Estudio del rendimiento y la escalabilidad de aplicaciones MPI en entornos distribuidos utilizando SIMCAN

Trabajo de Fin de Grado

Grado en Ingeniería de Computadores

Facultad de Informática

Universidad Complutense de Madrid

2017/2018

Bryan Raúl Vaca Vargas

Dirigido por:

Alberto Nuñez Covarrubias

Agradecimientos

A mi familia, por apoyarme en estos últimos años y sobre todo en estos últimos meses en los que la presión y la exigencia han sido notables. Sin ellos esto no habría sido posible.

A mi pareja, la cual me ha ayudado a afrontar todos los baches que han aparecido al lo largo del desarrollo de este proyecto.

A mi director Alberto, porque a pesar de todos los inconvenientes surgidos, ha estado siempre para ayudarme y guiarme hacia la dirección adecuada.

A mis amigos, porque ellos también han vivido a mi lado los momentos difíciles del desarrollo de este proyecto.

Muchas gracias a todos.

Resumen

Este proyecto tiene como objetivo representar el comportamiento de aplicaciones MPI en entornos distribuidos simulados. En concreto, los entornos se representarán en la plataforma de simulación SIMCAN, una plataforma para modelar y simular entornos y aplicaciones distribuidas.

Las aplicaciones MPI son programas que trasmiten mensajes entre varios procesos las cuales pueden ser ejecutados en una o varias máquinas, consiguiendo ejecutar y procesar datos de forma paralela.

Para lograr el objetivo propuesto, las aplicaciones MPI se procesan en 3 Fases principales. La primera Fase consiste en el desarrollo de una biblioteca en lenguaje C con la que se genera un fichero de registro que contiene todas las llamadas MPI y de E/S ejecutadas por cada proceso. La segunda Fase consiste en integrar en SIMCAN el registro obtenido en la Fase anterior. Esto permitirá simular la aplicación MPI en distintos tipos de entornos distribuidos. Como resultado se genera un nuevo fichero de registro. La tercera Fase procesa uno o varios ficheros generados en las Fases anteriores y representa gráficamente la ejecución de la aplicación MPI. Con la representación obtenida es posible estudiar de forma detallada las ejecuciones de los programas MPI. Es importante destacar que así se puede comparar el comportamiento de la aplicación, ejecutada en entornos reales, con la ejecución de la misma aplicación en entornos simulados utilizando distintas configuraciones.

Palabras clave: Sistemas distribuidos, Computación distribuida, MPI, SIMCAN, Ejecución paralela, Simulación de sistemas distribuidos.

Abstract

This project aims to represent the behaviour of MPI applications in distributed simulated environments. Specifically, the modeled environments will be represented in the SIMCAN simulation platform.

MPI applications are programs that transmit messages among a number of processes, which can be executed in one or various machines. This enables the application to execute and process data in parallel.
SIMCAN is a platform used for modelling and simulating distributed systems and applications.

In order to achieve this objective, the MPI applications will be process in 3 main phases. The first phase, consists in developing a library in language C. This will generate a register file that contains all MPI and E/S calls executed by each process. The second phase consists in integrating the register file generated in the first phase into SIMCAN. This will enable the simulation of the MPI application in different types of distributed environments. As a result, this will generate a new register file. The third phase will process one or more generated files from the previous phases and will then represent graphically the execution of the MPI applications. The obtained representation makes it possible to study the executions of the MPI programs in detail. It is important to highlight that this enables the comparison of the behaviour of the executed application in real environments with the execution of the same application in simulated environments.

Keywords: Distributed Systems, Distributed Computing, MPI, SIMCAN, Parallel computing, Simulation of distributed systems.

Indice

1. Introducción

En este capítulo se describen los objetivos principales y secundarios del proyecto, así como el alcance y motivación. Además, se detalla el plan de trabajo establecido para la realización del mismo.

1.1. Objetivos

El objetivo principal de este proyecto es representar gráficamente el comportamiento de aplicaciones MPI utilizando entornos distribuidos.

Estos entornos se construirán con el simulador SIMCAN, el cual permite modelar distintos tipos de entornos distribuidos y ejecutar distintas aplicaciones software. En este caso, el tipo de aplicación simulada será MPI.

El objetivo principal, a su vez, puede descomponerse en varios objetivos secundarios.

- Desarrollar una biblioteca que capture las llamadas a funciones MPI y E/S. Esta biblioteca obtiene los tiempos e información necesaria de cada llamada, para luego ser volcada a un fichero único del que se pueda extraer la información y ejecutarla en el simulador SIMCAN.
- Ejecutar en SIMCAN la traza obtenida. Una vez se ha obtenido la traza de la aplicación MPI a partir de su ejecución en un entorno real, el simulador ejecuta la aplicación en un entorno, previamente modelado, y genera un nuevo fichero con los resultados de la simulación.
- Representar de forma gráfica los resultados. Estos resultados son obtenidos a partir de la ejecución de la aplicación MPI y de su simulación en SIMCAN. Esta representación permite obtener, visualizar y estudiar los resultados detalladamente.

1.2. Alcance y motivación

El alcance de este proyecto es meramente académico, es decir, se empleará principalmente para el estudio de aplicaciones MPI de una manera más accesible a los estudiantes.

En este aspecto radica el principal problema, y consecuente motivación, ya que en los laboratorios disponibles de la Facultad no es posible, por motivos de seguridad, ejecutar aplicaciones MPI usando varios ordenadores. De manera que ya no es necesario recurrir al uso de estos sistemas para poder analizar los resultados sino que, utilizando herramientas de simulación, como SIMCAN, podemos representar un amplio abanico de configuraciones hardware, ejecutar en ella cualquier aplicación software y obtener resultados.

Por tanto, la motivación principal para desarrollar este proyecto es el ahorro de costes y de tiempo a la hora de analizar aplicaciones MPI en entornos distribuidos.

1.3. Plan de trabajo

El desarrollo del proyecto esta formado por en varias etapas.

- La primera etapa, previa al desarrollo, consiste en el análisis del problema y la planificación del trabajo a realizar para poder llevar a cabo el proyecto. En esta etapa se planifican tiempos de desarrollo y objetivos de cada Fase. La figura 1.1 detalla el tiempo estimado en cada etapa. Como podemos observar, el análisis del problema requiere aproximadamente una semana de trabajo y por otra parte, la planificación y establecer el plan de trabajo conllevan, ambas, una semana. Una vez comprendidos los objetivos del proyecto, se pasa a la etapa del desarrollo, la cual se divide en 3 Fases principales.
- La segunda etapa, la cual se corresponde con el primer objetivo secundario, consiste en desarrollar una biblioteca que registre las llamadas invocadas por la aplicación. En la figura 1.1 y 1.2 se observa el transcurso de esta etapa en la que se dedica, una semana al diseño, dos semanas al desarrollo, una semana para pruebas y documentación, y varias semanas para adaptaciones necesarias, que surgen al desarrollar los siguientes objetivos. El desarrollo de esta biblioteca permite analizar los requisitos tanto para la tercera como para la cuarta etapa.
- La tercera etapa, la cual se corresponde al segundo objetivo secundario, consiste en conocer en detalle la implementación de SIMCAN y adaptar este simulador con el objetivo de simular las aplicaciones MPI ejecutadas en la segunda etapa. En la figura 1.2 se observa que la duración de esta etapa es de más de 3 semanas en las que media semana se dedica a conocer la aplicación, más de una semana al desarrollo y una semana para pruebas. La documentación de esta etapa se lleva a cabo en la última semana del proyecto.
- La cuarta etapa, la cual se corresponde al tercer objetivo secundario, consiste en el desarrollo de una aplicación en la que se unifican todas las etapas desarrolladas, necesarias para llegar a obtener representaciones gráficas a partir de la ejecución de una aplicación MPI en un entorno simulado en SIMCAN. Se observa en la figura 1.2 que la duración de esta etapa es de cinco semanas. En la primera semana se analiza los requisitos y diseña la aplicación, las siguientes dos semanas se dedican al desarrollo de la aplicación, una semana para pruebas y adaptaciones, y la última semana para la documentación.
- La última etapa, consiste en la realización de experimentos con los que poder demostrar la usabilidad de la aplicación desarrollada en la cuarta etapa. Se observa en la figura 1.2 que la duración de esta etapa es de una semana.

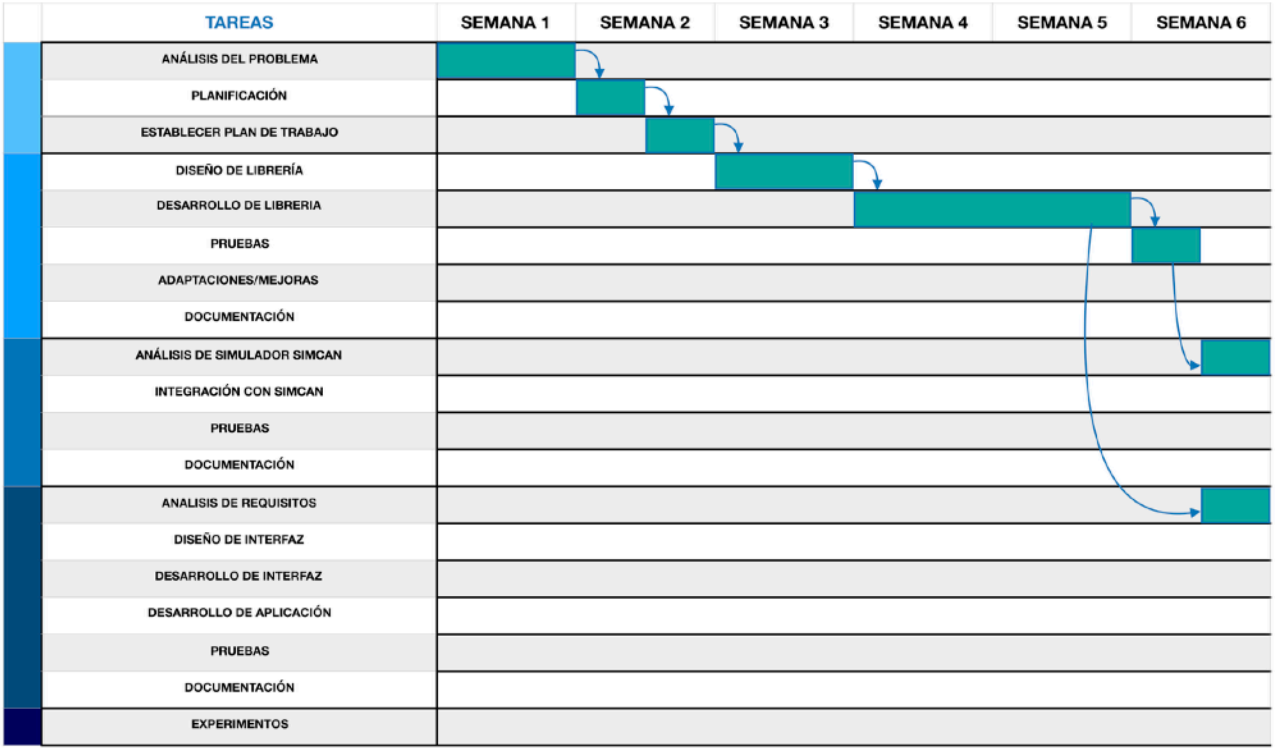

Figura 1.1 Planificación del proyecto

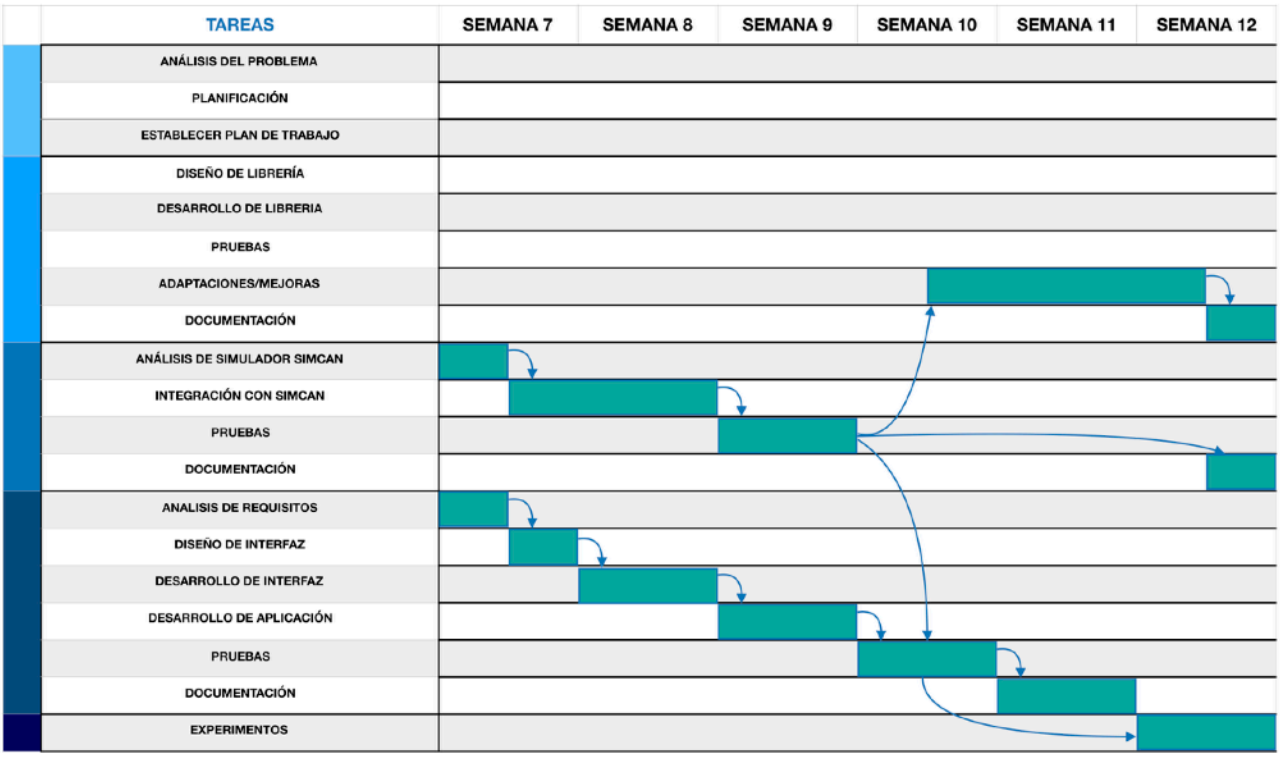

Figura 1.2 Planificacion del proyecto

1.4.Estructura

Este documento se divide en los siguientes capítulos:

- Capítulo 1. Este capítulo expone de manera introductoria los objetivos, alcance y motivación del el proyecto.
- Capítulo 3. El segundo capítulo expone el estado actual de las tecnologías utilizadas en el proyecto y las tecnologías similares a las desarrolladas en el mismo.
- Capítulo 4. El tercer capítulo detalla el proceso de desarrollo del proyecto, explicando cada Fase en detalle.
- Capítulo 4. El cuarto capítulo muestra los experimentos llevados a cabo.
- Capítulo 5 . Este capítulo describe las conclusiones alcanzadas una vez finalizado el desarrollo del proyecto. Además expone el posible trabajo futuro del proyecto.

1. Introduction

This chapter describes the main and secondary objectives of the project, as well as the scope and motivation. In addition, the work plan established to carry it out is detailed.

1.1. Goals

The main objective of this project is to graphically represent the behavior of MPI applications using distributed environments.

These environments will be built using the SIMCAN simulator, which allows modeling different types of distributed environments and running different software applications.

The main objective, in turn, can be divided in several secondary objectives.

- Develop a library that captures calls to MPI and I/O functions. This library obtains the necessary timestamps and information for each call, and then it is written into a single file from which the information can be extracted and executed in the SIMCAN simulator.
- Execute the obtained trace in SIMCAN. Once the trace of the MPI application has been obtained from its execution in a real environment, the simulator executes the application in a simulated environment, which has been previously modeled, and generates a new file with the results of the simulation.
- Graphically represent the results. These results are obtained from the execution of the MPI application and its simulation in SIMCAN. This representation allows to obtain, visualize and study the results in detail.

1.2. Scope y motivation

The scope of this project is purely academic, that is, it will be used mainly to study MPI applications in a more accessible way to students.

