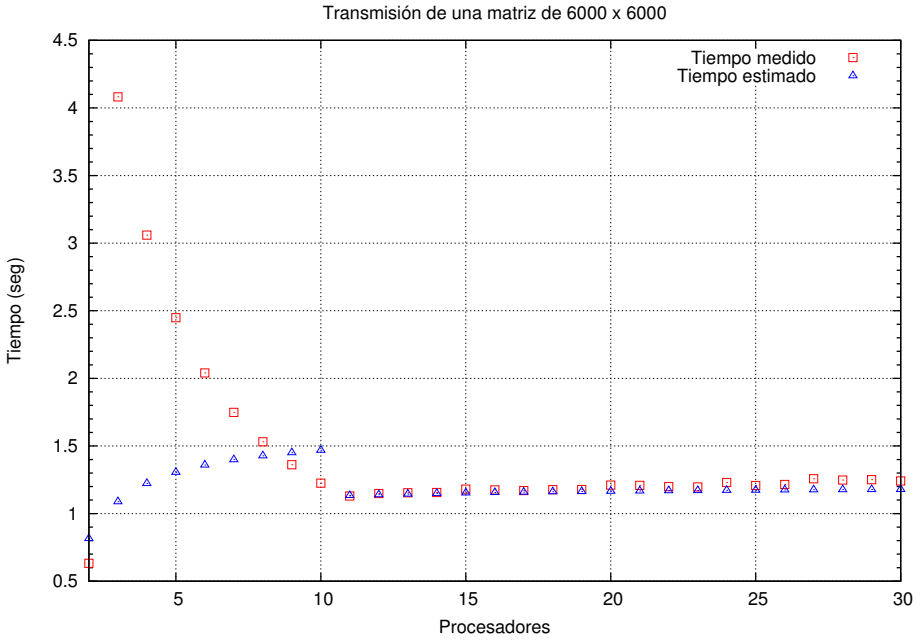

Figura 5.10: Comparación del tiempo de comunicación medido de la operación *gather* con el estimado al transmitir una matriz de 6000 x 6000

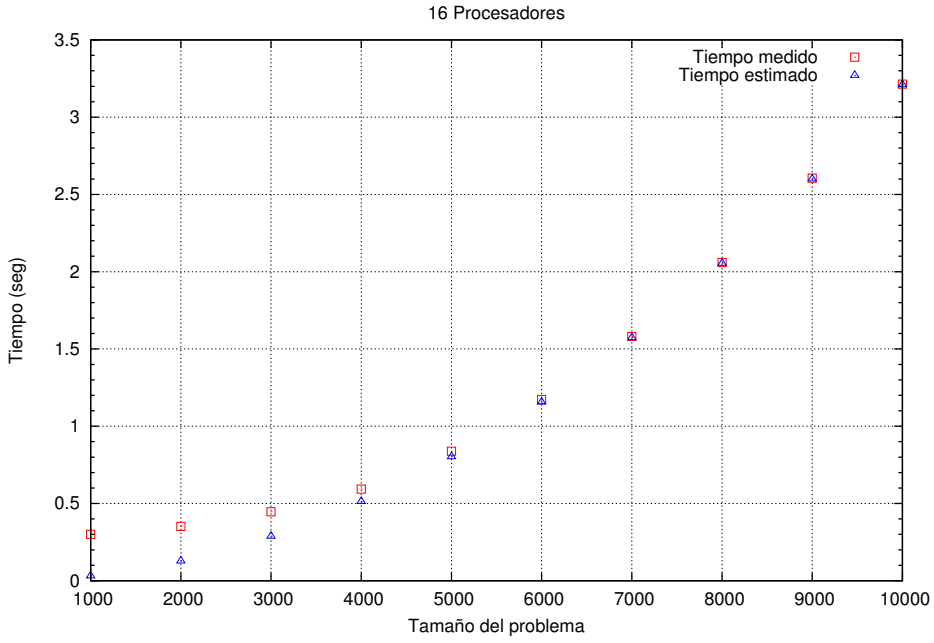

Figura 5.11: Comparación del tiempo de comunicación medido de la operación *gather* con el estimado al cambiar el tamaño del problema

5.2.3. Estimación del tiempo de cómputo

Se realizaron experimentos de un problema fijo y variando el número de procesadores (ver Figura 5.12), así como un conjunto fijo de procesadores y con diferentes tamaños de problemas (ver Figura 5.13). Para obtener el tiempo de cómputo paralelo, se promedió el tiempo que le lleva a un nodo en realizar el procesamiento del producto de dos matrices y ser comparados con los tiempos estimados aplicando el modelo de cómputo.

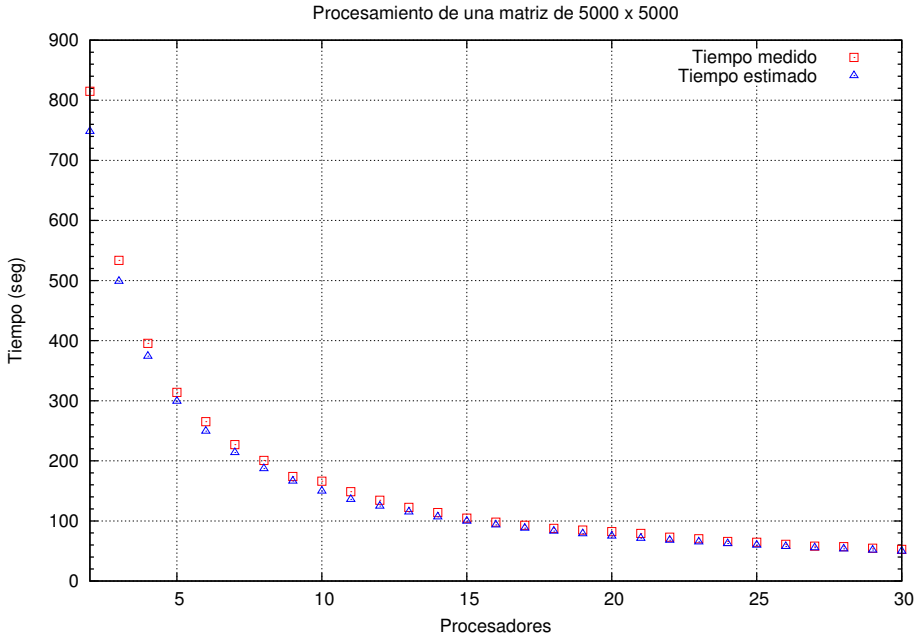

Figura 5.12: Comparación del tiempo de cómputo medido con el estimado al transmitir una matriz de 5000 x 5000

En las Figuras 5.12 y 5.13 se puede observar como el tiempo estimado de cómputo es muy aproximado con respecto al tiempo medido. Existe poca diferencia cuando el tamaño del problema es muy grande debido a la jerarquía de la memoria, esto es, el copiado de la memoria RAM a la cache, de la cache al procesador y viceversa. Pero en general presenta buen comportamiento al ser muy cercano al tiempo medido de ejecución que es lo que se busca en esta investigación.

5.2.4. Caso de estudio: Estimación del tiempo de ejecución de una multiplicación de matrices en paralelo

En este experimento se ejecutó un algoritmo de la multiplicación de matrices en paralelo del cual se tomaron 32 muestras del tiempo de ejecución. El algoritmo implementa las operaciones colectivas descritas en la sección 4.8, tanto para el algoritmo expresado en

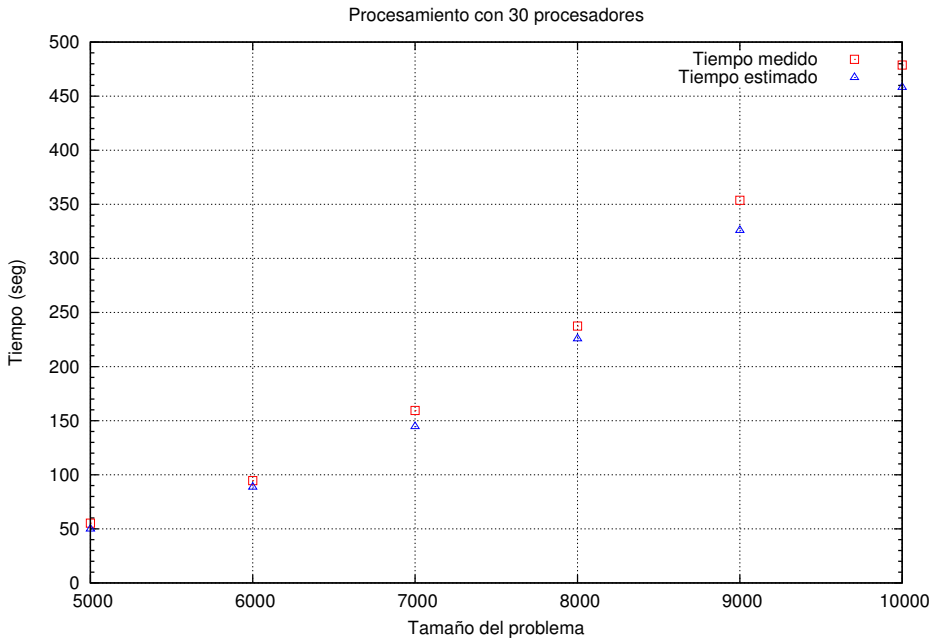

Figura 5.13: Comparación del tiempo de cómputo medido con el estimado al cambiar el tamaño del problema