In this aspect lies the main problem, and consequent motivation, since in the available laboratories of the university it is not possible, for security reasons, to execute MPI applications using several computers. However, using simulation tools, such as SIMCAN, we can represent a wide range of hardware configurations, execute any software application and obtain results.

Therefore, the main motivation to develop this project is the cost and time savings when analyzing MPI applications in distributed environments.

1.3. Workplan

The development of the project is consist of up of several stages.

- The first stage, prior to development, consists of analyzing the problem and planning the work to be carried out the project. In this stage, development times and objectives of each phase are planned. Figure 1.1 details the estimated time in each stage. As we can see, the

analysis of the problem, the planning and establishing the work plan, requires about a week of work.

- The second stage, which corresponds to the first secondary objective, is to develop a library that records the calls invoked by the application. Figure 1.1 and 1.2 show the course of this stage in which one week is dedicated to design, two weeks to development, one week for tests and documentation, and several weeks for necessary adaptations, which arise when developing the following objectives . The development of this library allows analyzing the requirements for both the third and the fourth stage.
- The third stage, which corresponds to the second secondary objective, consists of knowing in detail the SIMCAN implementation and adapting this simulator with the objective of simulating the MPI applications executed in the second stage. Figure 1.2 shows that the duration of this stage is more than 3 weeks in which half a week is dedicated to knowing the application, more than a week to development and a week for tests. The documentation of this stage is carried out in the last week of the project.
- The fourth stage, which corresponds to the third secondary objective, consists in the development of an application in which all the developed stages are unified this is necessary to produce a graphic representations from the execution of an MPI application in a simulated environment in SIMCAN. It is observed in figure 1.2 that the duration of this stage is five weeks. In the first week the requirements are analyzed and the application is designed, the next two weeks are dedicated to the development of the application, a week for tests and adaptations, and the last week for documentation.
- The last stage consists in carrying out experiments with which to demonstrate the usability of the application developed in the fourth stage. It is observed in figure 1.2 that the duration of this stage is one week.

1.4.Structure

This document is divided into the following chapters:

- Chapter 1. The chapter introduces the objectives, scope and motivation.
- Chapter 2. The second chapter exposes the current state of the technologies used in the project and the technologies similar to those developed in the project.
- Chapter 3. The third chapter details the process of project development, explaining each phase in detail.
- Chapter 4. The fourth chapter shows the experiments carried out.
- Chapter 5 . The fifth chapter describes the conclusions reached after the development of the project and exposes the possible future work of the project.

2. Estado del Arte

En este capítulo se definen las principales tecnologías usadas en este proyecto así como tecnologías existentes en el ámbito en el cual se enmarca este proyecto.

2.1. MPI

La necesidad de mejorar los tiempos de ejecución de las aplicaciones MPI o repartir tareas para evitar sobrecargar la memoria en un computador, genera la necesidad de paralelizar tareas, tanto a nivel software como a nivel hardware.

El objetivo principal de la programación paralela es conseguir una carga equitativa entre los procesos, minimizar la comunicación entre ellos y alternar comunicación y ejecución.

Uno de los métodos utilizados para conseguir ese paralelismo es el uso de la biblioteca MPI (Message Passing Interface). En esencia, esta biblioteca gestiona la transferencia de datos entre varios procesos.

MPI fue desarrollado por el MPI_FORUM, el cual estaba compuesto por aproximadamente sesenta personas y cuarenta organizaciones.

En 1994 se define el MPI-1 Standar, el cual especifica las subrutinas y funciones que se usarían en Fortran 77 y en C, respectivamente. Lo que se buscaba principalmente era poder compilar y ejecutar estos programas desde cualquier plataforma.

Seguidamente se definió MPI-2 Standar. Lo que hace este estándar es incluir herramientas para la E/S paralela, la gestión de procesos dinámicos y compatibilidad con C++ y Fortran 90.

Para manejar el paralelismo disponemos de dos tipos de arquitecturas. Por una parte la arquitectura de memoria distribuida y, por otra, la arquitectura de memoria compartida.

La arquitectura de memoria distribuida se compone de varios computadores (nodos) que trabajan juntos para resolver un problema. Cada nodo tiene acceso a su propia memoria de tal forma que cuando es necesario acceder a información alojada en memoria de otros nodos, se establece comunicación a través de la red e intercambian mensajes.

En una arquitectura de memoria compartida, múltiples procesadores acceden a la memoria a través de un BUS de alta velocidad. El problema de esta arquitectura son los cuellos de botella en el acceso a la memoria, además del número limitado de unidades de cómputo.

Existen también arquitecturas como la DSM (Distributed Shared Memory) que implementa el modelo de memoria compartida, pero en sistemas distribuidos.

Los objetivos principales de MPI son proveer portabilidad y eficiencia en todas las plataformas.

La biblioteca MPI se divide en cuatro tipo de llamadas:

- Llamadas usadas para inicializar, gestionar y terminar comunicaciones. Se encargan de inicializar comunicaciones, identificar el número de procesos usados y el proceso concreto que está llamando a las funcionalidades disponibles.
- Llamadas usadas para comunicar entre pares de procesos. Operaciones de envío y recepción de mensajes.

- Llamadas encargadas de gestionar las comunicaciones. Proveen operaciones de sincronización, de tratamiento de estructuras de datos complejas y operaciones de comunicación entre grupos de procesos.
- Llamadas usadas para crear cualquier tipo de datos, con lo que se consigue poder hacer uso de estructuras de datos propias.

Las distribuciones más recientes se encuentran disponible en:

- https://www.open-mpi.org/
- http://lam-mpi.miscellaneousmirror.org
- https://www.mpich.org

2.2. MPE

MPE es una aplicación destinada a programadores de aplicaciones MPI, la cual proporciona distintas herramientas con las que se recopila el comportamiento del programa MPI ejecutado. A partir de esta ejecución se construye un registro para poder generar la visualización correspondiente. Esta aplicación es compatible con las implementación MPICH y OpenMPI.

Existen varias versiones de MPE, la primera fue desarrollada para sistemas compatibles con X-Windows, un terminal gráfico virtual. Tras esta primera versión, debido a su dificultad de mantenimiento y escalabilidad, se reescribe en lenguaje de Scripting TeL utilizando paquetes gráficos Tk. El inconveniente de esta versión era la lentitud a la hora de procesar grandes ejecuciones. Por ello se vuelve a desarrollar la parte gráfica de la aplicación, esta vez en C, usando la interfaz C de Tel. Este cambio mejoró el rendimiento, pero generó dependencia con una parte inestable de Tel. Este último problema conllevó a una nueva mejora: el desarrollo de la aplicación gráfica en Java. Existen 4 versiones de esta implementación. Estas ultimas versiones mejoran notablemente el rendimiento, portabilidad y mantenimiento y añadiendo nuevas funcionalidades gráficas.

La distribución mas reciente de MPE está disponible en: http://www.mcs.anl.gov/research/projects/perfvis/download/index.htm

2.3. Simulación de procesos distribuidos en entornos virtuales

Actualmente existen varias distribuciones software que simulan ejecuciones de procesos distribuidos. Es el caso de SIMGRID y SIMCAN, entre otras.

2.3.1. SIMCAN

SIMCAN es una simulador de aplicaciones y entornos distribuidos. Está programado en C++ y utiliza Omnet INE. Este simulador permite modelar entornos distribuidos. La primera versión de

SIMCAN se desarrollo con fines de investigación sin embargo, la aplicación se ha adaptado para su uso en la enseñanza.

SIMCAN dispone de distintos tipos de componentes con distintas características, CPUs, Memorias RAM, Discos Duros, Redes. La combinación de estos componentes dan lugar a una gran cantidad de sistemas modelos.

El objetivo principal de SIMCAN es la reutilización de componentes. Por otra parte se busca un alto nivel de flexibilidad y escalabilidad, facilita el desarrollo de aplicaciones distribuidas al proporcionar APIs intuitivas y por último permite investigar el rendimiento de las aplicaciones distribuidas con distintos tipos de configuraciones.

Para modelar un sistema distribuido es necesario modelar tanto la parte Software como la parte Hardware.

En el apartado Hardware encontramos las máquinas físicas, llamados nodos que son los encargados de realizar tareas de computo y del almacenamiento. Estos nodos en conjunto forman un Racks.

En el apartado Software se establecen protocolos de comunicación, sistemas de ficheros, y las aplicaciones distribuidas y paralelas MPI. Es posible modelar aplicaciones API similares a POSIX y aplicaciones distribuidas MPI.

La versión más reciente de SIMCAN está disponible en:

- http://antares.sip.ucm.es/cana/simcan/index.html

2.3.2.SIMGRID

SIMGRID es un conjunto de herramientas utilizado para estudiar el comportamiento de sistemas distribuidos. Destaca por su portabilidad, escalabilidad y rendimiento.

Funciona en cualquier sistema operativo y es capaz de simular aplicaciones escritas en C, C++ y Java. Produce datos que pueden ser visualizados gráficamente.

Se trata de una aplicación ampliamente utilizada, bajo licencia libre, y que es utilizada por el CERN (Organización Europea para la Investigación Nuclear) para mejorar los algoritmos de gestión de dato.

Sus desarrolladores denominan SIMGRID como una biblioteca, con la que es posible interactuar a través de programas que hacen uso de las funcionalidades disponibles en la biblioteca.

Se compone principalmente de una aplicación desarrollada por el usuario, una plataforma virtual que describe en formato xml, un sistema distribuido, una descripción de implementación detallando como se va a ejecutar la aplicación en cada componente de la arquitectura descrita en la plataforma virtual y por último modelos de plataforma en los que es necesario especificar cómo reacciona el sistema distribuido a las acciones llamadas por la aplicación.

La combinación de todos estos componentes dan lugar a las simulaciones, y estas generan resultados que permiten comparar aplicaciones, depurar aplicaciones reales o para diseñar una sistema distribuido ideal para una aplicación específica.

La versión más reciente de SIMGRID está disponible en:

- http://simgrid.gforge.inria.fr/

3. Estructura del proyecto

En este capítulo se describe en detalle la propuesta de este Trabajo de Fin de Grado, el transcurso del desarrollo y los detalles de cada Fase.

3.1. Introducción

La propuesta de este proyecto consiste en 3 Fases de desarrollo. Cada Fase se centra en una parte software.

En la primera Fase se lleva a cabo el desarrollo de una biblioteca, encargada de registrar un subconjunto de llamadas MPI y llamadas E/S de la biblioteca estándar de C. Para esta Fase se hace uso de lenguaje C. Como vemos en la figura 3.1, la aplicación original se transforma para invocar a las funciones de la biblioteca creada. Una vez transformada, se compila enlazándola con la biblioteca, lo que genera un ejecutable. Al ejecutar la aplicación, esta genera varios ficheros, uno por cada proceso ejecutado, que son integrados en un mismo fichero. Esta traza generada puede ser introducido en SIMCAN para simularla en un entorno distribuido previamente modelado o puede ser añadida a la aplicación desarrollada en la Fase 3 y ser representada gráficamente.

En la segunda Fase se desarrolla una aplicación en SIMCAN. Como vemos en la figura 3.1, principalmente, esta aplicación lee una traza de un fichero (generado en la Fase 1) y la reproduce en el simulador. La ejecución de esta aplicación dará lugar a un fichero de registro que posteriormente será leído y tratado en la Fase 3.

En la tercera Fase se desarrolla la aplicación gráfica. Básicamente, esta aplicación representará gráficamente el comportamiento de las aplicaciones MPI. Como vemos en la figura 3.1 la aplicación puede representar tanto los resultados de la Fase 1, como los resultados obtenidos de la Fase 2. Para desarrollar esta aplicación se hace uso de diversas tecnologías como Netbeans, Eclipse, Java, Bash, Maven y Json.

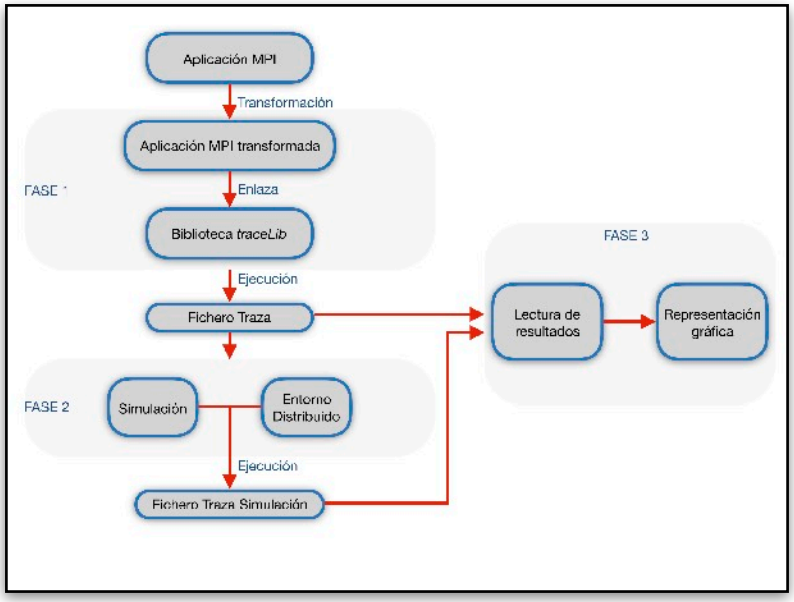

Figura 3.1 Estructura del proyecto

3.2. Fase 1: Biblioteca para generar trazas de aplicaciones MPI

La primera Fase consiste en desarrollar una biblioteca *traceLib,* encargada de registrar las llamadas MPI y de E/S invocadas por cualquier programa que haga uso de MPI. De esta forma se genera un fichero en el que se registra la información necesaria para poder reproducir el comportamiento de la aplicación en SIMCAN.
La importancia de esta primera Fase radica en la necesidad de obtener datos relevantes y completos de la ejecución de una aplicación MPI, teniendo en cuenta que se busca poder estudiar detalladamente el comportamiento de estas ejecuciones.

Con el fin de facilitar la descripción de las Fases presentadas en este capítulo, detallamos el significado de algunos de los términos utilizados. Denominamos *rank* al ID del proceso que invoca las llamadas, este puede ser *worker* o *master*. Denominamos *worker* a un proceso que recibe mensajes de un *master* para realizar un determinado trabajo. Denominamos *master* al proceso principal encargado de distribuir trabajos entre los procesos *workers.*

Para desarrollar la biblioteca propuesta, se ha utilizado una estructura homogénea, a pesar de que cada llamada tenga signaturas diferentes. Es el caso, por ejemplo, de las llamadas más simples como pueden ser :

```
open(const char *path, int oflag)
```

A otras más complejas como pueden ser:

```
MPI_Recv( void* mensaje, int count, MPI_Datatype tipo_de_dato , int emisor, int etiqueta, MPI_Comm comunicador,
MPI_Status* estado).
```

Por ello, todas las llamadas contempladas en la biblioteca desarrollada tendrán la siguiente sintaxis:

```
typeReturn nameFunction_trace( [parámetros propios de la llamada], const int myRank).
```

donde:

- *typeReturn es el t*ipo de valor devuelto por la función, siempre será el mismo que el de la función invocada original.
- *nameFunction_trace* es el nombre de la llamada de la biblioteca. Se compone por el nombre de la llamada original y la adición del sufijo “_trace”.
- *[parámetros propios de la llamada]* sos parámetros serán los mismos que los usados en la llamada original invocada.
- *myRank*: Rank del proceso que invoca la llamada.

A continuación detallaremos las llamadas implementadas en la biblioteca.

3.2.1.Funciones de gestión de llamadas

En esta sección se describen las funciones internas de la biblioteca que gestionan las llamadas de los procesos a cada una de las funciones MPI y E/S.

```
void addTaskInfo(char *name, struct timeval *tvIni, struct timeval *tvEnd,
                 int rankSource, int rankDest, int dataSize);


void addTaskFileInfo(const char *name, const char *path,
                 struct timeval *tvIni, struct timeval *tvEnd,
                 int my_rank, int rank_dst_src, int dataSize,
                 int offset);
```

Estas funciones son las encargadas de calcular el inicio y el final de una tarea. Por cada llamada invocada por un proceso se generan las partes de la tarea (inicio y fin) para posteriormente añadirlas al listado de tareas que se vuelcan al fichero de registro. Cada vez que se invoca una llamada de la biblioteca *traceLib,* se llama a una de las funciones, proporcionándole los parámetros necesarios, donde:

- *name* es el nombre de la función invocada.
- *path* es el nombre del fichero creado, abierto, leído, escrito o cerrado. Solo presente en la función *addTaskFileInfo.*
- *tvIni* es la marca de tiempo inicial. Instante previo a la invocación de la llamada original.
- *tvEnd* es la marca de tiempo final. Instante posterior a la invocación de la llamada original.
- *rankSource* es el identificador del proceso emisor y/o del proceso que invoca la llamada.
- *rankDest* es el identificador del proceso receptor y/o del proceso que invoca la llamada.
- *dataSize* es el tamaño de datos leídos, escritos o transmitidos.
- *offset* es el desplazamiento en fichero en operaciones de lectura o escritura. Solo presente en la función *addTaskFileInfo.*

```
void traceInit();
```

La función *treaceInit*, se ejecuta al invocar a la función *MPI_Init_trace*, se encarga de inicializar y reservar espacio para las estructuras de datos relativas a las *tareas*, inicializar el contador de Ids para las *tareas* y registrar la marca de tiempo inicial relativa al Worker que lo ejecuta.

```
void traceEnd();
```

La función *traceEnd* se ejecuta al invocar a la función *MPI_Finalize_trace* y se encarga de volcar al fichero asociado a su proceso, todas las llamadas capturadas previamente. Cada proceso invocará a *MPI_Finalize_trace* generando un fichero de registro asociado a su rank.

```
void writeListTaskInfo(FILE *file);
```

La función *writeListTaskInfo* es invocada desde la función *traceEnd* y se encarga de recorrer la lista de tareas y volcarlas al fichero de registro, donde:

- *file* es el puntero al descriptor de fichero de la traza a generar

```
void showListTaskInfo(struct taskInfo *fun);
```

La función *showListTaskInfo* es una llamada utilizada para depurar. Esta llamada muestra las tareas que se registran en cada proceso, donde:

- *fun* es el puntero a la tarea actual de la que va a mostrar su información

```
char * obtainFileName(const int fd);
```

La función *obtainFileName* obtiene el nombre de un archivo a partir de su descriptor de fichero, donde:

- *fd* es el descriptor del fichero del que se quiere obtener el nombre.

3.2.2.Llamadas de E/S

En esta sección se describen las funciones desarrolladas de *traceLib* para capturar y gestionar las llamadas de E/S

```
int creat_trace(const char *path, mode_t mode, const int myRank);
```

Esta función procesa las invocaciones a *creat* de la biblioteca estándar de C, encargada de crear un fichero, donde:

- *path* es nombre del fichero a crear.
- *mode* es el modo de creación del fichero.
- *myRank* es el identificador del proceso que invoca la llamada.

```
int open_trace(const char *pathname, int flags, const int myRank);
```

Esta función procesa las invocaciones a *open* de la biblioteca estándar de C, encargada de abrir un fichero, donde:

- *pathname* es el nombre del fichero a abrir.
- *flags* es el modo de apertura del fichero.
- *myRank* es el identificador del proceso que invoca la llamada.

```
FILE *fopen_trace(const char *path, const char *mode, const int myRank);
```

Esta función procesa las invocaciones a *fopen* de la biblioteca estándar de C, encargada de abrir un fichero, donde:

- *path* es el nombre del fichero a abrir.
- *mode* es el modo de apertura del fichero.
- *myRank* es el identificador del proceso que invoca la llamada.

```
int close_trace(int fd, const int myRank);
```

Esta función procesa las invocaciones a *close* de la biblioteca estándar de C, encargada de cerrar un fichero, donde:

- *fd* es el descriptor de fichero que se va a cerrar.
- *myRank* es el identificador del proceso que invoca la llamada.

```
int fclose_trace(FILE *stream, const int myRank);
```

Esta función procesa las invocaciones a *fclose* de la biblioteca estándar de C, encargada de cerrar un fichero, donde:

- *stream* es el descriptor de fichero que se va a cerrar.
- *myRank* es el identificador del proceso que invoca la llamada.

```
ssize_t read_trace(int fd, void *buf, size_t count, const int myRank);
```

Esta función procesa las invocaciones a *read* de la biblioteca estándar de C, encargada de leer un fichero , donde:

- *fd* es el descriptor de fichero a leer.
- *buf* es el buffer donde se guardan los datos leídos.
- *count* es el número de bytes leídos.
- *myRank* es el identificador del proceso que invoca la llamada.

```
ssize_t pread_trace(int fd, void *buf, size_t count, off_t offset,
                    const int myRank);
```

Esta función procesa las invocaciones a *pread* de la biblioteca estándar de C, encargada de leer un fichero, donde:

- *fd* es el descriptor de fichero a leer
- *buf* es el buffer donde se guardan los datos leídos.
- *count* es el número de datos leídos.
- *offset* es el desplazamiento en el fichero.
- *myRank* es el identificador del proceso que invoca la llamada.

```
ssize_t fread_trace(void *ptr, size_t size, size_t nmemb, FILE *stream,
                    const int myRank);
```

Esta función procesa las invocaciones a *fread* de la biblioteca estándar de C, encargada de leer un fichero, donde:

- *ptr* es el puntero al buffer donde se guardan los datos leídos.
- *size* es el tamaño de los datos a leer.
- *nmemb* es el número de datos a leer.
- *stream* es el puntero al descriptor del fichero del que se leen los datos.
- *myRank* es el identificador del proceso que invoca la llamada.

```
ssize_t write_trace(int fd, void *buffer, size_t count, const int myRank);
```

Esta función procesa las invocaciones a *write* de la biblioteca estándar de C, encargada de escribir en un fichero. Los parámetros que recibe son:

- *fd* es el descriptor del fichero donde se escribe .
- *buf* es el buffer de donde se leen los datos a escribir.
- *count* es el número de bytes escritos.

- *myRank* es el identificador del proceso que invoca la llamada.

```
ssize_t pwrite_trace(int fd, void *buffer, size_t count, off_t offset,
                     const int myRank);
```

Esta función procesa las invocaciones a *pwrite* de la biblioteca estándar de C, encargada de escribir en un fichero, donde:

- *fd* es el descriptor de fichero a escribir
- *buffer* es el buffer donde se leen los datos a escribir.
- *count* es el número de datos a escribir.
- *offset* es el desplazamiento en el fichero.
- *myRank* es el identificador del proceso que invoca la llamada.

```
ssize_t fwrite_trace(const void *ptr, size_t size, size_t nmemb, FILE
                   *stream, const int myRank);
```

Esta función procesa las invocaciones a *fwrite* de la biblioteca estándar de C, encargada de escribir en un fichero, donde:

- *ptr* es el puntero al buffer de donde se leen los datos a escribir.
- *size* es el tamaño de los datos a escribir.
- *nmemb* es el número de datos a escribir.
- *stream* es el puntero al descriptor del fichero donde se escriben los datos.
- *myRank* es el identificador del proceso que invoca la llamada.

3.2.3.Llamadas MPI

En esta sección se describen las funciones desarrolladas de *traceLib* para capturar y gestionar un subconjunto de llamadas MPI.

```
int MPI_Send_trace(const void *buf, int count, MPI_Datatype datatype,
                   int dest, int tag, MPI_Comm comm, const int myRank);
```

Esta función procesa las invocaciones a *MPI_Send* de la biblioteca MPI, encargada de enviar datos de un proceso origen a un Worker destino, donde:

- *buf* es el puntero a los datos enviados.

- *count* es el número de datos enviados.
- *datatype* es el tipo de datos enviados.
- *dest* es el identificador del proceso destino.
- *tag* es la etiqueta o id único del mensaje.
- *comm* es el comunicador.
- *myRank* es el identificador del proceso que invoca la llamada.

```
int MPI_Recv_trace(void *buf, int count, MPI_Datatype datatype, int source,
                   int tag, MPI_Comm comm, MPI_Status *status,
```

Esta función procesa las invocaciones a *MPI_Recv* de la biblioteca MPI, encargada de recibir datos, donde:

- *buf* es el puntero a la variable donde se guarda los datos recibidos.
- *count* es el número de datos recibidos.
- *datatype* es el tipo de datos recibidos.
- *source* es el identificador del proceso emisor del mensaje.
- *tag* es la etiqueta o id único del mensaje.
- *comm* es el comunicador.
- *myRank* es el identificador del proceso que invoca la llamada.

```
int MPI_Scatter_trace(const void *sendbuf, int sendcount, MPI_Datatype
                   sendtype, void *recvbuf, int recvcount, MPI_Datatype
                   recvtype, int root, MPI_Comm comm, const int myRank);
```

Esta función procesa las invocaciones a *MPI_Scatter* de la biblioteca MPI, encargada de dividir los datos desde un proceso *master* y repartirlos a todos los procesos disponibles, donde:

- *sendbuf* es el puntero a la variable con los datos que se van a enviar.
- *sendcount* es el número de datos a enviar a cada Worker.
- *sendtype* es el tipo de datos enviados.
- *recvbuf* es el puntero a la variable donde se va a guardar los datos recibidos.
- *recvcount es el número de datos a recibir.*
- *recvtype* es el tipo de datos recibidos.
- *root* es el identificador del proceso que emite el mensaje.
- *tag* es la etiqueta o id único del mensaje.
- *comm* es el comunicador.
- *myRank* es el identificador del proceso que invoca la llamada.

```
int MPI_Gather_trace(const void *sendbuf, int sendcount, MPI_Datatype
                sendtype, void *recvbuf, int recvcount, MPI_Datatype
                recvtype, int root, MPI_Comm comm, const int myRank);
```

Esta función procesa las invocaciones a *MPI_Gather* de la biblioteca MPI, encargada de agrupar los mensajes recibidos desde todos los procesos disponibles, donde:

- *sendbuf* es el puntero a la variable con los datos que se van a enviar a root.
- *sendcount* es el número de datos a enviar a root.
- *sendtype* es el tipo de datos enviados.
- *recvbuf* es el puntero a la variable donde se va a guardar los datos recibidos.
- *recvcount* es el número de datos a recibir.
- *recvtype* es el tipo de datos recibidos.
- *root* es el identificador del proceso que recibe los datos.
- *tag* es la etiqueta o id único del mensaje.
- *comm* es el comunicador.
- *myRank* es el identificador del proceso que invoca la llamada.

```
int MPI_Bcast_trace(void *buffer, int count, MPI_Datatype datatype,
                    int root, MPI_Comm comm, const int myRank);
```

Esta función procesa las invocaciones a *MPI_Bcast* de la biblioteca MPI, encargada de enviar datos desde un proceso *master* origen a todos los procesos *workers* disponibles, donde:

- *buffer* es el puntero a la variable con los datos a repartir
- *count* es el número de datos a transmitir.
- *datatype* es el tipo de datos transmitidos.
- *root* es el identificador del proceso emisor.
- *tag* es la etiqueta o id único del mensaje.
- *comm* es el comunicador.
- *myRank* es el identificador del proceso que invoca la llamada.

```
int MPI_Init_trace (int * argc, char *** argv);
```

Esta función procesa las invocaciones a *MPI_Init*, encargada de iniciar el entorno MPI, donde:

- *argc* son los argumentos recibidos al ejecutar la aplicación.
- *argv* son los Argumentos recibidos al ejecutar la aplicación.

```
int MPI_Comm_size_trace (MPI_Comm comm, int * size);
```

Esta función procesa las invocaciones a *MPI_Comm_size,* encargada de obtener el número de procesos lanzados en la ejecución MPI. Los parámetros que recibe son:

- *comm* es el comunicador.
- *size* es el puntero a la variable donde se guardará el número de Workers lanzados.

```
int MPI_Comm_rank_trace (MPI_Comm comm, int * rank);
```

Esta función procesa las invocaciones a *MPI_Comm_rank,* encargada de obtener el rank del proceso en ejecución, donde:

- *comm* es el comunicador.
- *rank* es el puntero a la variable en la que registra el rank del Worker que realiza el proceso

```
int MPI_Finalize_trace ();
```

Esta función procesa las invocaciones a MPI_Finalize, se encarga de finalizar el entorno MPI, y llamar a la función interna de la biblioteca traceEnd.

3.2.4.Estructura de datos para el registro de tareas

La siguiente figura muestra la estructura de datos que representa cada *tarea* registrada por la biblioteca:

```
struct taskInfo{
      char name[9];
      char *pathname;
      double timeStamp;
      int data_size;
      int offset;
      int my_rank;
      int rank_dst_src;
      int id;/
      struct taskInfo *next;
};
```

Figura 3.2 Estructura de datos biblioteca

donde:

- *name* registra el nombre de la tarea.

- *pathname* registra el nombre del fichero pasado por parámetro en las llamadas de E/S.
- *timeStamp* registra la marca de tiempo vinculada al momento inicio o final en el que es invocada la llamada actual.
- *data_size* registra el tamaño de los datos que se transmiten, leen o escriben por la llamada actual.
- *offset* registra el desplazamiento en el fichero, sólo en las llamadas pwrite() y pread().
- *my_rank* registra el *rank* del proceso que invoca la llamada.
- *rank_dst_src* registra el *rank* del proceso destino u origen, dependiendo de la llamada MPI.
- *id* Registra el Id único asociado a cada par de *tareas,* inicial y final, y al *rank* del proceso que realiza la llamada.
- *next* puntero hacia la siguiente *tarea*.

3.2.5.Fichero registro obtenido

Al ejecutar un programa MPI enlazado con *TraceLib*, se obtienen varios ficheros de traza. Cada fichero se corresponde con la ejecución de un proceso. Para poder obtener una traza que represente la ejecución de la aplicación completa deben tenerse en cuenta las siguientes consideraciones:

- Escribir el fichero de traza en tiempo real, mientras se ejecuta la aplicación MPI genera un problema. Este inconveniente consiste en que al acceder de forma concurrente a un mismo fichero, se mezclan escrituras, lo que en la mayoría de casos generan perdidas de registros. La opción de usar *mutex* se descarta ya que se desvirtúan los tiempos. Por ello es necesario que cada proceso genere su propio fichero de traza, y estos se integren posteriormente de forma global.
- Las marcas de tiempo inicial y final relacionadas a una *tarea* se deben separar, con el fin de poder ordenar las tareas por orden de ejecución al integrar los ficheros de trazas. La figura 3.3 es un ejemplo del resultado de una invocación a una llamada *MPI_Bcast*:

```
0.042719 Ini_bcas 0 2 1 0
0.987113 End_bcas 0 2 1 0
```

Figura 3.3

donde, en ambos registros:

- El primer parámetro es la marca de tiempo asociada a la *tarea.*
- El segundo parámetro es el nombre asociado a la tarea, tanto inicial como final.
- El tercer parámetro representa el Id de la tarea.
- El cuarto parámetro representa el rank. El identificador del proceso receptor, que invoca la llamada.
- El quinto parámetro representa la cantidad de datos transmitidos.
- El sexto parámetro representa el rank del emisor. El identificador del proceso que envía los datos a los procesos receptores.

- Generar marcas de tiempos independientemente en cada proceso conlleva que los tiempos registrados en cada traza sean relativos al proceso, es decir, el tiempo inicial en cada proceso es distinto. Por tanto las marcas de tiempo son calculadas sobre su marca inicial, esto da lugar a un posible orden incorrecto de tareas.
- El registro de cada invocación requiere que cada *tarea* sólo registre los datos asociados a los parámetros usados en la llamada actual, por lo que es necesario adaptar la escritura de cada tarea a distintas estructuras de llamadas. En la figura 3.4 observamos el registro de una invocación a *pwrite,*

```
0,068015 Ini_pwri 1 0 13 5 salida3.txt
0,068027 End_pwri 1 0 13 5 salida3.txt
```

Figura 3.4

donde:

- El primer parámetro es la marca de tiempo asociada a la *tarea.*
- El segundo parámetro es el nombre asociado a la tarea, tanto inicial como final.
- El tercer parámetro representa el Id de la tarea.
- El cuarto parámetro representa el rank. El identificador del proceso que invoca la llamada.
- El quinto parámetro representa la cantidad de datos transmitidos.
- El sexto parámetro representa el offset, el desplazamiento en el fichero escrito.
- El séptimo parámetro representa el nombre del fichero en el que se escribe.

Al comparar la figura 3.3 y la figura 3.4 se observa que cada invocación genera registros con sólo los datos necesarios de cada llamada.

Teniendo en cuenta los problemas surgidos durante el desarrollo, se obtiene una versión global de la traza en la que los datos registrados representan las llamadas en el mismo orden en que fueron invocadas.