MPI como para el algoritmo expresado en el intérprete que se elaboró en la herramienta *octave*. De las 32 muestras que se tomaron se eliminó la ejecución de mayor y menor tiempo, y el resto se promedió. Además se cambió el número de procesadores involucrados en la ejecución, así como la distribución de los datos, debido a las operaciones colectivas que involucra el algoritmo. En la Figura 5.14 se puede ver el comportamiento del tiempo medido de ejecución y del tiempo estimado. En la Figura 5.14 se puede observar que el tiempo estimado es muy cercano al tiempo medido de ejecución, con una ligera diferencia, esto debido a que aun faltan factores por considerar en los modelos propuesto, tales como: la contención de la red, la pérdida de paquetes y la retransmisión de los mismos, entre otros. Por lo tanto, los modelos propuestos son factibles para estimar el tiempo de ejecución de una aplicación paralela. Y para comprobar su efectividad de los modelos en la Figura 5.15 se muestran los intervalos de confianza, donde se manejo un nivel de confianza del 95 % y un error aleatorio del 5 %.

En las Figuras 5.16, y 5.17 se muestra la eficiencia de los modelos al estimar el tiempo de ejecución para matrices de 2000 y 5000, el cual es muy cercano al tiempo medido. Además, se presentan los intervalos de confianza, donde se manejo un nivel de confianza del 95 % y un error aleatorio del 5 %.

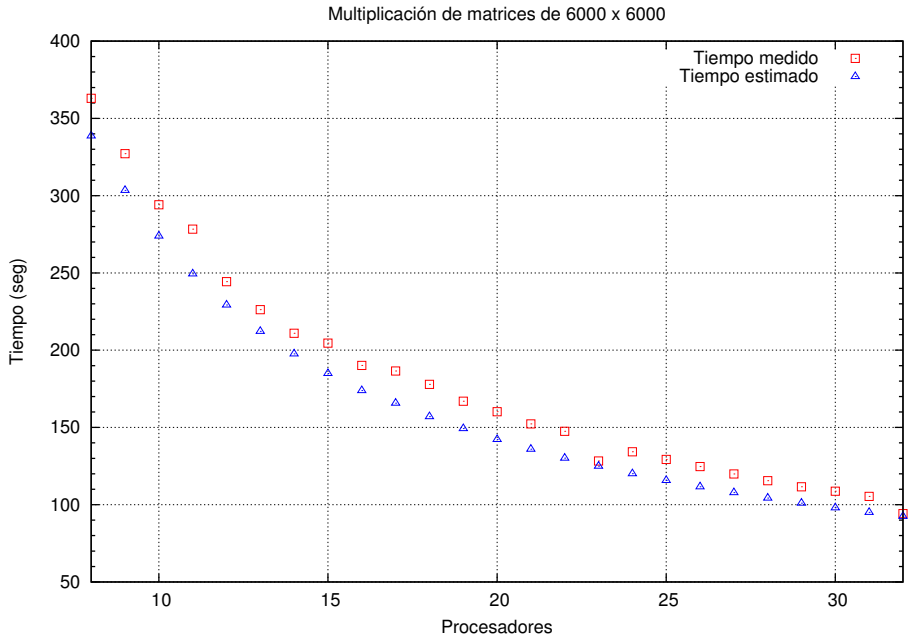

Figura 5.14: Comparación del tiempo medido de ejecución de una aplicación paralela (multiplicación de matrices) de tamaño 6000 con el tiempo estimado de ejecución

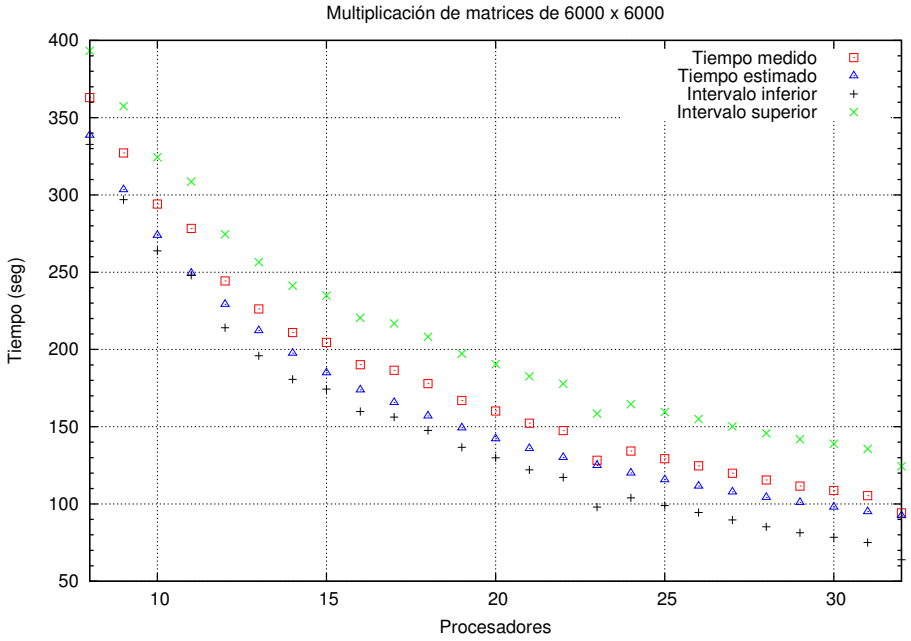

Figura 5.15: Intervalos de confianza del tiempo medido de ejecución de una aplicación paralela (multiplicación de matrices) de tamaño 6000 y la efectividad de los modelos propuestos

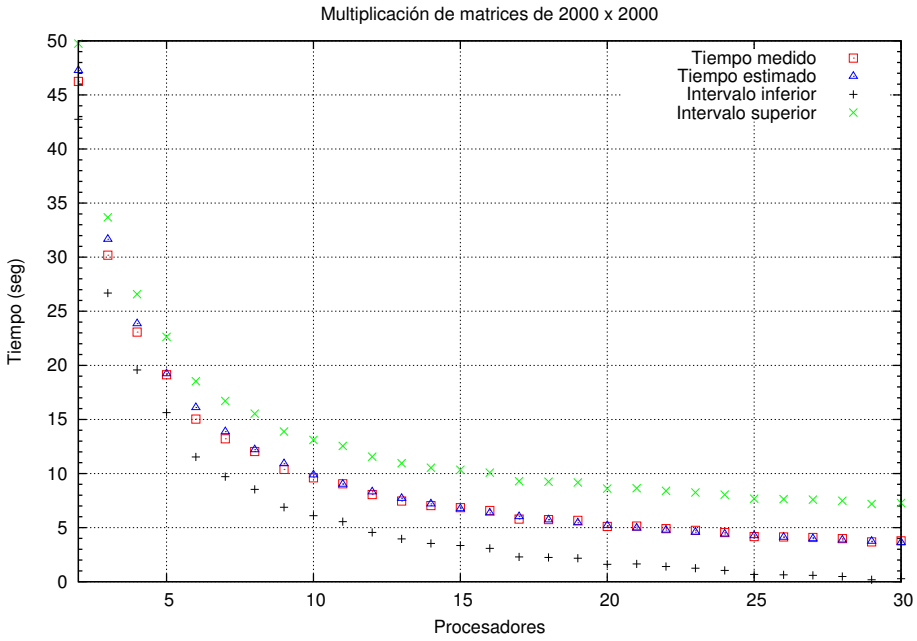

Figura 5.16: Intervalos de confianza del tiempo medido de ejecución de una aplicación paralela (multiplicación de matrices) de tamaño 2000 y la efectividad de los modelos propuestos