3.2.6.Ejemplo

En la figura 3.5 podemos observar la diferencia entre un programa MPI y un programa MPI que hace uso de *traceLib*. Como podemos observar, las diferencias entra ambas son mínimas. En el primer ejemplo vemos cómo se añade la biblioteca “traceLib.h”, las llamadas tienen el sufijo “_trace” y la llamada *MPI_Broadcast* tiene un parámetro añadido, el rank del proceso que invoca la llamada.

```c
#include <stdio.h>
#include "mpi.h"


#define MASTER 0

int main( int argc, char **argv){

    int rank, valor, nprocs;

    //Inicializamos MPI y obtenemos el rank
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);

    // Proceso Master?
    if (rank == 0){
                printf ("Introduce un valor:\n");
                scanf( "%d", &valor);
    }

    // Broadcast del mensaje
    MPI_Bcast(&valor, 1, MPI_INT,MASTER, MPI_COMM_WORLD);

    printf( "Proceso %d recibe:[%d]\n", rank, valor);


    MPI_Finalize();
    return 0;
}
```

Figura 3.5.1 Aplicación original

```c
#include <stdio.h>
#include "mpi.h"
#include "traceLib.h"

#define MASTER 0

int main( int argc, char **argv){

    int rank, valor, nprocs;

    //Inicializamos MPI y obtenemos el rank
    MPI_Init_trace(&argc, &argv);
    MPI_Comm_size_trace(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank_trace( MPI_COMM_WORLD, &rank);

    // Proceso Master?
    if (rank == 0){
                printf ("Introduce un valor:\n");
                scanf( "%d", &valor);
    }

    // Broadcast del mensaje
    MPI_Bcast_trace(&valor, 1, MPI_INT, MASTER,
                    MPI_COMM_WORLD,rank);

    printf( "Proceso %d recibe:[%d]\n", rank, valor);

    MPI_Finalize_trace();
    return 0;
}
```

Figura 3.5.2 Aplicación transformada

La ejecución de esta aplicación con 3 procesos, genera la siguiente traza:

```
0.062846 Ini_bcas 0 1 1 0
0.062860 Ini_bcas 0 2 1 0
1.007204 End_bcas 0 0 1 0
1.007254 End_bcas 0 1 1 0
1.007090 Ini_bcas 0 0 1 0
1.007254 End_bcas 0 2 1 0
```

Donde observamos la invocación *MPI_Broadcast* por parte de los 3 procesos y las tareas están ordenadas por su marca de tiempo .

3.3. Fase 2: Integración con SIMCAN

En esta sección se detalla el desarrollo realizado sobre el simulador SIMCAN. Para ello se explicará brevemente la arquitectura del simulador. En la figura 3.7 observamos la arquitectura del SIMCAN.

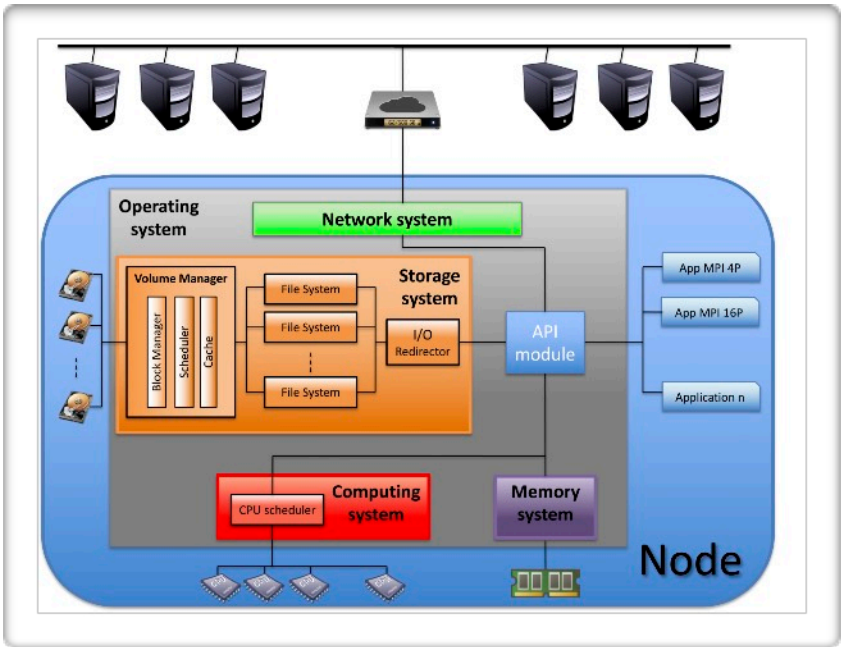

Figura 3.6 Arquitectura SIMCAN

Como se observa, los componentes necesarios para ejecutar una simulación son:

- *Operating system* procesa las llamadas de cada aplicación y las redirecciona a la parte correspondiente.
- *Network system* representa la red de comunicaciones.
- Storage system representa el sistema almacenamiento.
- *Computing system* representa el sistema de cómputo. Gestiona 1 o varios procesadores.
- *Memory system* representa el sistema de memoria.
- *API module* contiene las llamadas que permiten utilizar el hardware del nodo y subsistemas principales

De forma detallada, observamos la arquitectura de SIMCAN.

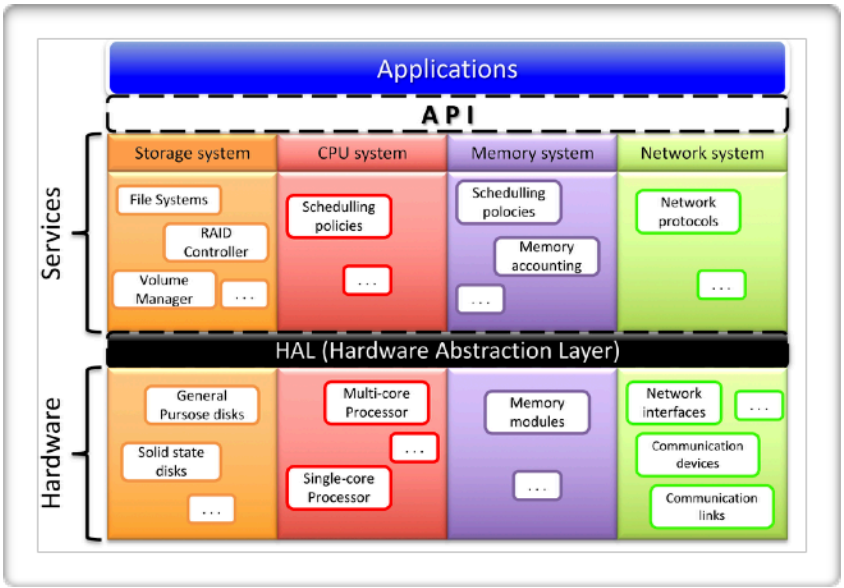

Figura 3.7 Arquitectura API

Como se observa, la aplicación hace uso directo de la API. Esta API tiene acceso a los servicios disponibles, donde cada servicio ha sido previamente modelado con una configuración hardware.

Para desarrollar la aplicación, es necesario conocer las llamadas API disponibles. En la figura 3.8 observamos estas llamadas.

```
void SIMCAN_request_create (const char* fileName);
void SIMCAN_request_open (const char* fileName);
void SIMCAN_request_close (const char* fileName);
void SIMCAN_request_read (const char* fileName, unsigned int offset, unsigned int size);
void SIMCAN_request_write (const char* fileName, unsigned int offset, unsigned int size);
virtual void mpi_send (unsigned int procID, int bufferSize);
virtual void mpi_send (SIMCAN_MPI_Message *sm_mpi);
virtual SIMCAN_MPI_Message * mpi_recv (unsigned int procID, int bufferSize);
virtual SIMCAN_MPI_Message * mpi_recv (unsigned int procID, int bufferSize);
virtual void mpi_bcast (unsigned int rootID, int bufferSize);
virtual void mpi_scatter (unsigned int rootID, int bufferSize);
virtual void mpi_gather (unsigned int rootID, int bufferSize);
```

Figura 3.8 Llamadas disponibles a la API de SIMCAN

Tras conocer la arquitectura y las llamadas disponibles, se lleva a cabo el desarrollo de la aplicación MPI.

Para reproducir el comportamiento de la aplicación MPI , es necesario:

- Leer el inicio de la llamada invocada. Al conocer el tipo de llamada, se elige la llamada a la API que se va a ejecutar. Cada llamada no solo reproduce el tiempo e ejecución, sino que también calcula el tiempo de computo necesario para ejecutar la llamada.
- Modificar las marcas temporales de cada tarea. Al leer cada tarea y calcular el tiempo de ejecución, se generan nuevas marcas de tiempo, tanto inicial como final. Estas marcas, son volcadas al fichero de traza del proceso en ejecución.
- Cada proceso reproducido registra su propio fichero traza debido a la concurrencia producida al intentar registrar todas las tareas de todos los procesos a un mismo fichero. Este problema surgió también en la Fase 1 por lo que abordarlo fue una tarea sencilla.

3.4. Fase 3: Representación gráfica del comportamiento de aplicaciones MPI

La Fase 3 consiste en el desarrollo de una aplicación encargada de mostrar gráficamente los resultados obtenidos de las ejecuciones MPI, tanto las ejecuciones reales generadas en la Fase 1 como las aplicaciones simuladas de la Fase 2.

Esta aplicación se ha desarrollado principalmente en lenguaje Java y el uso de Script en Bash. Se ha llevado a cabo en distintas Fases.

3.4.1. Procesamiento de los datos de entrada

Las trazas generadas en la Fase 1 y 2 tienen el mismo formato. La importancia de haber elegido una estructura adecuada para el registro de las tareas ejecutadas ayuda notablemente en esta etapa, ya que es el momento en el que los datos se adaptarán y utilizarán para generar el gráfico que representa el comportamiento de las aplicaciones.

Los datos leídos se agruparán de distintas formas necesarias para los distintas funcionalidades disponibles al mostrar el gráfico.

En primer lugar, se transforma la traza obtenida por la ejecución de la aplicación MPI a un formato JSON. Este formato JSON nos permite hacer uso de estructuras de datos ya implementadas en Java, sencillas y con un rendimiento conocido.

Veamos un ejemplo de como se transforman estos datos a un formado JSON.

Formato obtenido en la traza en la Fase 1 y 2	Formato generado en JSON
0,06791 Ini_fope 0 0 salida3.txt 0,067933 End_fope 0 0 salida3.txt 0,068015 Ini_pwri 1 0 13 5 salida3.txt 0,068027 End_pwri 1 0 13 5 salida3.txt	

El formato generado es una lista de objetos de tipo Map. Este formato JSON nos permite obtener una estructura List<Map < String , String >> haciendo uso de la API Gson y JsonReader, de modo que el tratamiento de estos datos sea sencillo.

3.4.2.Funcionalidades

La aplicación se ha desarrollado con el fin de unificar todas las tareas necesarias para simular programas en distintos entornos simulados de la manera más sencilla y transparente para el usuario. Cada funcionalidad conlleva una consecución de tareas que, sin la aplicación, darían lugar a largos procesos y necesitando varios comandos para llegar al resultado que obtenemos. Es en este punto donde todo el trabajo previo, realizado en la Fase 1 y Fase 2, se integra para dar lugar a la representación gráfica de nuestros programas. A continuación se detallará.

3.4.2.1.Carga de aplicaciones MPI

La primera funcionalidad que nos encontramos es la carga de nuestro programa MPI implementado con las llamadas a nuestra biblioteca TraceLib.

Al pulsar el botón “Browse…”, se muestra una ventana emergente, lanzada por *chooseFile*, en la que podremos elegir el programa MPI que deseamos ejecutar y simular. Una vez se elige el fichero, se envía al Controlador mediante *controller.loadMPIFile* y éste a su vez al Modelo, mediante *model.loadMPIFile*. En el modelo, se hace el control de errores. En él se verifica que la extensión del fichero MPI sea “.c”. Si la comprobación es correcta se devolverá un mensaje indicando el OK de la operación y la vista habilitará los botones “loaded” y “compile” del menú mediante *enableButtonLoadMPI*, en caso contrario, la vista recibe un mensaje de KO que avisa al controlador del error y éste devuelve una petición a la vista para lanzar un PopUp con el motivo del error.

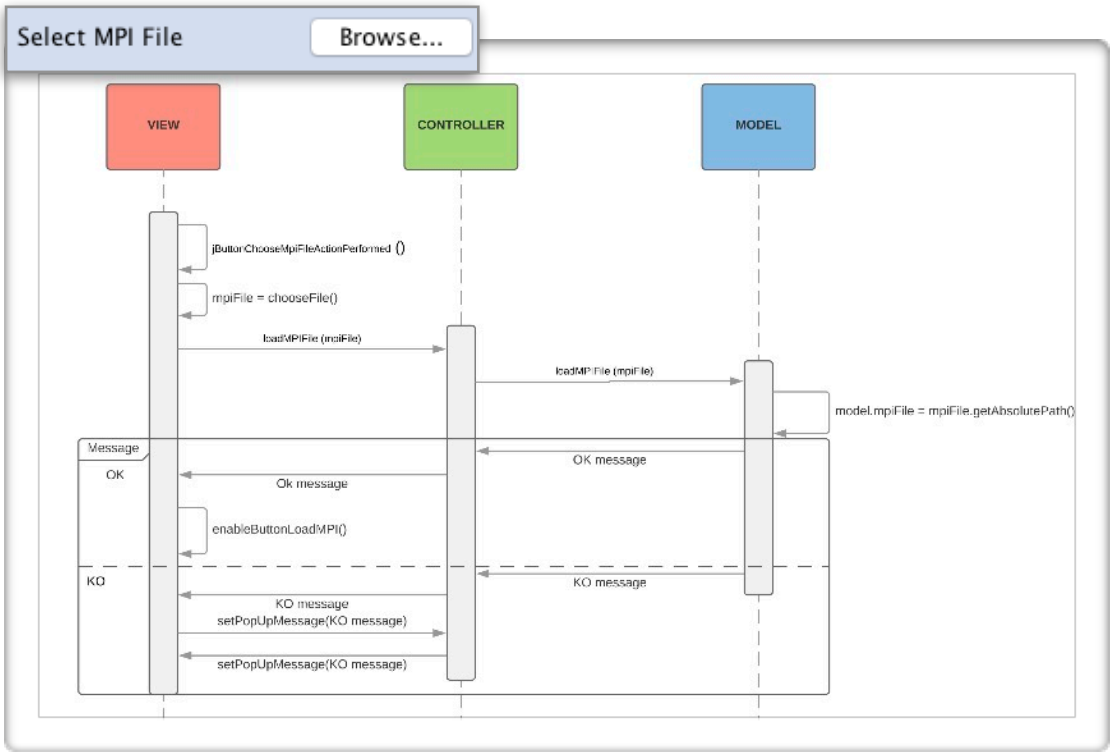

Figura 3.9 Diagrama de secuencia. Carga de aplicación MPI

3.4.2.2. Compilado de programa MPI

Una vez se ha cargado el fichero .c, el botón “Load” ha pasado a ser “Loaded” y el botón “Compile” es habilitado. Este botón es el encargado de compilar el fichero y generar el ejecutable con el que se podrá lanzar el programa.

Una vez se pulsa el botón, una ventana emergente nos solicitará el directorio y el nombre del ejecutable que vamos a generar, con esto permitimos al usuario conocer y decidir donde se generan los ficheros necesarios para la simulación. Tras elegir el directorio y nombre, se envía al controlador la información mediante controller.compileMPIFile() y éste a su vez llama a model.compileMPIFile() con la misma información. En el modelo es donde se llevan a cabo las diversas tareas que requiere esta funcionalidad.

En primer lugar se copia la cabecera de la biblioteca, *TraceLib.h*, en el directorio donde se encuentra el fichero .c que se va a compilar. Esto es necesario ya que al ejecutar el comando de compilado, no solo es necesario tener nuestro fichero TraceLib.o sino que también la cabecera TraceLib.h debe estar disponible para ser leída y compilada junto a nuestro fichero MPI. Una vez se ha copiado, se procede a ejecutar el script “compileMPIFile.sh” mediante el uso de comandos Bash. Este Script puede recibir también, como parámetro, una biblioteca auxiliar si fuera necesaria para nuestro programa. Para la ejecución de comandos Bash se ha utilizado una clase intermedia, Command, a la que solo es necesario pasar el comando y/o el directorio de ejecución para que proceda a realizar las llamadas necesarias para ejecutar el comando. Tras ejecutar el Script de compilado, si no ha habido error en la ejecución se procede a borrar la cabecera TraceLib.h del directorio donde se copió y se guarda el directorio y nombre del fichero ejecutable generado para posteriormente ser ejecutado.

Si todo ha ido bien, se envía un mensaje de OK a la vista que habilitará el botón “Execute” , en caso contrario, mostrará un PopUp con el mensaje de error generado.

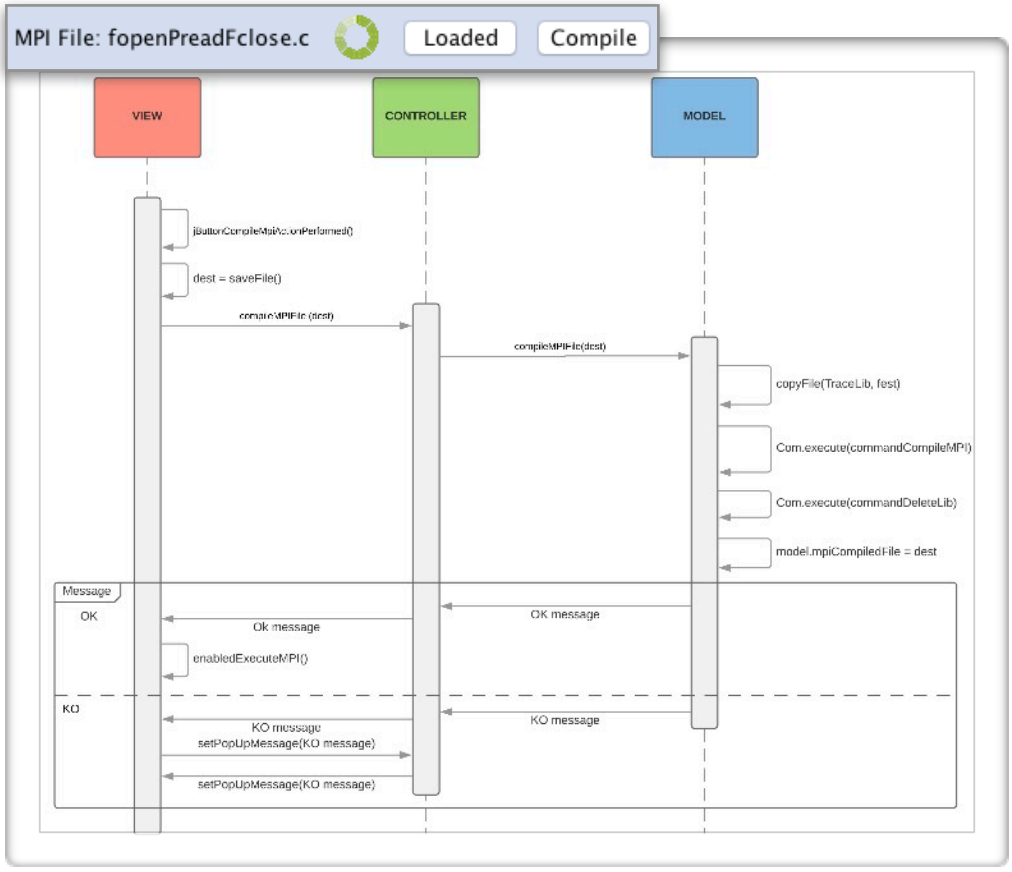

Figura 3.10 Diagrama de secuencia. Compilado de aplicación MPI

3.4.2.3. Ejecución de programa MPI

Tras generar el fichero ejecutable, el botón “Execute” ha sido activado. Al pulsar este botón, se lanzan dos ventanas emergentes. Una en la que es necesario indicar el número de *Workers* con los que se quiere lanzar la ejecución del programa, y otra en la que es posible introducir parámetros que puedan ser necesarios para su ejecución.

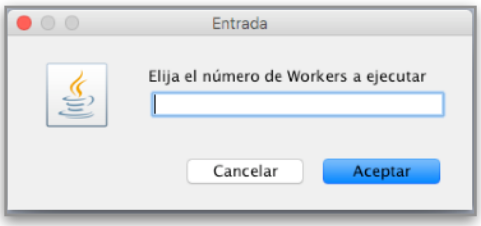

Figura 3.11 Cuadro de dialogo. Procesos

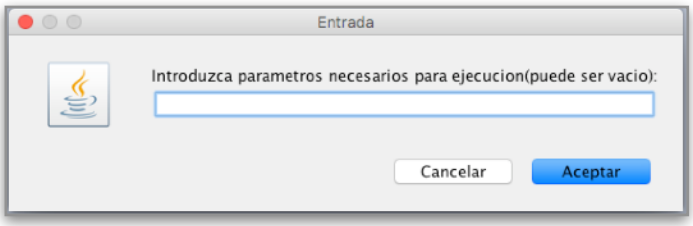

Figura 3.12 Cuadro de dialogo. Parámetros

Estos parámetros son enviados al Controlador mediante controller.executeMPIFile() que a su vez los envía al Modelo mediante model.executeMPIFile().

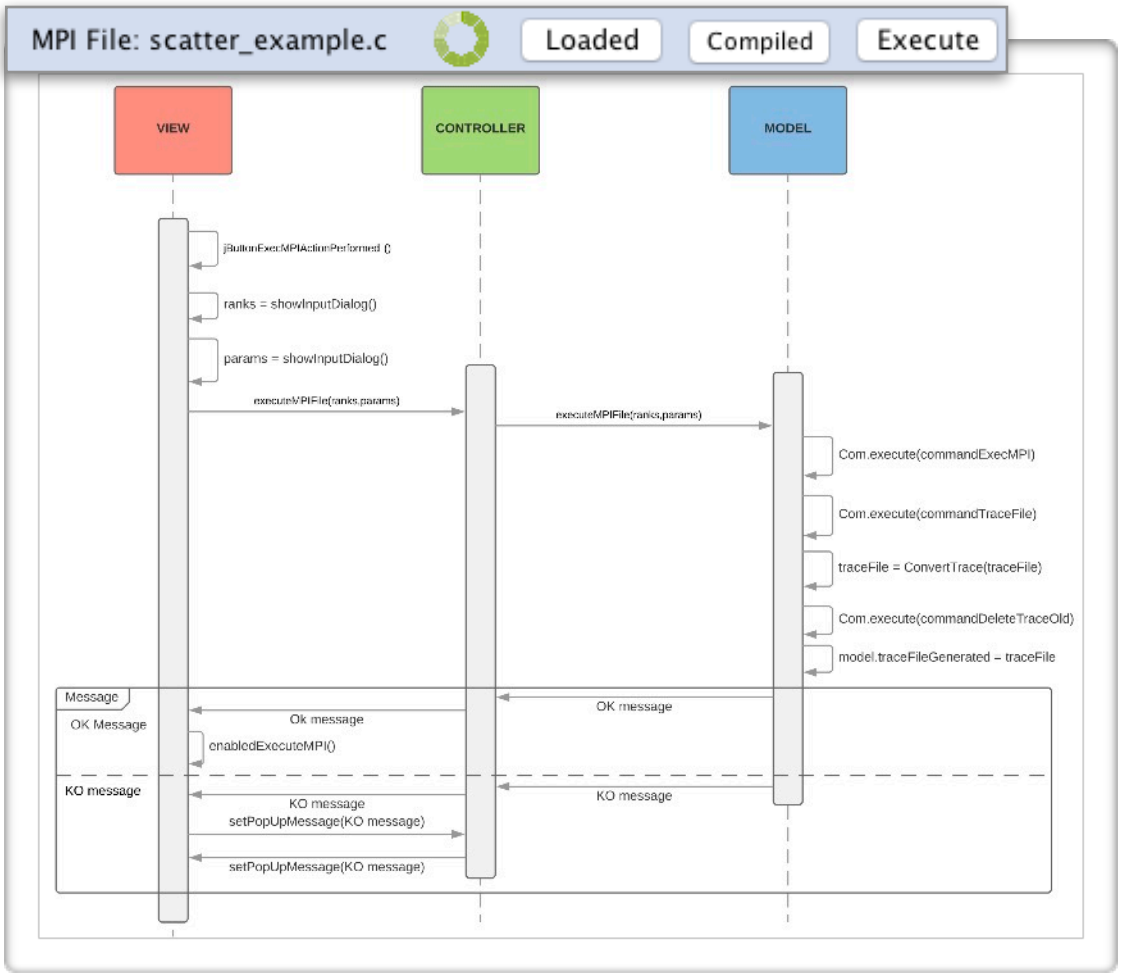

Figura 3.13 Diagrama de secuencia. Ejecución de aplicación MPI.

En el Modelo es donde llevamos a cabo todos los pasos necesarios para ejecutar nuestro programa. En primer lugar se genera el Script “executeMPI.sh” para lanzar la ejecución MPI

teniendo en cuenta los parámetros recibidos. En el caso de Workers es obligatorio y los parámetros son opcionales, depende de la aplicación cargada. Una vez tenemos todos los parámetros listos, el Script es lanzado. Esta ejecución producirá tantos ficheros “trace_X.txt” como workers se han solicitado lanzar. Todos los fichero son generados en la ruta que el usuario eligió al compilar el fichero.

Si la ejecución ha ido correctamente, se lanza el Script “generateTrace.sh”. Este Script es el encargado de integrar todas las trazas generadas por cada Worker y ordenarlas por su marca de tiempo asociada a cada tarea. Este nuevo fichero se guarda en el mismo directorio donde se encuentra el ejecutable del programa. Tras generarla, se procede a borrar las trazas respectivas a cada Worker.

Como ya comentamos en la Fase 1, ha sido necesario crear un programa que modifique las marcas de tiempo de cada tarea relacionando todas a un mismo instante inicial. Este proceso es lanzado en este instante, con la traza generada como entrada.

Una vez se obtiene esta traza definitiva, se guarda el nombre y directorio de este fichero y se devuelve un mensaje de OK a la Vista, a través del Controlador. En caso de KO se generará, como en anteriores ocasiones, un PopUp con el mensaje de error asociado.

La finalización correcta de este paso, activará un Flag MPI_Executed, que combinado a un Flag SIMCAN_Executed relacionado con la simulación SIMCAN, activará el botón “RUN” y “Step by Step” con los que se podrá generar el gráfico asociado al programa y simulación elegidos.

3.4.2.4. Carga de entorno SIMCAN

La carga del entorno SIMCAN es similar a la carga del programa MPI, con la única diferencia del control de errores, que en este caso se comprobará que el fichero sea un ejecutable y se encuentre en un directorio con los ficheros SIMCAN necesarios. En este caso el botón “Load” y “Simulate” también pasa a estar activado

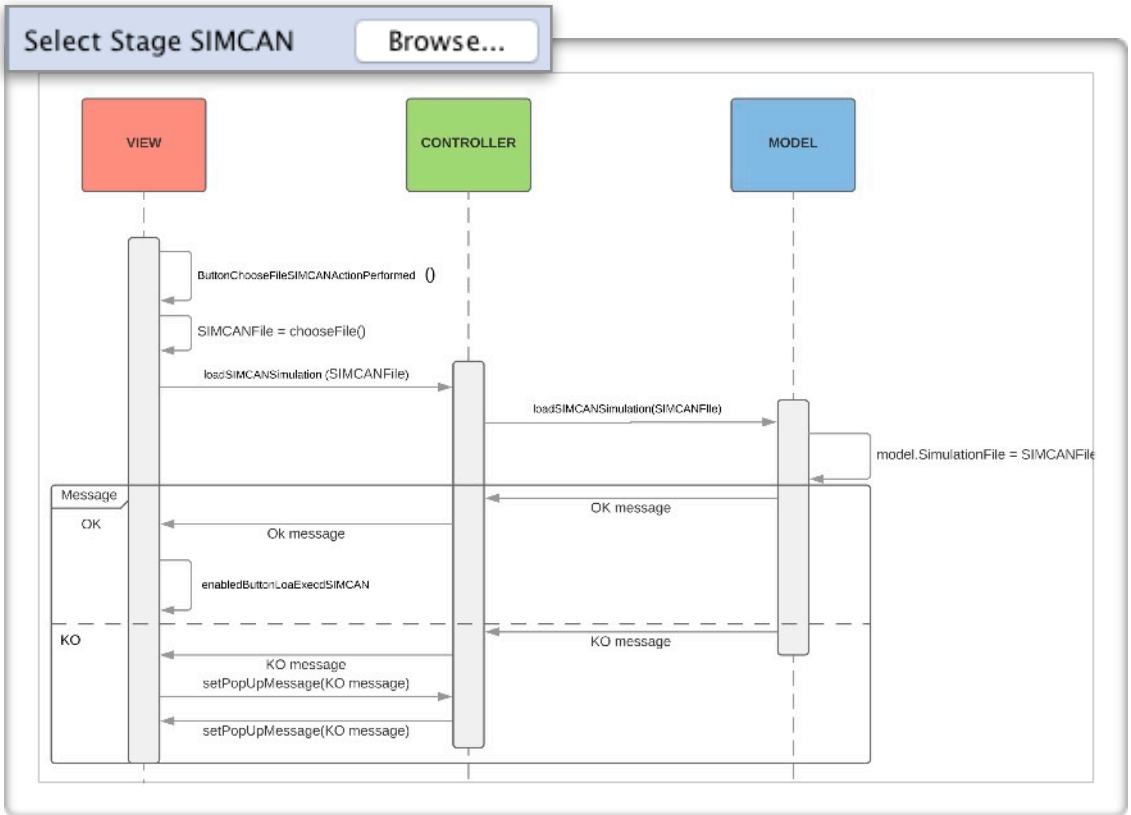

Figura 3.14 Diagrama de secuencia. Carga de entorno SIMCAN.

3.4.2.5. Ejecución de entorno de simulación SIMCAN

Al lanzar la ejecución de la simulación, se llama al Controlador mediante executeSIMCAN() y a su vez se llama a executeSIMCANSimulation(), del Modelo. Primero se copia el fichero de traza, generado por la aplicación MPI, al directorio donde se encuentra el ejecutable. Tras ello se ejecuta el Script executeSIMCAN.sh. Tras ejecutarlo se envía el resultado de la operación a la Vista a través del Controlador. Si la ejecución ha sido correcta los botones “RUN” y “Step by Step” son habilitados y se podrá generar el gráfico.

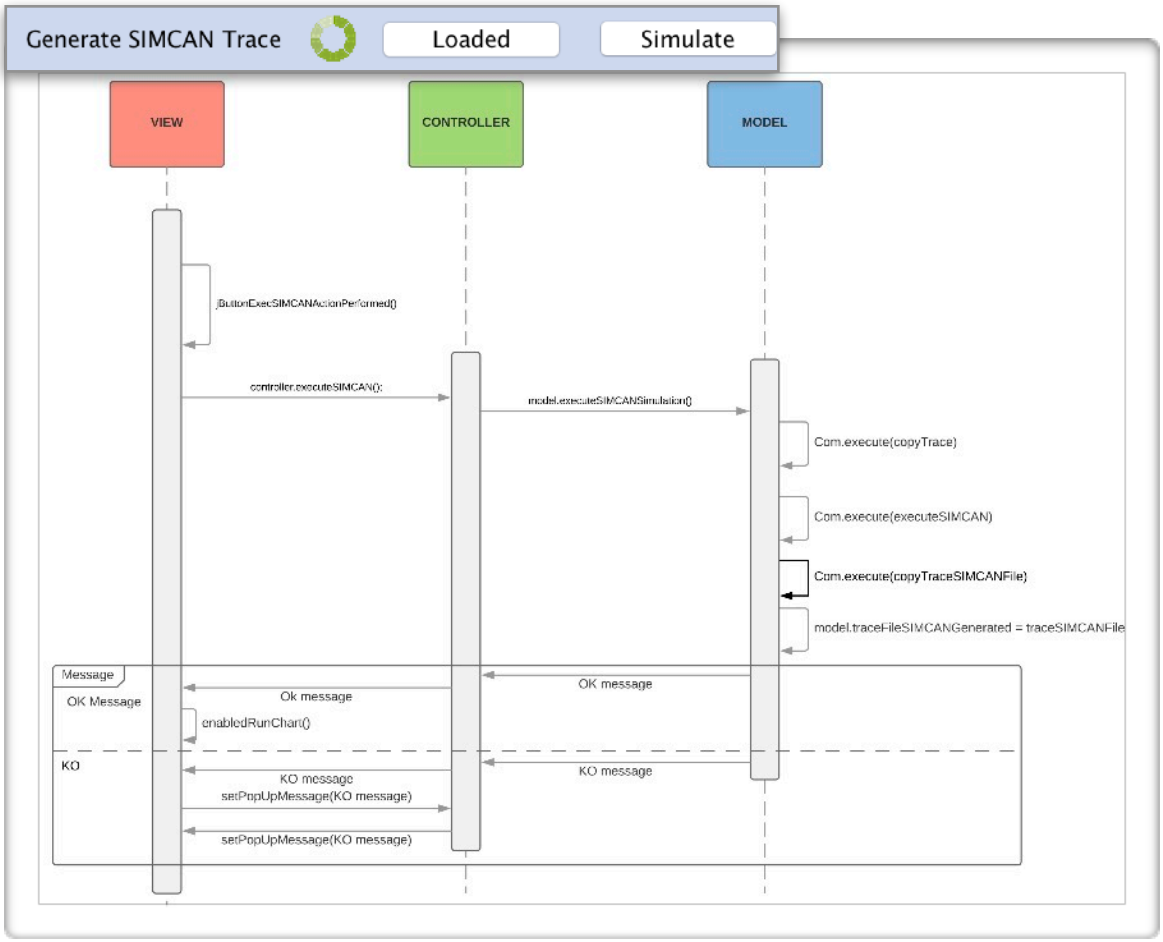

Figura 3.15 Diagrama de secuencia. Ejecución de simulación.