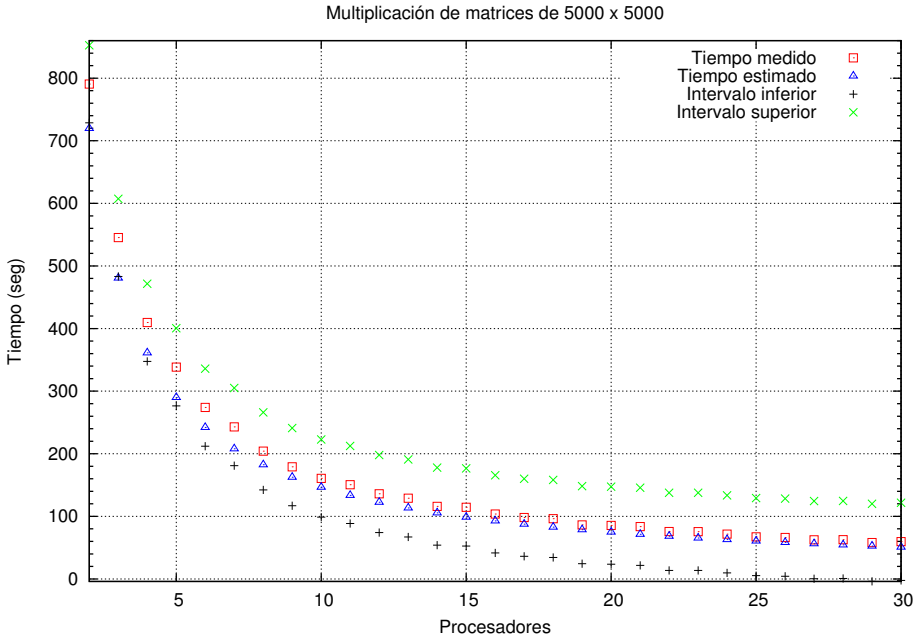

Figura 5.17: Intervalos de confianza del tiempo medido de ejecución de una aplicación paralela (multiplicación de matrices) de tamaño 5000 y la efectividad de los modelos propuestos

5.3. Resumen

El problema de la multiplicación de matrices en un esquema paralelo, requiere el uso continuo del procesador y la transferencia de los datos a través de la red entre los nodos. Además, es flexible para realizar cambios en el tamaño del problema, la distribución de los datos y el tamaño del bloque de los datos a transmitir. Por lo tanto, se eligió para transmitir diferentes tamaños de bloques de datos entre los procesadores para obtener el tiempo de comunicación, así como cambiar el tamaño del problema para obtener el tiempo de cómputo. Por otro lado, se estimó el tiempo de comunicación y de cómputo con los modelos propuestos.

Para obtener el tiempo medido de comunicación, se realizaron experimentos con los dos tipos de comunicación de MPI, que son: punto a punto y colectivas, donde se cambió el número de procesadores y se varió el tamaño del bloque a transmitir. Para obtener el tiempo medido de cómputo, se varió el tamaño del problema y el número de procesadores que intervienen en el procesamiento. Al comparar el tiempo medido y el estimado, se observa que los modelos propuestos realizan una buena aproximación respecto al tiempo medido, por lo tanto los modelos propuestos son factibles para realizar la estimación del tiempo de ejecución.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

En esta tesis se presentó el análisis y el modelado del tiempo de comunicación y cómputo, con el fin de estimar el tiempo de ejecución de una aplicación paralela en un *cluster*.

Para estimar el tiempo de comunicación se estudió el funcionamiento de MPI, así como el proceso que lleva para realizar la transmisión de los datos. MPI puede transmitir los datos de dos formas, con operaciones punto a punto y colectivas. Las operaciones punto a punto son usadas entre un par de procesadores y se observó que en ellas puede variar la latencia de la red al utilizar diferentes tamaños del bloque de datos a transmitir. En las operaciones colectivas se observó que MPI considera distintos patrones de comunicación, y se determinó que utiliza diferentes algoritmos que llevan acabo la transmisión, que la hacen más eficiente. Estos algoritmos son utilizados dinámicamente basados en el número de procesadores involucrados en la operación colectiva y el tamaño del bloque de datos. Por lo que en esta tesis se propusieron varios modelos analíticos que consideran estos algoritmos, lo cual permite hacer una estimación del tiempo de ejecución de estas operaciones que es muy aproximada al tiempo medido de ellas en un *cluster*.

Los algoritmos de MPI son implementados en los modelos desarrollados en esta tesis mediante una biblioteca de funciones de MPI para *octave*. Con esta biblioteca se tienen las siguientes ventajas al estimar el tiempo de ejecución de una aplicación paralela.

- Fácil de usar.
- No se requiere tener conocimiento de análisis matemático ni tener un modelo

matemático del algoritmo paralelo.

- No se requiere aprender un nuevo lenguaje.
- Fácil la migración de un algoritmo en MPI a *octave*.
- Permite obtener el tiempo de ejecución de una aplicación paralela al variar los diferentes parámetros de la aplicación paralela sin realizar grandes cambios al código de la aplicación.

También encontramos otros parámetros que influyen en la transmisión de los datos, que son: latencia y ancho de banda de la red, latencia y número de canales del *switch*, distribución y tamaño del bloque de datos, tamaño del problema, velocidad del procesador y número de procesadores. Además, existen otros aspectos que afectan de manera directa el desempeño de la aplicación paralela, tales como: el tiempo de lectura a la memoria RAM, el encolado de paquetes que realiza MPI, y la contención en la red, entre otros. Sin embargo, determinar el tiempo de operación de estos últimos parámetros es una tarea difícil que está fuera del alcance de esta tesis. De aquí concluimos que los parámetros propuestos abarcan muchos de los aspectos que conllevan las comunicaciones, por lo tanto, se logra estimar el tiempo de comunicación lo más cercano posible.

Adicionalmente, en nuestro trabajo concluimos que no sólo el tamaño de los datos a transmitir y el número de procesadores influyen, sino también la distribución de los datos, el tamaño del bloque de datos, la latencia y el número de canales del *switch*. La distribución de los datos en especial para comunicaciones colectivas, se lleva acabo con algoritmos que son activados dinámicamente por MPI, es decir MPI realiza la transmisión de los datos en una cierta cantidad de pasos que dependen del algoritmo invocado para que los nodos involucrados tengan los datos que les corresponden. Algunos algoritmos usan la técnica de saturación de la red y otros toman ventaja de las características de la red, es decir que los nodos pueden recibir y transmitir al mismo tiempo. Las técnicas que se mencionan son aplicadas para hacer más eficiente el uso de la red. Considerando que todo el tráfico que circula por la red por cualquier técnica empleada debe pasar por un *switch*, el *switch* verifica cada paquete de TCP/IP en busca de errores, donde la verificación de cada paquete involucra una latencia.

De manera general, los modelos propuestos en este trabajo son una opción competitiva para estimar el tiempo de ejecución de aplicaciones paralelas. Esto pudo observarse en el problema de multiplicación de matrices utilizado como caso de estudio, en donde se evidenció los resultados obtenidos en esta tesis, ya que los tiempos de ejecución estimados son muy cercanos a los medidos utilizando un *cluster*.

Adicionalmente, este trabajo también puede ser utilizado para diseñar, construir, cambiar y/o actualizar las características de un *cluster* en particular. Además, cuando los únicos parámetros conocidos son los del problema a resolver, la aplicación será capaz

de encontrar los parámetros que minimicen el tiempo de ejecución para el problema dado.

6.2. Trabajo futuro

Como posibles alternativas para el trabajo futuro de esta tesis podemos señalar los siguientes puntos:

- Proponer un modelo o anexar uno existente para estimar el tiempo de contención en la red.
- Proponer un modelo o anexar uno existente para estimar la pérdida de paquetes y el tiempo de retransmisión de los mismos.
- Anexar un modelo para estimar el tiempo de entrada y salida a disco duro.
- Proponer más parámetros que puedan afectar el tiempo de comunicación.
- Extender el modelo propuesto para modelar el tiempo de ejecución considerando procesadores con más de un núcleo, así como la comunicación entre ellos.
- Extender el modelo propuesto considerando nodos con GPUs.
- Desarrollar una aplicación que interprete el algoritmo de una manera más eficiente, ya que la propuesta esta muy limitada.
- Estimar el tiempo de ejecución para *clusters* heterogéneos.
- Estimar el tiempo de ejecución para *grids*.
- Modelar el tiempo de comunicación para el resto de las operaciones colectivas de MPI.
- Extender el conjunto de funciones de la biblioteca desarrollada.

Referencias

[1] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the logp model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71 – 79, 1997.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] Luiz Angelo Barchet-Estefanel and Gregory Mounie. Performance characterisation of intra-cluster collective communications. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '04, pages 254–261, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Mike Barnett, David G. Payne, Robert A. van de Geijn, and Jerrell Watts. Broadcasting on meshes with wormhole routing. *Journal of Parallel and Distributed Computing*, 35(2):111 – 122, 1996.

[5] E. Baydal, P. Lopez, and J. Duato. A simple and efficient mechanism to prevent saturation in wormhole networks. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 617 – 622, May 2000.

[6] Harvey G. Cragon. *Computer architecture and implementation*. Cambridge University Press, New York, NY, USA, 2000.

[7] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, Jul 1993.

[8] John W Eaton, David Bateman, and Soren Hauberg. *GNU Octave Manual Version 3*. Network Theory Ltd., 2008.

[9] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing*. Wiley-Interscience, 2005.

[10] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21:948–960, September 1972.

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

[12] Fayez Gebali. *Algorithms and Parallel Computing.* Wiley, 2011.

[13] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing.* MIT Press, Cambridge, MA, USA, 1994.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface.* Scientific and engineering computation. MIT Press, 1994.

[15] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI - The Complete Reference: Volume 2, The MPI-2 Extensions.* MIT Press, Cambridge, MA, USA, 1998.

[16] B. M. Hill, D. B. Harris, and J. Vyas. *Debian GNU/Linux 3.1 Bible.* Wiley, New York, 2005.

[17] Roger W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Comput.*, 20(3):389–398, Mar 1994.

[18] Sándor Juhász and Hassan Charaf. Execution time prediction for parallel data processing tasks. In *Proceedings of the 10th Euromicro conference on Parallel, distributed and network-based processing*, EUROMICRO-PDP'02, pages 31–38, Washington, DC, USA, 2002. IEEE Computer Society.

[19] T. Kielmann. Bandwidth-efficient collective communication for clustered wide area systems. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, IPDPS '00, pages 492–499, Washington, DC, USA, 2000. IEEE Computer Society.

[20] Alexey Lastovetsky, Is-Haka Mkwawa, and Maureen O'Flynn. An accurate communication model of a heterogeneous cluster based on a switch-enabled ethernet network. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems - Volume 2*, ICPADS '06, pages 15–20, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Alexey Lastovetsky, Vladimir Rychkov, and Maureen O'Flynn. Revisiting communication performance models for computational clusters. In *Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing*, IPDPS '09, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[22] Bonnie L. Miller. *AIX for UNIX professionals*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[23] Christopher Negus. *Fedora 9 and Red Hat Enterprise Linux Bible*. Wiley Publishing, 2008.

[24] Akihiro Nomura, Hiroya Matsuba, and Yutaka Ishikawa. Network performance model for TCP/IP based cluster computing. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, CLUSTER '07, pages 194–203, Washington, DC, USA, 2007. IEEE Computer Society.

[25] Behrooz Parhami. *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[26] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143, Jun 2007.

[27] J. Postel. Internet protocol. RFC 791 (Standard), Sep 1981. Updated by RFC 1349.

[28] J. Postel. Transmission control protocol. RFC 793 (Standard), Sep 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[29] T. Rauber and G. Rünger. *Parallel Programming for Multicore and Cluster Systems*. Springer Verlag, 2010.

[30] Chris Sanders. *Practical packet analysis*. No Starch Press, San Francisco, CA, USA, first edition, 2007.

[31] Domínguez Domínguez Santiago. *Difusión automática de datos bajo cómputo paralelo en clusters*. PhD thesis, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, Enero 2006.

[32] Joseph D. Sloan. *High Performance Linux Clusters: With OSCAR, Rocks, openMosix, and MPI*. O'Reilly Media, Inc., 2004.

[33] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI - The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[34] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[35] Anthony T. C. Tam and Cho-Li Wang. Realistic communication model for parallel computing on cluster. In *Proceedings of the 1st IEEE Computer Society International Workshop on Cluster Computing*, IWCC '99, pages 92–, Washington, DC, USA, 1999. IEEE Computer Society.

[36] Wi Bing Tan and P. Strazdins. The analysis and optimization of collective communications on a beowulf cluster. In *Parallel and Distributed Systems, 2002. Proceedings. Ninth International Conference on*, pages 659 – 666, Dec 2002.

[37] Paul Watters. *Solaris 10: The Complete Reference*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2005.

ANEXOS

Anexo A: Funciones implementadas en *Octave*

A continuación se presenta el código de cada una de las funciones de MPI implementadas en *octave*.

Función **MPIO_Send**:

```
% Funci n que emula la funci n MPI_Send
function MPIO_Send(buf, N, MPI_Data_type, P, tag, MPI_WORLD)
  global Tcomm;
  global OCT_Datos;
  global OCT_Procs;
  global OCT_Bandw;
  global OCT_Latencia;
  global OCT_Lsw;
  global OCT_Tbas;
  global OCT_Ttrans;
  global OCT_P_iters;
  global OCT_P_act;

  % Procesadores que interactuan
  if(P != OCT_P_act)
    OCT_P_act = P;
    OCT_P_iters = OCT_P_iters + 1;
  endif
  % Cabecera de MPI
  Hmpi = 22;
  % ********* SALTOS ************
  k9 = 9500; % En Bytes
  % Los datos + la cabecera de MPI
  bytes_transmitidos = (N * MPI_Data_type) + Hmpi;

  % Datos + las cabeceras TCP/IP
  [bytes_transmitidos, Npaq] = det_btrans(bytes_transmitidos);
```

```
28
29    % Tiempo de transmision sin overhead
30    Ttrans = (bytes_transmitidos * 8) / OCT_Bandw;
31    OCT_Ttrans = Ttrans + (Npaq * OCT_Lsw);
32
33    % ********* SALTOS ************
34    % Overhead del salto
35    if(bytes_transmitidos >= k9)
36      OCT_O_salto = 0.0000374934;
37    else
38      OCT_O_salto = 0;
39    endif
40
41    % Tiempo estimado de transmision sin latencia de la red
42    OCT_Ttrans = OCT_Ttrans + OCT_O_salto; % Cuando solo se hace
         un send
43    Tcomm = Tcomm + OCT_Ttrans; % Cuando se realizan varios sends
         y se acumula el tiempo
44  endfunction
```

Función **MPIO_Recv**:

```
1   % Funci n que emula la funci n MPI_Send
2   function MPIO_Recv(buf, N, MPI_Data_type, P, tag, MPI_WORLD,
       status)
3     global Tcomm;
4     global OCT_Datos;
5     global OCT_Procs;
6     global OCT_Bandw;
7     global OCT_Latencia;
8     global OCT_Lsw;
9     global OCT_Tbas;
10    global OCT_Ttrans;
11    global OCT_P_iters;
12    global OCT_P_act;
13
14    % Procesadores que interactuan
15    if(P != OCT_P_act)
16      OCT_P_act = P;
17      OCT_P_iters = OCT_P_iters + 1;
18    endif
19    % Cabecera de MPI
20    Hmpi = 22;
21    % ********* SALTOS ************
22    k9 = 9500; % En Bytes
23    % Los datos + la cabecera de MPI
```

```
24    bytes_recibidos = (N * MPI_Data_type) + Hmpi;
25
26    % Datos + las cabeceras TCP/IP
27    [bytes_recibidos, Npaq] = det_btrans(bytes_recibidos);
28
29    % Tiempo de transmision sin overhead
30    Ttrans = (bytes_recibidos * 8) / OCT_Bandw;
31    OCT_Ttrans = Ttrans + (Npaq * OCT_Lsw);
32
33    % ********* SALTOS ************
34    % Overhead del salto
35    if(bytes_recibidos >= k9)
36      OCT_O_salto = 0.0000374934;
37    else
38      OCT_O_salto = 0;
39    endif
40
41    % Tiempo estimado de recepcion sin latencia de la red
42    OCT_Ttrans = OCT_Ttrans + OCT_O_salto; % Cuando solo se hace
         un send
43    Tcomm = Tcomm + OCT_Ttrans; % Cuando se realizan varios recvs
         y se acumula el tiempo
44  endfunction
```

Función **MPIO_Bcast**:

```
1   % Funci n que emula la funci n MPI_Bcast
2   function MPIO_Bcast(buf, total_elems, MPI_Data_type, root,
       MPI_COMM_WORLD)
3     global Tcomm;
4     global OCT_Datos;
5     global OCT_Procs;
6     global OCT_Bandw;
7     global OCT_Latencia;
8     global OCT_Lsw;
9     global OCT_Tbas;
10    global OCT_P_iters;
11
12    OCT_O_salto = 0;
13    Ttrans = 0;
14    OCT_Canales_sw = 4;
15    OCT_Num_Lsw = 0;
16    % Cabecera de MPI
17    Hmpi = 22;
18    % ********* SALTOS ************
19    _9k = 9500; % En Bytes
```

```
20      _2k = 2048;
21      _1M = 1000000;
22
23      bytes_trans = total_elems * MPI_Data_type;
24      bytes_trans = (double)(total_elems * MPI_Data_type) / 2; #
           Swap
25
26      % Inicio para una matriz
27      if (OCT_Procs <= 10 && bytes_trans => _2k && bytes_trans <
           _1M)
28        segmento = bytes_trans / 2;
29        %Binomial with swap
30        if (rem (OCT_Procs, 2) == 0 && OCT_Procs > 2)
31          iters = 3; % Procesadores que interactuan
32        else
33          iters = 2; % Procesadores que interactuan
34        endif
35
36        segmento = segmento + (Hmpi * iters); % Se le agrega las
             cabeceras de MPI
37        bytes_trans = segmento;
38
39        OCT_P_iters = OCT_P_iters + iters; % Procesadores que
             interactuan en Pipeline
40      endif
41
42      if (OCT_Procs <= 10 && bytes_trans < _2k)
43        iters = ceil(log2(OCT_Procs)); % Cuando es binomial
44        bytes_trans = bytes_trans + (Hmpi * iters); % Se le agrega
             las cabeceras de MPI
45
46        OCT_P_iters = OCT_P_iters + iters; % Procesadores que
             interactuan en Pipeline
47      endif
48
49      if (bytes_trans > _1M)
50        iters = OCT_Procs - 1;
51        iters = 1;
52        bytes_trans + (Hmpi * iters); % Se le agrega las cabeceras
             de MPI
53        bytes_trans = bytes_trans + (Hmpi * (OCT_Procs - 1)); % Se
             le agrega las cabeceras de MPI
54
55        % Estimaci n del n mero de canales
56        OCT_Num_Lsw = ceil(OCT_Procs / OCT_Canales_sw); % Nuevo
57        if (OCT_Num_Lsw > 2)
```

```
58          OCT_Num_Lsw = 2;
59        endif
60        OCT_P_iters = OCT_Procs - 1; % Procesadores que interactuan
              en Pipeline
61      endif
62      % Bytes transmitidos
63      [bytes_transmitidos, Npaq] = det_btrans(bytes_trans);
64
65      % ********* SALTOS ************
66      if (bytes_transmitidos >= _9k)
67        OCT_O_salto = 0.0000374934;
68      else
69        OCT_O_salto = 0;
70      endif
71
72      % Tiempo de transmisi n
73      Ttrans = (bytes_transmitidos * 8) / OCT_Bandw;
74      % Tiempo de transmisi n mas el salto
75      Ttrans = Ttrans + (OCT_O_salto * (OCT_Procs - 1));
76
77      % Tiempo de comunicaci n estimado
78      Tcomm = Tcomm + (Ttrans * iters) + ((Npaq * OCT_Lsw) *
           OCT_Num_Lsw); % Nuevo
79  endfunction
```