3.4.3. Diseño de la interfaz

Una de las partes mas importantes del desarrollo de este proyecto ha sido el diseño de la interfaz gráfica. El diseño de la aplicación busca que los usuarios puedan hacer uso de la aplicación de manera intuitiva y rápida, ya que esta aplicación pretende acercar al alumno a su propio desarrollo, esto es, al programa que él ha creado.

Teniendo en cuenta las necesidades de los alumnos, se han llevado a cabo distintos diseños buscando el mejor resultado.

La interfaz consta de 3 secciones principales. Cada botón se asocia a una funcionalidad, explicadas en el apartado 3.4.2.

La sección superior de la interfaz es un panel de control destinado a las funcionalidades necesarias para generar el gráfico. Esta sección a su vez se divide en 3 secciones.

La primera sección se destina a cargar, compilar y ejecutar la aplicación MPI que se desea simular. La segunda sección se encarga de cargar y ejecutar la simulación de un entorno SIMCAN. Y por último, la sección inferior se encarga de generar el gráfico, ya sea de manera completa, o en modo “Paso a paso”.

En la sección central e inferior es donde se cargan los gráficos generados a partir de aplicaciones MPI. Ambos permiten la carga de ficheros de traza generados, por lo que no es necesario ejecutar siempre las aplicaciones, sino que se pueden guardar para posteriormente ser comparadas.

En el caso de ejecutar una aplicación MPI y simularla, el gráfico generado se representa en la sección central.

Ambas secciones tienen un panel de control, en el que tienen las opciones de guardado, carga, desplazamiento de gráfico y desplazamiento en tareas.

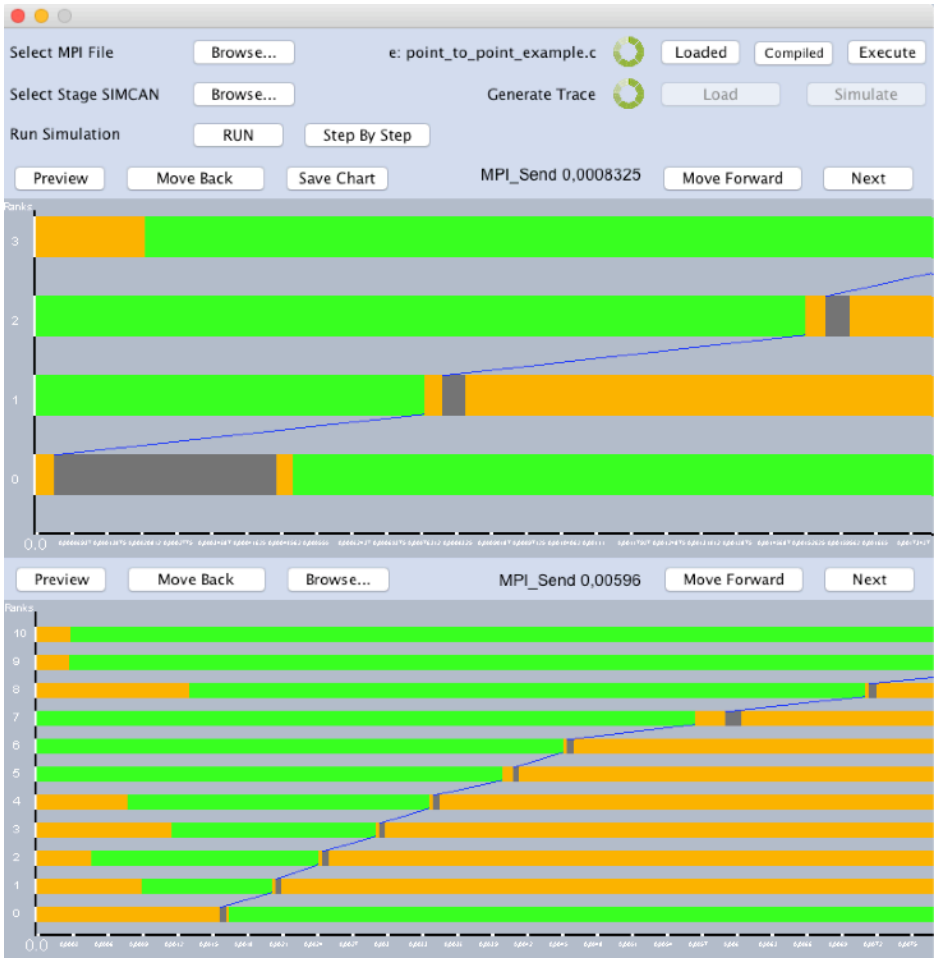

Figura 3.16. Aplicación para la representación de aplicaciones MPI

3.4.4.Generación de gráfico

La primera dificultad surge al ejecutar un programa en el que se hacen multitud de llamadas, una detrás de otra, tanto de E/S como de MPI. El problema reside en que los tiempos entre unas y otras tareas son unidades de microsegundos, por lo que hacer que sean “perceptibles” y visibles, en un gráfico de cierta amplitud, es realmente complicado. Si por ejemplo, tenemos un programa que tarda 2 segundos aproximadamente, tendremos 2 millones de microsegundos.

Una primera idea es relacionar un pixel a un microsegundo, esta solución en algunas aplicaciones en las que las ejecuciones duran algunos milisegundos, es aceptable y sobre todo, visible, pero en aplicaciones tan “largas” como las de 2 segundos, se convierten en intratables debido a la magnitud necesaria para el gráfico. Intentar relacionar un pixel a un microsegundo provocaría que tuviésemos gráficos de una longitud de 2 millones de pixeles para una ejecución de 2 segundos, esto serían más de 1500 pantallas de portátiles de tamaño medio colocadas de manera contigua. Por tanto, esto convierte esta alternativa en inviable.

Un primer acercamiento a una solución es comprobar que tarea es la que menor tiempo conlleva, estableciendo que, por ejemplo una tarea que dure 50 microsegundos sea relacionada con un pixel. Así por ejemplo aunando el ejemplo de un programa de 2 segundos y esta solución teniendo en cuenta la tarea mas corta, llegaríamos a tener un gráfico de aproximadamente 40 mil pixeles, algo ya más aceptable, teniendo en cuenta que si tenemos un panel deslizable, podremos llegar hasta el final de la simulación. Ahora bien, que ocurre cuando tenemos una ejecución de 75 microsegundos, ¿dibujamos 1 ó 2 pixeles? , si 1 se relaciona con 50 microsegundo. Una u otra decisión nos va a llevar a perder cierta información en el gráfico con la que tendremos que lidiar a lo largo del desarrollo.

Surge así un nuevo problema, calcular el pixel exacto o la longitud de la imagen que va a representar una tarea, ya que solo lo podemos expresar en pixeles, es decir, en número entero. Por tanto vamos a perder precisión sea cual sea la solución, teniendo en cuenta en que no vamos a relaciones pixel-microsegundo.

Teniendo en cuenta ambos problemas, avanzamos y probamos distintas soluciones que nos permitan mantener una buena relación entre precisión-visibilidad.

Otra problemática viene derivada de esta perdida de precisión es que nos encontramos en ciertas situación en las que 2 o más tareas consecutivas se solapan, por lo que es necesario, también, reflejar todas las tareas sin perder detalles.

Todos estos problemas surgidos nos llevan a plantear una solución que mitigue todos o la mayor parte de los problemas.

- **Relación pixel-tiempo:** esta relación se establece tomando la experiencia de las primeras versiones del gráfico. Por una parte, la tarea más corta, será la que marque esta relación. Por otra, la tarea más corta no es representada por un pixel, sino por un tamaño inicial fijo de pixeles, inicial porque si el tamaño ocupado por la simulación no ocupa al menos todo el panel de visualización, se recalcula hasta poder sobrepasar este tamaño. Esto lo hacemos con el fin de que las aplicaciones muy rápidas no se vean en una mínima sección del panel, sino que se extiendan hasta tener un tamaño lo suficientemente visible.
- **Tamaño total del panel:** El tamaño del panel depende directamente de la relación pixel-tiempo. Por tanto, en ejecuciones largas, obtendremos un gráfico con una extensión en el eje X considerable. Para mitigar este problema de amplitud, añadimos una funcionalidad al gráfico, con el que, al desplazarnos de una tarea a otra, el gráfico se mueve automáticamente hacia la tarea, con el fin de no tener que desplazar continuamente el panel hasta llegar a la

siguiente tarea. Por otra parte, la altura del gráfico siempre es la misma, independientemente del número de *Workers* ejecutados, por lo que el grosor de cada línea asociada a un Worker será dinámico.

- **Representación de tareas:** A cada Worker le corresponde una línea horizontal en la que se representarán las tareas. Cada tarea será representada por un color único con lo que se consigue una mejor visualización del comienzo y final de cada tarea. Los tiempos y los nombres de las tareas son visualizados en la zona superior del panel. De manera que al pasar el cursor por encima de una tarea, se muestre tanto la tarea como la marca de tiempo asociada a la posición.

Con todas estas consideraciones, conseguimos unos gráficos claros y, lo más importante, dinámicos, con lo que podremos ejecutar todo tipo de aplicaciones sin restricciones. Vemos ejemplos de distintas ejecuciones de un mismo programa.

Figura 3.17 Ejemplo de dos ejecuciones MPI.

Como vemos en estos ejemplos, tenemos en el panel superior una ejecución con 4 Workers y en la inferior con 11 Workers. Cada una muestra las tareas y las comunicaciones de manera clara. Vemos como los tamaños de las tareas son completamente distintos, debido al dinamismo a la hora de calcular cada medida necesaria para generar el gráfico.

La simulación se habilitará cuando el programa MPI haya sido ejecutado y el entorno SIMCAN haya generado la simulación correctamente.

Las opciones disponibles son, “RUN” que genera el gráfico con el total de las tareas ejecutadas y “Step By Step” que genera el gráfico vacío con el objetivo de poder ejecutar tarea a tarea la simulación y ver los momentos exactos en los que se lanza cada tarea.

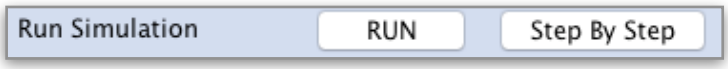

Figura 3.18 Botones para generar la simulación.

Al pulsar el botón “Step by Step”, se obtiene un fichero JSON que servirá para modelar y ordenar los datos. En este caso, el gráfico se mostrará sin tareas y cada una será mostrada en orden de ejecución al pulsar el botón “Next” o “Preview”.

Figura 3.19 Estado de la ejecución en t=0.

Al pulsar el botón “RUN”, la traza generada por el simulador SIMCAN es leída y transformada a formato JSON. Este formato JSON sirve de entrada para modelar los datos de las tareas y poder generar el gráfico. En este caso, todas las tareas son reflejadas al pulsar el botón.

Figura 3.20 Estado de la ejecución completa

El botón “Save Chart”, sólo disponible en el gráfico superior, nos permite guardar la simulación generada. Para ello abre una ventana emergente en la que debemos elegir el nombre y el directorio donde queremos guardar el fichero. Este fichero será una copia de la traza generada por SIMCAN al ejecutar la aplicación MPI.

El botón “Browse”, disponible en el menú del gráfico inferior, abre una ventana emergente, en la que es necesario elegir un fichero .txt con los datos de una aplicación previamente ejecutada o simulada. En este caso podemos cargar el fichero generado por la simulación o el fichero generado por la aplicación en el entorno local.

Cada gráfico tendrá un panel de control casi similar, la única diferencia está en que en el panel superior tendrá la opción “Save Chart” cuando una simulación haya sido generada por la aplicación y el panel inferior siempre tendrá la opción “Browse”.

Panel de control superior:

Figura 3.21 Panel de control de gráfico superior

Panel de control inferior:

Figura 3.22 Panel de control del gráfico inferior

Los botones “Previous” y “Next” nos permiten avanzar y retroceder en la ejecución de las tareas mostradas en el gráfico. De esta manera podemos comparar ejecuciones de manera más detallada.Los botones “Move Backward” y “Move Forward” nos permiten avanzar en el gráfico cuando supera los límites de nuestra ventana.

3.5.Scripts de ejecución

El desarrollo y uso de scripts ha sido necesario para poder ejecutar diversas tareas que no se pueden tratar desde Java. La implementación de estos Scripts no se ha realizado controlando errores en ellos, sino que el control de los errores se realiza en la aplicación Java. Las funcionalidades de estos scripts son:

- Copia de ficheros necesarios para la compilación del Programa MPI. ”copyFile.sh”

```
#!/bin/bash
echo Copy File in Folder
file=$1 # Source path/file
destination=$2 # Destination path/file
echo Try copy $file
echo in folder: $destination
cp $file  $destination
```

Este Script copia un fichero “file” en el destino “destination”.

- Compilación de los programas en C y linkado con la biblioteca TraceLib. “compileMPIFile.sh”

```
#!/bin/bash
echo Compile MPI File

file=$1 #Path/File .c to compile
lib=$2 #Path/File library needed to compila
where=$3 #Path/File executable generated
libAux=$4 #Library aux
pathMPI=$5 #Path of MPI library

export PATH=$PATH:$pathMPI

echo Try compile MPI file: $file
mpicc -o $where $file $lib $libAux
```

Este Script se encarga de compilar y enlazar las bibliotecas necesarias para el compilado. En este caso no solo enlazamos la biblioteca TraceLib en el parámetro “lib” sino que permitimos añadir otra biblioteca opcional en el caso de ser necesario, este parámetro es solicitado por la interfaz al intentar compilar la aplicación.

- Integrar trazas obtenidas a partir de la ejecución del programa MPI. “generateTrace.sh”

```
#!/bin/bash
echo Generate Trace
cat *simulation_mpi_* | sort > traceComplete.txt #Order by timeStamp
echo Deleting Traces simulation_mpi_
rm *simulation_mpi_* #Delete traces of each rank
```

Este Script se encarga de integrar todas las trazas obtenidas en la ejecución MPI en un solo fichero “traceComplete.txt” y ordenarlo con el comando “sort”, que ordena por orden numérico y alfabético cada linea de la traza, de este modo obtenemos las tareas en orden temporal respecto a su ejecución.

- Ejecución de aplicación MPI. “executeMPI.sh“

```
#!/bin/bash
echo Execute MPI File

file=$1 #Name of exec mpi file
workers=$2 # Num of workers
params=$3 #params
pathMPI=$4 #Path of MPI library
paramSplit=" "

export PATH=$PATH:$pathMPI

for param in $(echo $params | tr "," "\n")
do
      echo param leido: $param
      paramSplit=$paramSplit$param
      paramSplit=$paramSplit" "
done

echo Try execute MPI MPIfile: $file , workers: $workers , params:
$paramSplit

mpiexec -np  $workers $file $paramSplit
```

Este Script es el encargado de ejecutar la aplicación MPI. Recibe el número de Workers que se van a lanzar en la ejecución, el fichero “file” ejecutable con la aplicación MPI y los parámetros necesarios para su ejecución. En el caso de recibir parámetros, estos se envían separados por comas y son recorridos uno a uno para formar el comando completo y poder ejecutar la aplicación. Tanto el número de Workers y los parámetros son solicitados al usuario a través de la interfaz.