Función **MPIO_Scatter**:

```
1   # Funci n que emula la funci n MPI_Scatter
2   function MPIO_Scatter(send_A, seg_A, MPI_Data_type_s, recv_a,
        seg_a, MPI_Data_type_r, root, MPI_COMM_WORLD)
3     global Tcomm;
4     global OCT_Datos;
5     global OCT_Procs;
6     global OCT_Bandw;
7     global OCT_Latencia;
8     global OCT_Lsw;
9     global OCT_Tbas;
10    global OCT_P_iters;
11
12    % N mero de canales
13    OCT_Canales_sw = 8;
14    OCT_Num_Lsw = 0;
15
16    OCT_O_salto = 0;
17    Ttrans = 0;
18
```

```
19      % ********* SALTOS ************
20      k9 = 9500; % En Bytes
21
22      Hmpi = 22;
23      segm_bytes = 0;
24      Npaq = 0;
25
26      segm_bytes = seg_A * MPI_Data_type_s;
27
28      % Flat tree
29      % Datos obtenidos experimentalmente
30      % Si el segmento es >= 300 Bytes    P <= 10
31      if (segm_bytes >= 300 && segm_bytes < 1000000) # && OCT_Procs
            <= 10)  #Flat tree o secuencial
32        segm_bytes = segm_bytes + Hmpi;
33
34        % Bytes transmitidos + Cabeceras
35        [bytes_transmitidos, Npaq] = det_btrans(segm_bytes);
36
37        # Distribuci n
38        iters = OCT_Procs - 1;
39
40        % Tiempo de transmisi n ideal
41        Ttrans = (bytes_transmitidos * 8) / OCT_Bandw;
42
43        % Tiempo de transmisi n estimado
44        Tcomm = Tcomm + (iters * Ttrans) + ((Npaq * OCT_Lsw) *
              iters);
45
46        % # Procesadores que interactuan en Secuencial
47        OCT_P_iters = OCT_P_iters + iters;
48      elseif (segm_bytes < 300)  # Binomial tree
49        bytes_transmitidos = 0;
50        Npaq = 0;
51        procs = OCT_Procs;
52        iters = ceil(log2(OCT_Procs));
53
54        for i=0:iters
55          bytes_trans = 0;
56          paqs = 0;
57          bloqs = 2^i;
58
59          if procs - bloqs <= 0
60            bloqs = procs;
61          endif
62
```

```
63        bytes_trans = (segm_bytes * bloqs) + Hmpi;
64        [bytes_trans, paqs] = det_btrans(bytes_trans);
65        bytes_transmitidos = bytes_transmitidos + bytes_trans;
66        Npaq = Npaq + paqs;
67        if procs - bloqs > 0
68          procs = procs - bloqs;
69        endif
70      endfor
71
72      % Tiempo de transmisi n ideal
73      Ttrans = (bytes_transmitidos * 8) / OCT_Bandw;
74
75      % Tiempo de transmisi n estimado
76      Tcomm = Tcomm + Ttrans + (Npaq * OCT_Lsw);
77
78      % Procesadores que interactuan en Binomial tree
79      OCT_P_iters = OCT_P_iters + iters;
80    elseif (segm_bytes >= 1000000)
81      segm_bytes = segm_bytes + Hmpi;
82      % Bytes transmitidos + Cabeceras
83      [bytes_transmitidos, Npaq] = det_btrans(segm_bytes);
84
85      % Tiempo de transmisi n sin overhead
86      Ttrans = (bytes_transmitidos * 8) / OCT_Bandw;
87
88      % ********* SALTOS ************
89      if (bytes_transmitidos >= k9)
90        OCT_O_salto = 0.0000374934;
91      else
92        OCT_O_salto = 0;
93      endif
94
95      % Estimaci n del n mero de canales
96      OCT_Num_Lsw = ceil(OCT_Procs / OCT_Canales_sw);
97
98      % Tiempo de transmisi n mas el salto
99      Ttrans = Ttrans + OCT_O_salto;
100
101     % Tiempo de transmisi n estimado
102     Tcomm = Tcomm + (Ttrans * (OCT_Procs - 1)) + ((Npaq *
          OCT_Lsw) * OCT_Num_Lsw);
103     % Procesadores que interactuan en Secuencial
104     OCT_P_iters = OCT_P_iters + 1;
105   endif
106 endfunction
```

Función **MPIO_Gather**:

```
% Funci n que emula la funci n MPI_Gather
function MPIO_Gather(send_a, seg_A, MPI_Data_type_s, recv_A,
   seg_a, MPI_Data_type_r, root, MPI_COMM_WORLD)
  global Tcomm;
  global OCT_Datos;
  global OCT_Procs;
  global OCT_Bandw;
  global OCT_Latencia;
  global OCT_Lsw;
  global OCT_Tbas;
  global OCT_P_iters;

  % N mero de canales
  OCT_Canales_sw = 8;
  OCT_Num_Lsw = 0;
  OCT_O_salto = 0;
  Ttrans = 0;

  % ********* SALTOS ************
  k9 = 9500; % En Bytes
  Hmpi = 22;
  segm_bytes = 0;
  Npaq = 0;
  segm_bytes = seg_A * MPI_Data_type_s;
  segm_bytes = segm_bytes + Hmpi;

  if (OCT_Procs < 11) # Secuencial
    [bytes_transmitidos, Npaq] = det_btrans(segm_bytes);
    % Tiempo de transmisi n sin latencia
    Ttrans = (bytes_transmitidos * 8) / OCT_Bandw;

    Tcomm = Tcomm + ((OCT_Procs - 1) * Ttrans) + (Npaq *
       OCT_Lsw) * (OCT_Procs - 1);
    % Procesadores que interactuan en Secuencial
    OCT_P_iters = OCT_P_iters + (OCT_Procs - 1);
  else % pipeline
    bytes_trans = segm_bytes;
    [bytes_transmitidos, Npaq] = det_btrans(bytes_trans);
    % Tiempo de transmisi n sin latencia
    Ttrans = (bytes_transmitidos * 8) / OCT_Bandw;
    % ********* SALTOS ************
    if (bytes_transmitidos >= k9)
      OCT_O_salto = 0.0000374934;
    else
```

```
43          OCT_O_salto = 0;
44        endif
45
46        % Tiempo de transmisi n mas el salto
47        Ttrans = Ttrans + OCT_O_salto;
48        % Estimaci n del n mero de canales
49        OCT_Num_Lsw = ceil(OCT_Procs / OCT_Canales_sw);
50
51        Tcomm = Tcomm + (Ttrans * (OCT_Procs - 1)) + ((Npaq *
             OCT_Lsw) * OCT_Num_Lsw);
52        % Procesadores que interactuan en Pipeline
53        OCT_P_iters = OCT_P_iters + 1;
54      endif
55    endfunction
```

Función **MPIO_Barrier**:

```
1   % Funci n que emula la funci n MPI_Barrier
2   function MPIO_Barrier(MPI_COMM_WORLD)
3     global Tcomm;
4     global Tsync;
5     global OCT_Procs;
6     global OCT_P_iters;
7
8     Ttrans = 0;
9     Hmpi = 22;
10    Npaq = 0;
11    bytes_trans = 4; % Si solo transmite un entero
12    bytes_trans = bytes_trans + Hmpi;
13
14    [bytes_transmitidos, Npaq] = det_btrans(bytes_trans);
15    % Tiempo de transmisi n sin latencias
16    Ttrans = (bytes_transmitidos * 8) / OCT_Bandw;
17
18    % Tiempo de sincronizacion
19    Tsync = (Ttrans * (OCT_Procs - 1)) + ((Npaq * OCT_Lsw) * (
         OCT_Procs - 1));
20  endfunction
```

Anexo B: Algoritmo de multiplicación de matrices en un esquema paralelo y en *octave*

B.1. Algoritmo implementado en MPI con comunicaciones colectivas

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define root    0

int main(int argc, char *argv[])
{
  int nprocs, rank, muestras;
  int N, segmento, telems, seg, elems;
  float *A, *B, *C;
  float *FA, *FC;
  int i, j, p, q, m, k;
  int imax, imin;

  double ti_comm, tf_comm, tp_comm = 0, *m_comm;
  double ti_comp, tf_comp, tp_comp = 0, *m_comp;
  double tp_exec, *m_exec;

  if(argc < 3)
  {
    printf("USO: programa problema muestras\n");
    return 1;
  }
```

```
25
26      N = atoi(argv[1]);
27      muestras = atoi(argv[2]);
28
29      m_exec = (double*)calloc(muestras + 2, sizeof(double));
30      m_comm = (double*)calloc(muestras + 2, sizeof(double));
31      m_comp = (double*)calloc(muestras + 2, sizeof(double));
32
33      MPI_Init(&argc, &argv);
34      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
35      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
36      MPI_Status status;
37
38      telems = N * N;
39      segmento = telems / nprocs;
40
41      B = (float*)calloc(telems, sizeof(float));
42
43      elems = N / nprocs;
44      FA = (float*)calloc(segmento, sizeof(float));
45      FC = (float*)calloc(segmento, sizeof(float));
46
47      if(rank == 0)
48      {
49        A = (float*)calloc(telems, sizeof(float));
50        C = (float*)calloc(telems, sizeof(float));
51
52        srand(time(NULL));
53        for(i=0, p=0; i < N; i++, p+=N)
54        {
55          for(j=0, q=p; j < N; j++, q++)
56          {
57            A[q] = rand()%10+1;
58            B[q] = rand()%10+1;
59          }
60        }
61      }
62
63      MPI_Barrier(MPI_COMM_WORLD);
64
65      // Transmitir matriz A
66      MPI_Scatter(A, segmento, MPI_FLOAT, FA, segmento, MPI_FLOAT,
           root, MPI_COMM_WORLD);
67      // Transmitir matriz B
68      MPI_Bcast(B, telems, MPI_FLOAT, root, MPI_COMM_WORLD);
69
```

```
70      // COMPUTO
71      seg = 0;
72      for(i=0; i < elems; i++)
73      {
74        for(j = seg, p=0; j < seg + N; j++, p++)
75        {
76          for(k = p, q = seg; k < telems; k += N, q++)
77            FC[j] += FA[q] * B[k];
78        }
79        seg += N;
80      }
81
82      MPI_Gather(FC, segmento, MPI_FLOAT, C, segmento, MPI_FLOAT,
           root, MPI_COMM_WORLD);
83
84      MPI_Finalize();
85  }
```

En las Figuras B.1 y B.3 se muestran los **tiempos de ejecución medidos** para el algoritmo de la multiplicación de matrices de tamaño 2000 y 5000 respectivamente. En las Figuras B.2 y B.4 se muestran los **tiempos de ejecución estimados** del algoritmo de la multiplicación de matrices. Todas las ejecuciones se realizaron con comunicaciones colectivas en 30 nodos.

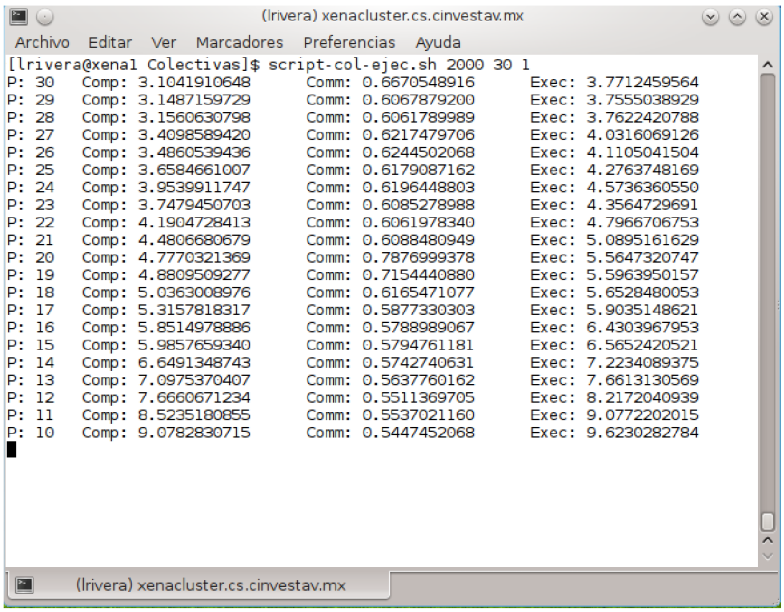

Figura B.1: Tiempo medido para el algoritmo de la multiplicación de matrices de tamaño 2000.

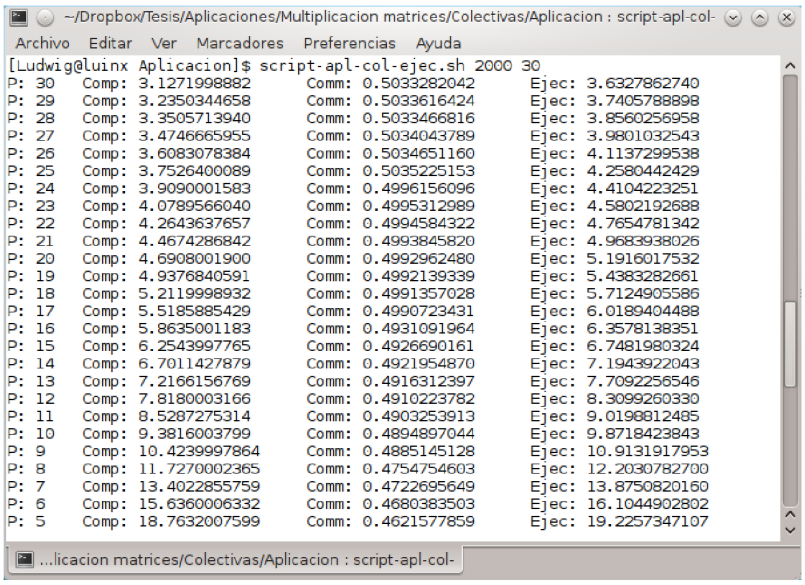

Figura B.2: Tiempo estimado para el algoritmo de la multiplicación de matrices de tamaño 2000.

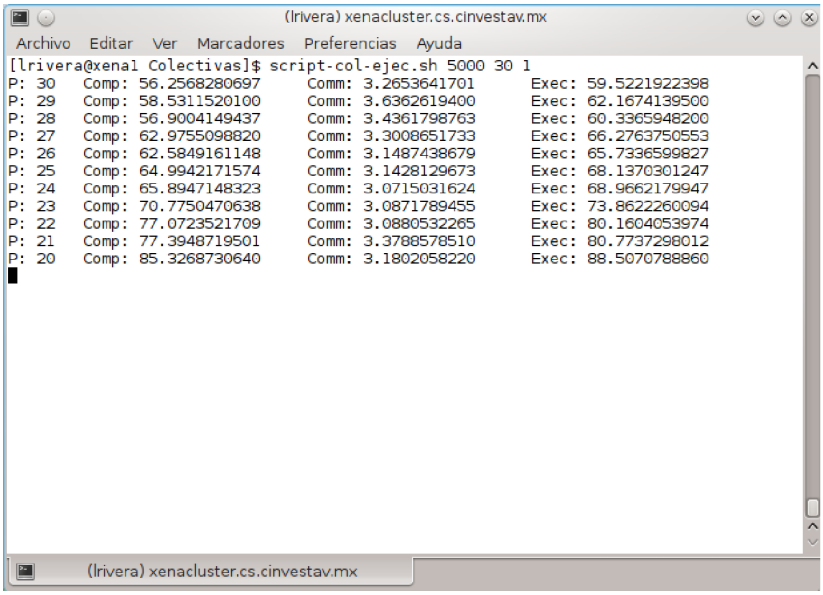

Figura B.3: Tiempo medido para el algoritmo de la multiplicación de matrices de tamaño 5000.

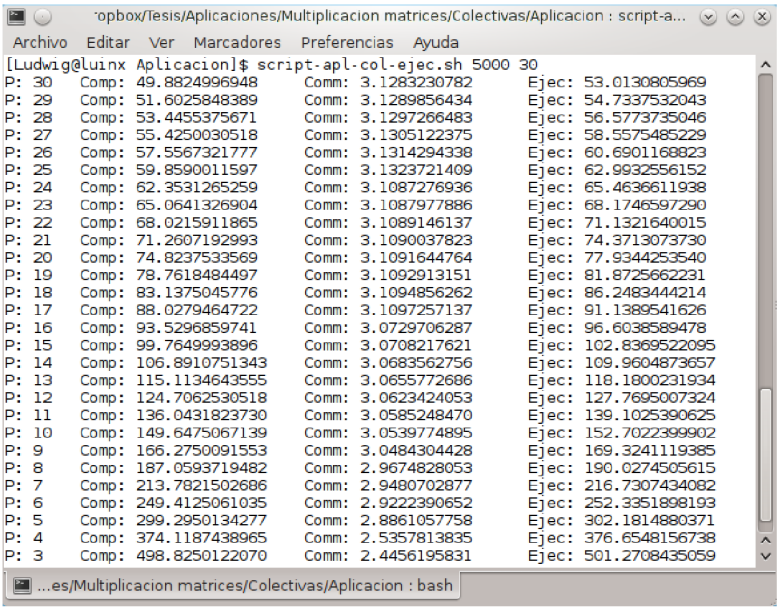

Figura B.4: Tiempo estimado para el algoritmo de la multiplicación de matrices de tamaño 5000.

B.2. Algoritmo implementado con la biblioteca desarrollada con comunicaciones colectivas