- Ejecución de entorno SIMCAN

```
#!/bin/bash
echo Execute SIMCAN

./run_release -u Cmdenv

echo The simulation has finished
```

Este Script es el encargado de ejecutar la simulación SIMCAN.

4. Experimentos

En esta sección se describen los experimentos realizados utilizando el sistema propuesto en este proyecto. Para ello se ha utilizado una aplicación MPI que consiste en filtrar una imagen de forma paralela. La imagen es abierta y leída por el proceso *Master*, esta imagen se reparte entre los *Workers.* Cada uno filtra su porción de imagen y la vuelva a enviar al proceso *Master*. Por último el proceso master escribe, cada porción recibida, en la imagen filtrada.

Cabe destacar que este programa ha sido una de las prácticas de la asignatura Programación de Sistemas Distribuidos, con lo que se quiere demostrar su posible uso académico.

Inicialmente la aplicación se ha ejecutado en un ordenador convencional para obtener las trazas. En la Figura 4.1 vemos las prestaciones hardware del ordenador utilizado.

```
        Mackbook Pro Mid 2014
Procesador: Intel Core i5
Velocidad del procesador: 2,6 GHz
Memoria Ram: 8GB
Almacenamiento: SSD 512GB
      Escritura: 446 MB/S (3568 Mb/s)
      Lectura: 763 MB/s (6104 Mb/s)
```

Figura 4.1 Prestaciones hardware Macbook Pro

Estas prestaciones hardware se han replicado en los nodos que componen los entornos distribuidos simulados que se han modelado.

Las ejecuciones se han llevado a cabo con 4, 8 y 16 procesos, obteniendo las trazas correspondientes. Seguidamente se han modelado distintos entornos distribuidos utilizando SIMCAN. La figura 4.2 muestra la configuración de cada nodo de cómputo.

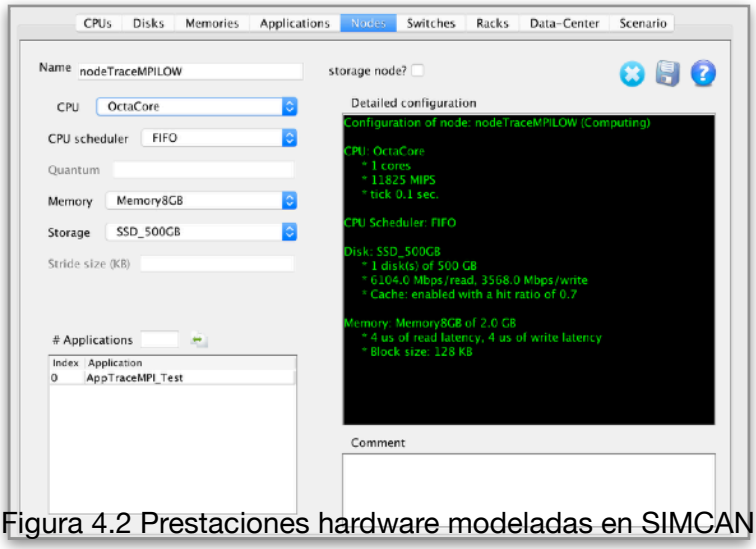

Figura 4.2 Prestaciones hardware modeladas en SIMCAN

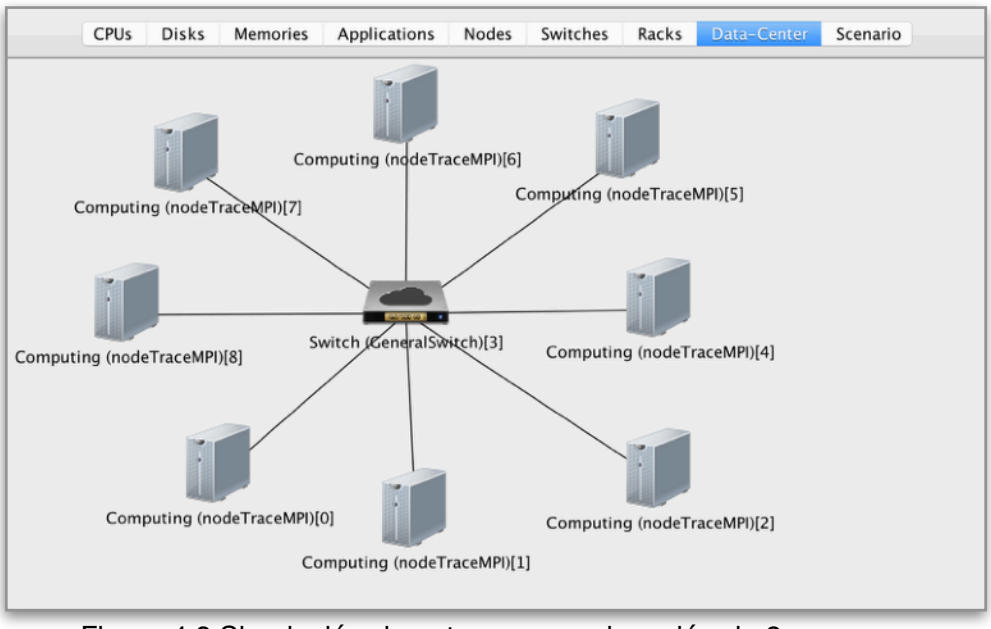

Figura 4.3 Simulación de entorno para ejecución de 8 procesos

La figura 4.3 muestra una de las configuraciones del entorno distribuido, en este caso para 8 procesos. Cada entorno utiliza 3 tipos de redes de forma que podremos observar cómo estas configuraciones afectan al rendimiento general de la aplicación.

Las simulaciones se mostrarán ordenadas, de mayor a menor, por el tiempo de ejecución total, por lo que, la configuración de red de 10Mb será la primera en ser explicada y tomada cómo referencia hasta llegar a la ejecución más rápida, en el entorno real.

Cabe destacar que, con el fin de poder mostrar claramente los resultados, la representación de las aplicaciones ha sido adaptada. Por ello, la representación de cada ejecución se acota a la anchura de la ventana de la interfaz. Con ello buscamos dar una visualización global y clara de las ejecuciones y las diferencias entre ellas.

A modo de aclaración, se detallan los significados de cada color.

- En color verde se observan las operaciones MPI_Recv.
- En color naranja las operaciones de CPU y de espera.
- En color turquesa la operación read.
- En gris las operaciones MPI_Send.
- En azul las comunicaciones con origen en el proceso Master.
- En fucsia las comunicaciones con el proceso Master como destino.
- En violeta la operación de close.

4.1.Ejecución de aplicación con 4 Procesos

4.1.1.Red de 10Mb

Figura 4.4 Representación gráfica de ejecución de 4 procesos con red de 10Mb

En primer lugar, la figura 4.4 muestra la simulación de la aplicación MPI con 4 procesos. En ella observamos las distintas comunicaciones y tareas ejecutadas. Como observamos en el visor de tareas, el tiempo necesario para ejecutar la aplicación es de 57 segundos aproximadamente. Esta red hace uso de un ancho de banda de 10Mb.

4.1.2.Red de 100Mb

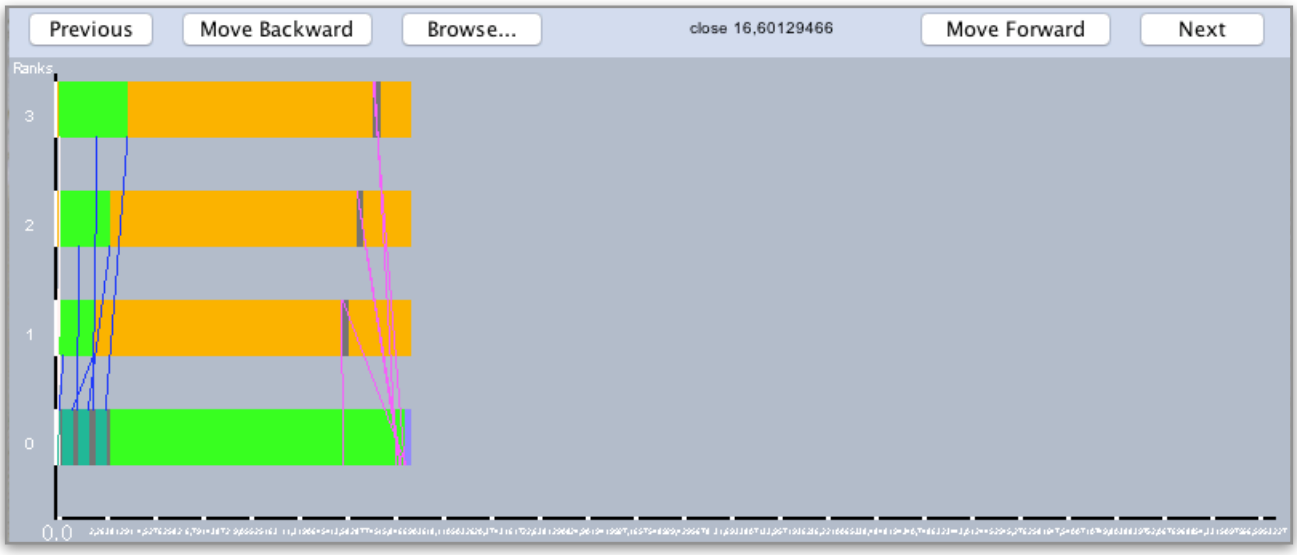

Figura 4.5 Representación gráfica de ejecución de 4 procesos con red de 100Mb

En este caso, se simula la aplicación con una red de 100Mb de ancho de banda. Como se puede observar, el tiempo total se reduce notablemente gracias al aumento de ancho de banda de la red, en este caso hasta los 16 segundos aproximadamente. El tiempo de procesamiento de cada *Worker* sigue siendo el mismo.

4.1.3.Red de 1Gb

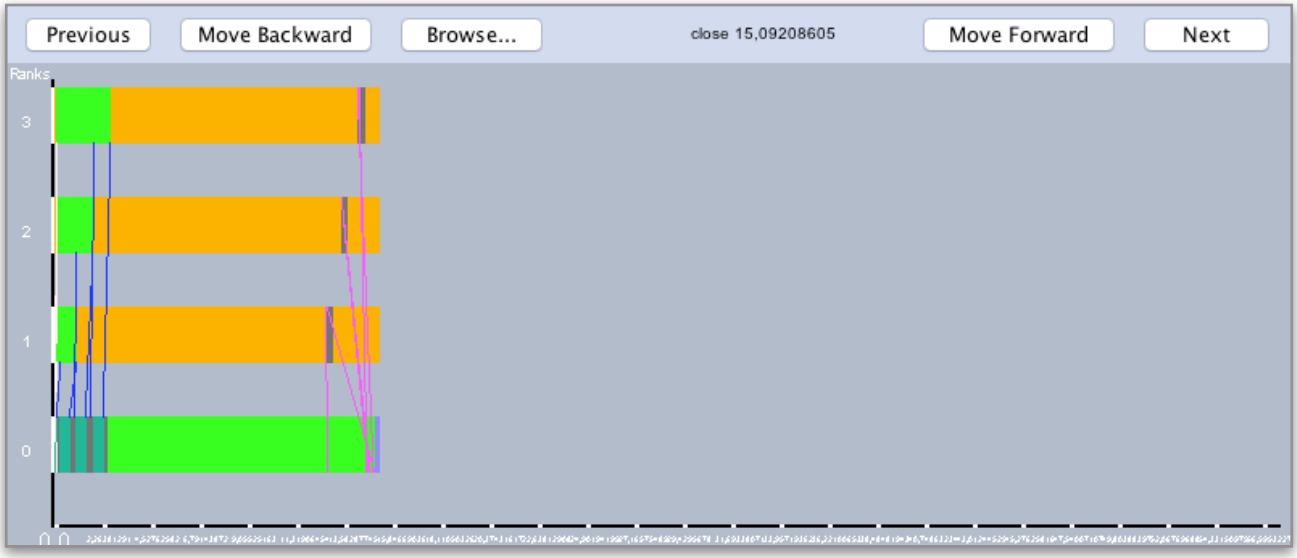

Figura 4.6 Representación gráfica de ejecución de 4 procesos con red de 1Gb

En este caso se simula la aplicación con una red de 1Gb de ancho de banda. Como se observa, al igual que en la red de 100Mb, la diferencia de tiempo total es considerable con respecto a la ejecución en una red de 10Mb. En este caso el tiempo total es de, aproximadamente 15 segundos. Si comparamos tiempos entre la red de 100Mb y 1Gb se observa que el tiempo es prácticamente similar. Esto se debe a que, en este caso, a partir de 100Mb, el ancho de banda apenas influye en la ejecución de la aplicación. En la figura 4.7 se observa de manera más detallada la comparación entre estas dos configuraciones tomando como referencia, para la comparación, la configuración de 100Mb.

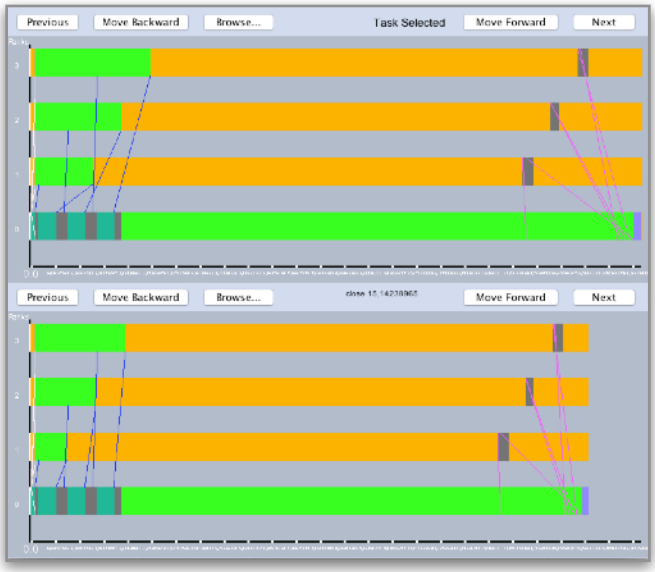

Figura 4.7 Comparación de ejecuciones en entornos de 100Mb y 1Gb

4.1.4.Entorno Real

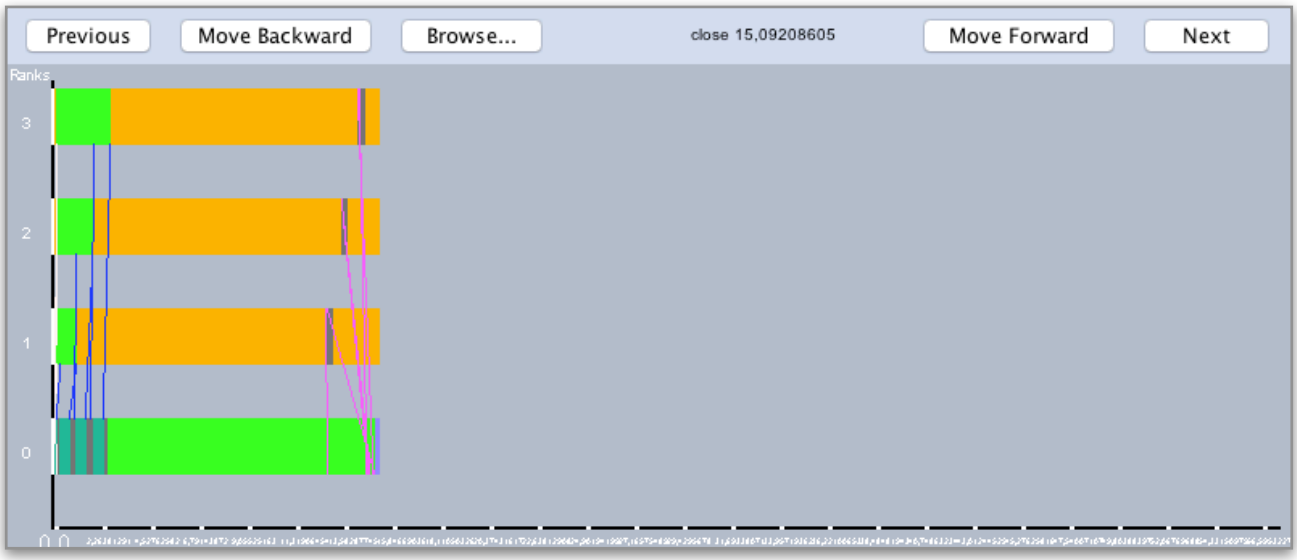

Figura 4.7 Ejecución de aplicación con 4 procesos en entorno real.