```
1  function [Texec, Tcomp, Tcomm] = mmt_col_exp(TP, P, B, L, Lsw,
       Tbas)
2      % Variables propias de la biblioteca
3      global OCT_Datos; % Datos a transmitir
4      global OCT_Procs; % N mero de procesadores
5      global OCT_Bandw; % Ancho de banda
6      global OCT_Latencia; % Latencia de la red
7      global OCT_Lsw; % Latencia del switch
8      global OCT_Tbas; % Tiempo b sico de la aplicaci n
9
10     % Tiempos
11     global Texec; % Tiempo estimado de ejecuci n
12     global Tcomm; % Tiempo de comunicaci n
13     global Tcomp; % Tiempo de c mputo
14
15     % Utileria
16     global OCT_P_iters; # N mero de iteracciones
17
18     % Variables de MPI
```

```
19      global MPIO_CHAR;
20      global MPIO_SHORT;
21      global MPIO_INT;
22      global MPIO_FLOAT;
23      global MPIO_DOUBLE;
24      global MPIO_LONG;
25
26      % Inicializar valores de la biblioteca
27      OCT_Ivar(TP, P, B, L, Lsw, Tbas);
28      Tcomm = 0;
29      Tcomp = 0;
30      Texec = 0;
31      OCT_P_iters = 0;
32      OCT_Tbas = Tbas;
33
34      % ### Inicio del Algoritmo ###
35      % Elementos que para cada proceso
36      elems = OCT_Datos / OCT_Procs;
37
38      % Segmento
39      segmento = elems * OCT_Datos;
40
41      % Total de elementos
42      total_elems = OCT_Datos * OCT_Datos;
43
44      % Inicializar variables
45      root = 0;
46      MPI_COMM_WORLD = 0;
47      send_A = 0;
48      send_c = 0;
49      recv_a = 0;
50      recv_C = 0;
51      bufB = 0;
52
53      % COMUNICACION
54      MPIO_Barrier(MPI_COMM_WORLD);
55      % Distribucion de A (Desde root)
56      MPIO_Scatter(send_A, segmento, MPIO_FLOAT, recv_a, segmento,
           MPI_FLOAT, root, MPI_COMM_WORLD);
57
58      % Distribucion de B (Desde root)
59      MPIO_Bcast(bufB, total_elems, MPIO_FLOAT, root,
           MPI_COMM_WORLD);
60
61      % COMPUTO
62      % Solo el tiempo de un proceso
```

```
63    total_opers = total_elems / OCT_Procs;
64
65    % Tiempo basico de una unidad
66    OCT_Procesamiento();
67
68    % Tiempo de computo
69    Tcomp = Tcomp * total_opers;
70
71    % COMUNICACION
72    % Recepcion de C (En root)
73    MPIO_Gather(send_c, segmento, MPIO_FLOAT, recv_C, segmento,
          MPI_FLOAT, root, MPI_COMM_WORLD);
74
75    % ### Fin del Algoritmo ###
76
77    % Estimar el tiempo de ejecucion
78    OCT_Estimacion();
79  endfunction
```

B.3. Algoritmo implementado en MPI con comunicaciones punto a punto

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4   #include "mpi.h"
5
6   #define root 0
7
8   typedef struct
9   {
10    unsigned int filas;
11    unsigned int offset;
12    unsigned int inicio;
13  }datosnodos;
14
15  int  fnc_nfilas(int n_filas, int nodo, int extras)
16  {
17    return (nodo < extras) ? n_filas + 1 : n_filas;
18  }
19
20  int fnc_offset(int n_filas, int nodo, int extras)
```

```
21  {
22    unsigned int offset;
23
24    if(extras)
25    {
26    if(nodo)
27    {
28      if(nodo < extras)
29        offset = (nodo * n_filas) + nodo;
30      else
31        offset = (nodo * n_filas) + extras;
32    }
33    else
34      offset = 0;
35    }
36    else
37      offset = nodo * n_filas;
38
39    return offset;
40  }
41
42  int main(int argc, char *argv[])
43  {
44    int i, j, n, k, l, m, p;
45    int total_elems, muestras;
46    int rank, nprocs;
47    int fila_ac, fila_b, prom_filas;
48    float *A, *B, *C;
49    int extra, imax, imin;
50
51    double ti_comm, tf_comm, tp_comm = 0, *m_comm;
52    double ti_comp, tf_comp, tp_comp = 0, *m_comp;
53    double tp_exec = 0;
54    double *m_exec;
55
56    datosnodos *nodos;
57    MPI_Status status;
58    if(argc != 3)
59    {
60      printf("USO: program tama o_matrix muestras\n");
61      return 1;
62    }
63
64    n = atoi(argv[1]);
65    muestras = atoi(argv[2]);
66    m_exec = (double*)calloc(muestras + 2, sizeof(double));
```

```
67      m_comm = (double*)calloc(muestras + 2, sizeof(double));
68      m_comp = (double*)calloc(muestras + 2, sizeof(double));
69
70      MPI_Init(&argc, &argv);
71      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
72      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
73
74      total_elems = n*n;
75      prom_filas = n/nprocs;
76      extra = n % nprocs;
77
78      B = (float*)calloc(total_elems, sizeof(float));
79
80      if(rank == root)
81        nodos = (datosnodos*)calloc(nprocs, sizeof(datosnodos));
82      else
83        nodos = (datosnodos*)calloc(1, sizeof(datosnodos));
84
85      nodos[0].filas = fnc_nfilas(prom_filas, rank, extra);
86
87      /*
88       * Inicializaci n tanto del maestro como los esclavos
89      */
90      if(rank == root)
91      {
92        srand(time(NULL));
93        A = (float*)calloc(total_elems, sizeof(float));
94        C = (float*)calloc(total_elems, sizeof(float));
95
96        fila_ac = 0;
97        for(i=0; i < n; i++)
98        {
99          for(j=0, k = fila_ac; j < n; j++, k++)
100         {
101           A[k] = rand() % 10 + 1;
102           B[k] = rand() % 10 + 1;
103         }
104         fila_ac += n;
105       }
106
107       for(i=1; i < nprocs; i++)
108       {
109         nodos[i].filas = fnc_nfilas(prom_filas, i, extra);
110         nodos[i].offset = fnc_offset(prom_filas, i, extra);
111         nodos[i].inicio = nodos[i].offset * n;
112       }
```

```
113
114       fila_ac = nodos[1].inicio;
115     }
116     else
117     {
118       C = (float*)calloc(nodos[0].filas * n, sizeof(float));
119       A = (float*)calloc(nodos[0].filas * n, sizeof(float));
120     }
121     // Fin de la inicializaci n
122
123     // Inicio de comunicacion
124     MPI_Barrier(MPI_COMM_WORLD);
125     if(rank == root)
126     {
127       for(i=1; i < nprocs; i++)
128       {
129         for(j=0; j < nodos[i].filas; j++)
130         {
131           MPI_Send(&A[fila_ac], n, MPI_FLOAT, i, 1,
                MPI_COMM_WORLD);
132           fila_ac += n;
133         }
134         fila_b = 0;
135         for(j=0; j < n; j++)
136         {
137           MPI_Send(&B[fila_b], n, MPI_FLOAT, i, 1, MPI_COMM_WORLD
                );
138           fila_b += n;
139         }
140       }
141     }
142     else
143     {
144       fila_ac = 0;
145       for(i=0; i < nodos[0].filas; i++)
146       {
147         MPI_Recv(&A[fila_ac], n, MPI_FLOAT, root, 1,
              MPI_COMM_WORLD, &status);
148         fila_ac += n;
149       }
150
151       fila_b = 0;
152       for(j=0; j < n; j++)
153       {
154         MPI_Recv(&B[fila_b], n, MPI_FLOAT, root, 1,
              MPI_COMM_WORLD, &status);
```

```
155         fila_b += n;
156       }
157     }
158     // Fin de comunicacion
159
160     // COMPUTO
161     fila_ac = 0;
162     //Se determina cuantas filas se van a procesar
163     for(i=0; i < nodos[0].filas; i++)
164     {
165       //Se procesa cada fila
166       for(j = fila_ac, l=0; j < fila_ac + n; j++, l++)
167       {
168         //Se realiza la multiplicaci n con la correspondiente
              fila y columna
169         for(k=l, p = fila_ac; k < total_elems; k += n, p++)
170           C[j] += A[p] * B[k];
171       }
172       fila_ac += n;
173     }
174     // Fin de c mputo
175     // Envio y recepci n de resultados
176     if(rank != root)
177     {
178       fila_ac = 0;
179       for(i=0; i < nodos[0].filas; i++)
180       {
181         MPI_Send(&C[fila_ac], n, MPI_FLOAT, root, 1,
              MPI_COMM_WORLD);
182         fila_ac += n;
183       }
184     }
185     else
186     {
187       fila_ac = nodos[1].inicio;
188       for(i=1; i < nprocs; i++)
189       {
190         for(j=0; j < nodos[i].filas; j++)
191         {
192           MPI_Recv(&C[fila_ac], n, MPI_FLOAT, i, 1,
                MPI_COMM_WORLD, &status);
193           fila_ac += n;
194         }
195       }
196     }
197
```

```
198     MPI_Finalize();
199     return 0;
200   }
```

B.4. Algoritmo implementado con la biblioteca desarrollada con comunicaciones punto a punto

```
1   function [Texec, Tcomp, Tcomm] = mmt_fc_nov(Datos, P, B, L, Lsw
        , Tbas)
2     % Variables propias de la biblioteca
3     global OCT_Datos; % Datos a transmitir
4     global OCT_Procs; % N mero de procesadores
5     global OCT_Bandw; % Ancho de banda
6     global OCT_Latencia; % Latencia de la red
7     global OCT_Lsw; % Latencia del switch
8     global OCT_Tbas; % Tiempo b sico de la aplicaci n
9     global OCT_Ttrans; % Tiempo de transmision
10
11    % Tiempos
12    global Texec; % Tiempo estimado de ejecuci n
13    global Tcomm; % Tiempo de comunicaci n
14    global Tcomp; % Tiempo de c mputo
15
16    % Utileria
17    global OCT_P_iters; % N mero de iteracciones
18
19    % Variables de MPI
20    global MPIO_CHAR;
21    global MPIO_SHORT;
22    global MPIO_INT;
23    global MPIO_FLOAT;
24    global MPIO_DOUBLE;
25    global MPIO_LONG;
26
27    % Inicializar valores de la biblioteca
28    OCT_Ivar(Datos, P, B, L, Lsw, Tbas);
29
30    Tcomm = 0;
31    Tcomp = 0;
32    Texec = 0;
33    OCT_P_iters = 0;
34    OCT_Tbas = Tbas;
```

```
35
36      % ### Inicio del Algoritmo ###
37
38      % Elementos que para cada proceso
39      prom_elems = OCT_Datos / OCT_Procs;
40
41      % Elementos extras
42      extras = mod(OCT_Datos, OCT_Procs);
43
44      buf = 0;
45      C = 0;
46      status = 0;
47      MPI_COMM_WORLD = 0;
48
49      for i=1:OCT_Procs
50        % Distribuye los elementos extras entre los procesos
51        [elems_proc(i)] = OCT_Elems_extras(prom_elems, extras);
52      endfor
53
54      % COMUNICACION
55      MPIO_Barrier(MPI_COMM_WORLD);
56      % Distribucion de A y B (Desde root)
57      for i=2:OCT_Procs
58        for j=1:elems_proc(i)
59          MPIO_Send(buf, OCT_Datos, MPIO_FLOAT, i, 1,
              MPI_COMM_WORLD);
60        endfor
61
62        for j=1:OCT_Datos
63          MPIO_Send(buf, OCT_Datos, MPIO_FLOAT, i, 1,
              MPI_COMM_WORLD);
64        endfor
65      endfor
66
67      % COMPUTO
68      for i=1:elems_proc(1) % Filas a procesar
69        for j=1:OCT_Datos % Filas
70          for k=1:OCT_Datos % Columna
71            OCT_Procesamiento();
72          endfor
73        endfor
74      endfor
75
76      % COMUNICACION
77      % Recepcion de C (En root)
78      for i=2:OCT_Procs
```

```
79      for j=1:elems_proc(i)
80        MPIO_Recv(C, OCT_Datos, MPIO_FLOAT, i, 1, MPI_COMM_WORLD,
              status);
81      endfor
82    endfor
83
84    % ### Fin del Algoritmo ###
85
86    % Estimar el tiempo de ejecucion
87    OCT_Estimacion();
88  endfunction
```

Anexo C: Proceso de caracterización de la red

En este anexo se describe el proceso para determinar la latencia de la red, así como la latencia del *switch*, que ayudan a estimar el tiempo de comunicación en la red.

C.1. Latencia de la red

Para obtener la latencia de la red se desarrolló una aplicación que es ejecutada en un par de nodos, donde el nodo origen (también como nodo raíz, debido a que es el nodo cero) transmite y recibe un byte desde y hacia un nodo destino (todos los otros nodos diferentes a raíz) y el nodo destino transmite y recibe un byte desde y hacia un nodo origen, como se ilustra en la Figura C.1. Este proceso se puede repetir cuantas veces sea necesario para obtener una mejor latencia de la red.

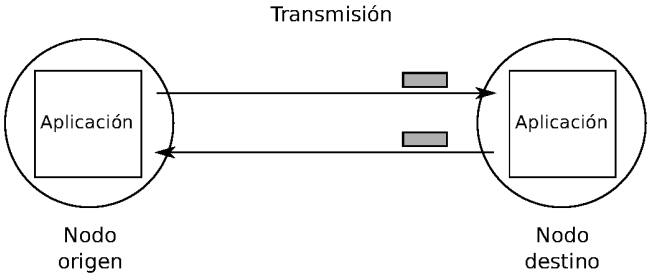

Figura C.1: Caracterización de la latencia de la red

C.2. Latencia del *switch*

Para obtener la latencia del *switch* se desarrolló una aplicación, donde un par de nodos intercambian mensajes, el nodo origen transmite y recibe un byte desde y hacia un nodo destino y el nodo destino transmite y recibe un byte desde y hacia un nodo origen. El intercambio de mensajes ocurre en dos etapas, la primera etapa el intercambio de los mensajes se lleva acabo mediante un *switch* (ver Figura C.2 inciso A) y en la segunda etapa el intercambio se realiza sin el uso de algún *switch* dado (ver Figura C.2 inciso B). Una vez que se obtienen los tiempos de comunicación de las dos etapas, se determina la diferencia y podemos obtener la latencia del *switch*, es decir el tiempo que tarda en procesar los paquetes.

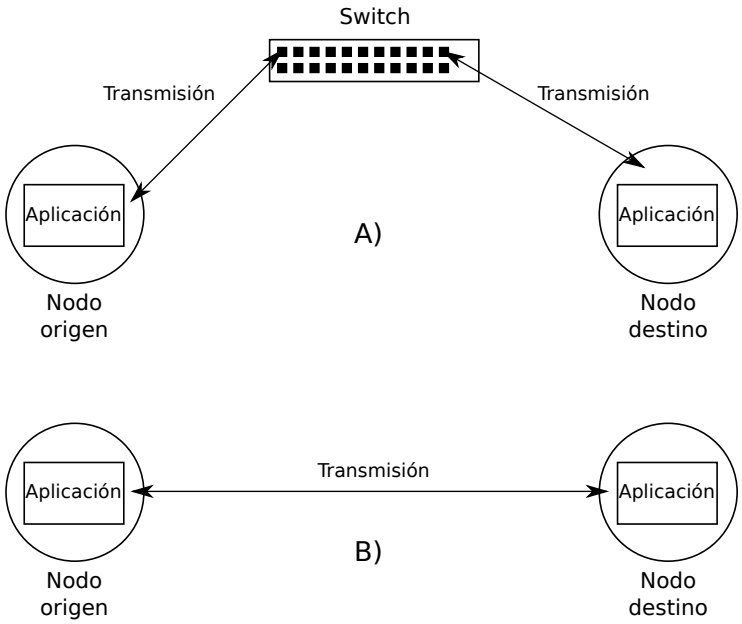

Figura C.2: Caracterización de la latencia del *switch*