Esta ejecución, mostrada en la Figura 4.7, se lleva a cabo en un ordenador convencional, con la configuración citada en la introducción de este capítulo. En este caso, observamos que el tiempo total es prácticamente el mismo que la ejecución en una red de 1Gb. Esto se debe a que al ser una ejecución en entorno local, el envío de datos a otros procesos conlleva un tiempo muy corto, al igual que en una red con gran ancho de banda.

En conclusión, observamos que al ejecutar una aplicación con 4 procesos, se obtienen unos tiempos mínimos de 15 segundos, de los cuales más de 12 segundos son dedicados al procesamiento de la imagen. Todos los tiempos que superen estos 15 segundos serán a causa de la configuración de red usada.

4.2. Ejecución de aplicación con 8 procesos

En este caso observaremos como los tiempos de computo globales serán menores que con una ejecución dividida en 4 procesos. Esto se debe a que cada proceso recibe una porción de menor tamaño, por lo que el tiempo de procesamiento de la imagen es menor a medida que se aumenta el número de procesos.

4.2.1.Red de 10Mb

Figura 4.8 Ejecución de aplicación con 8 procesos en una red de 10Mb

Al ejecutar la aplicación con más procesos, observamos que el tiempo total de ejecución desciende, hasta los 50 segundos aproximadamente, a comparación de los 57 segundos al ejecutarla con 4 procesos. Esta mejora principalmente se debe al tiempo de procesamiento de imagen de cada *Worker*

4.2.2.Red de 100Mb

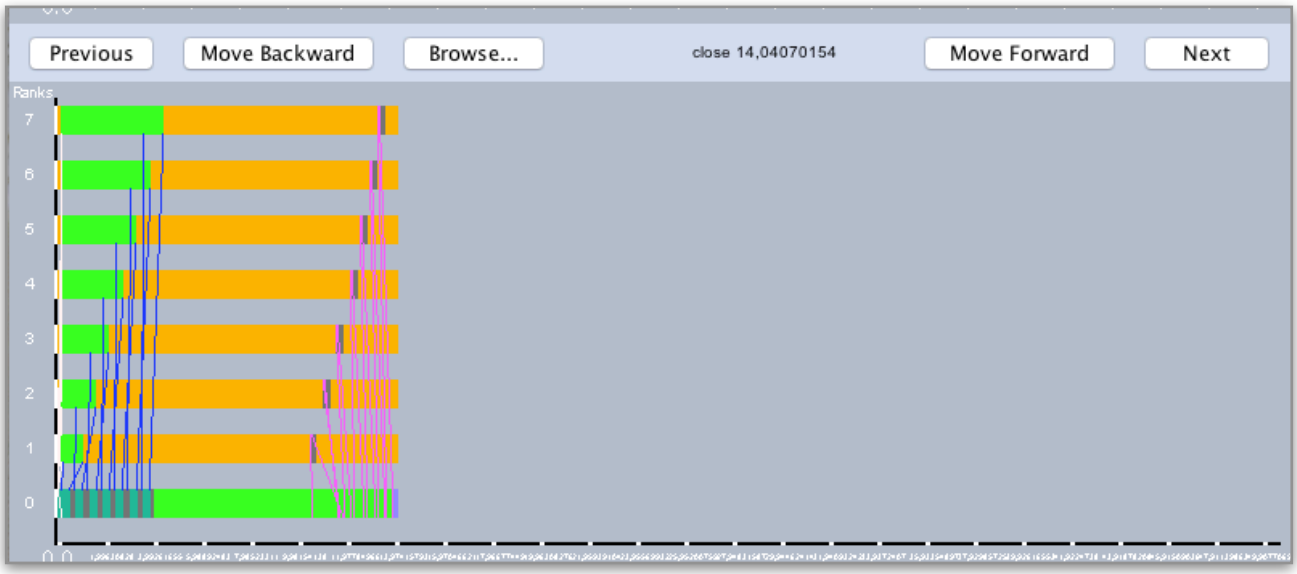

Figura 4.9 Representación gráfica de ejecución de 8 procesos con red de 100Mb

Una vez más, observamos la diferencia de tiempo al pasar a una red de 100Mb. Los tiempos de procesamiento se mantienen y los tiempos de envío de datos mejora notablemente la ejecución. Con ello obtenemos un tiempo total de 14 segundos aproximadamente.

4.2.3.Red de 1Gb

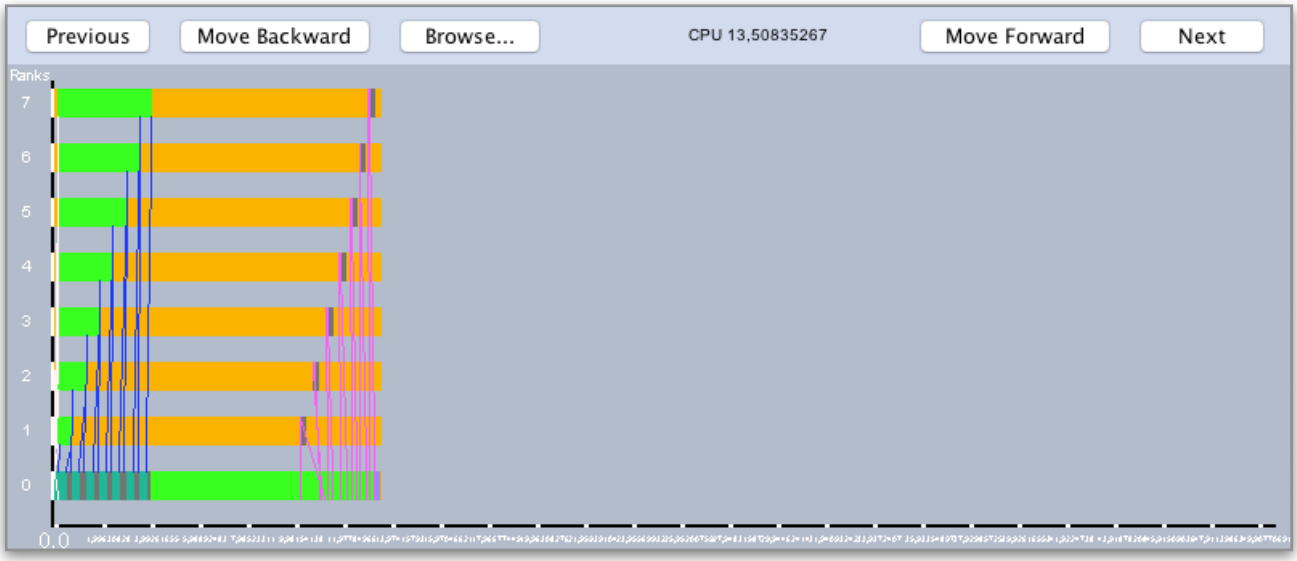

Figura 4.10 Representación gráfica de ejecución de 8 procesos con red de 1Gb

Al usar una configuración de una red de 1Gb observamos, en la Figura 4.10, como el tiempo total se reduce mínimamente con respecto a la red de 100Mb

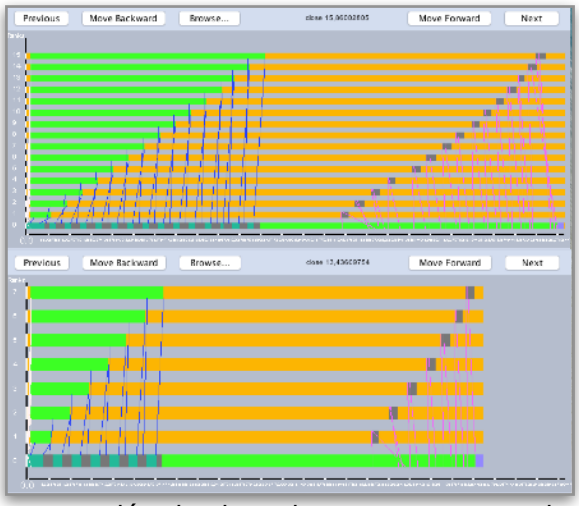

Figura 4.11 Comparación de ejecuciones en entornos de 100Mb y 1Gb

De manera detallada, en la Figura 4.11 comparamos estas dos últimas ejecuciones. Claramente comprobamos que la red a partir de 100Mb no influye en los tiempos.

4.2.4.Entorno real.

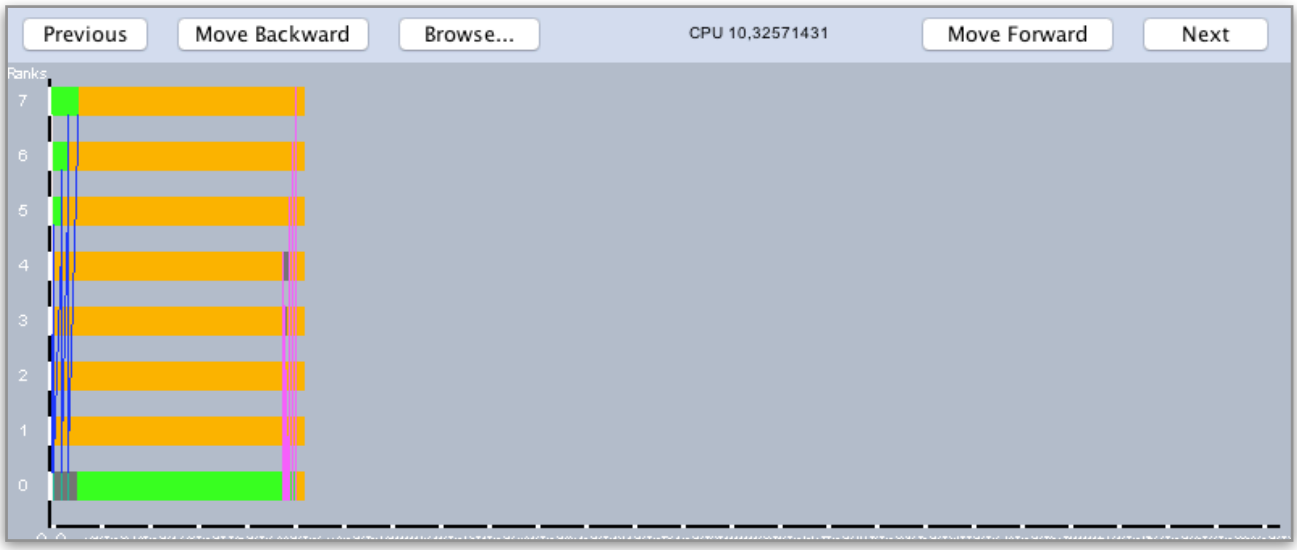

Figura 4.12 Ejecución de aplicación con 8 procesos en entorno real.

AL ejecutar la aplicación en el entorno real, se obtiene un tiempo 10 segundos aproximadamente, de los cuales 8 son de procesamiento de imagen en cada *Worker*

4.3.Ejecución de aplicación con 16 procesos

4.3.1.Red de 10Mb

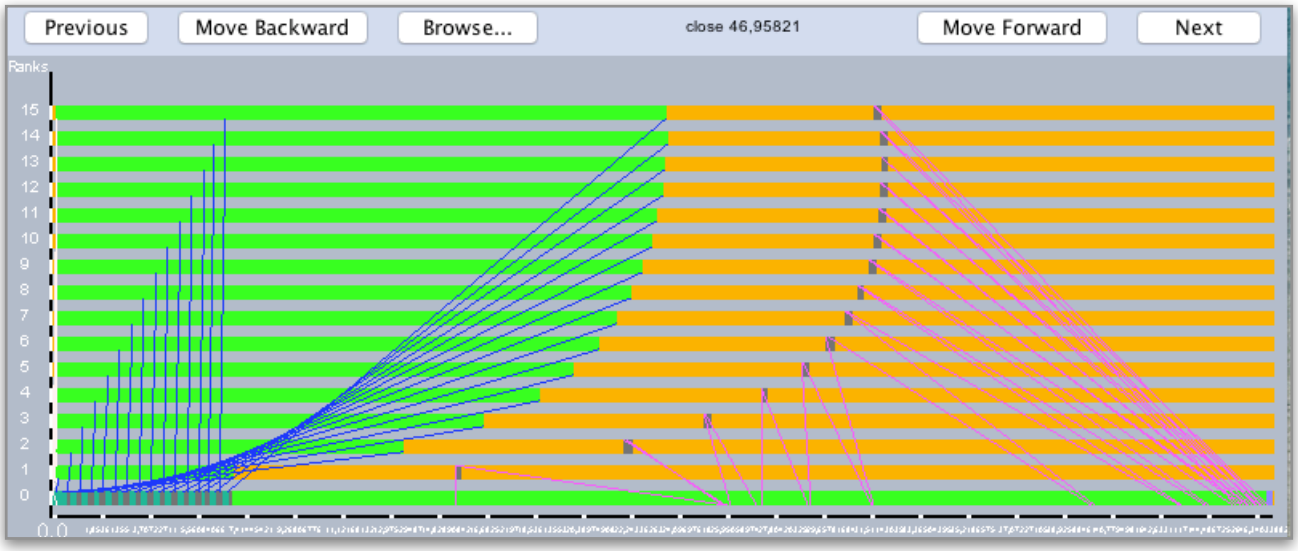

Figura 4.13 Ejecución de aplicación con 16 procesos en una red de 10Mb

Esta vez, se usan 16 procesos para el filtrado de la imagen. Los tiempos totales mejoran, pasando de 57 segundos iniciales con 4 procesos a, aproximadamente, 46 segundos. Una mejoría de mas de 11 segundos, aunque con solo 4 segundos de diferencia respecto a los 8 procesos.

4.3.2.Red de 100Mb

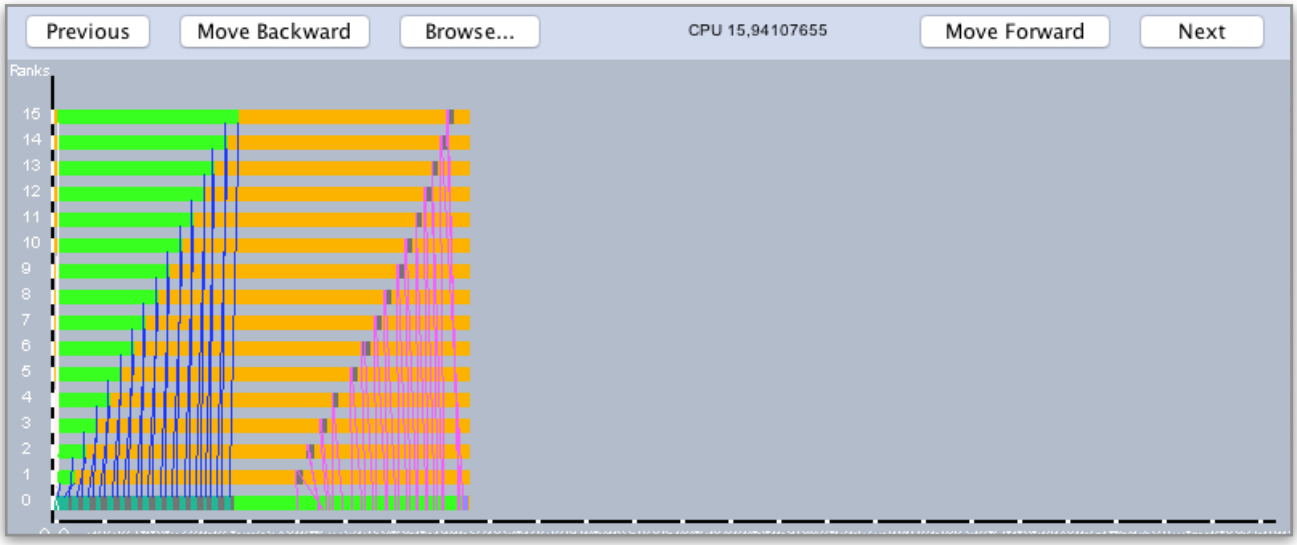

Figura 4.14 Representación gráfica de ejecución de 16 procesos con red de 100Mb

4.3.3.Red de 1Gb

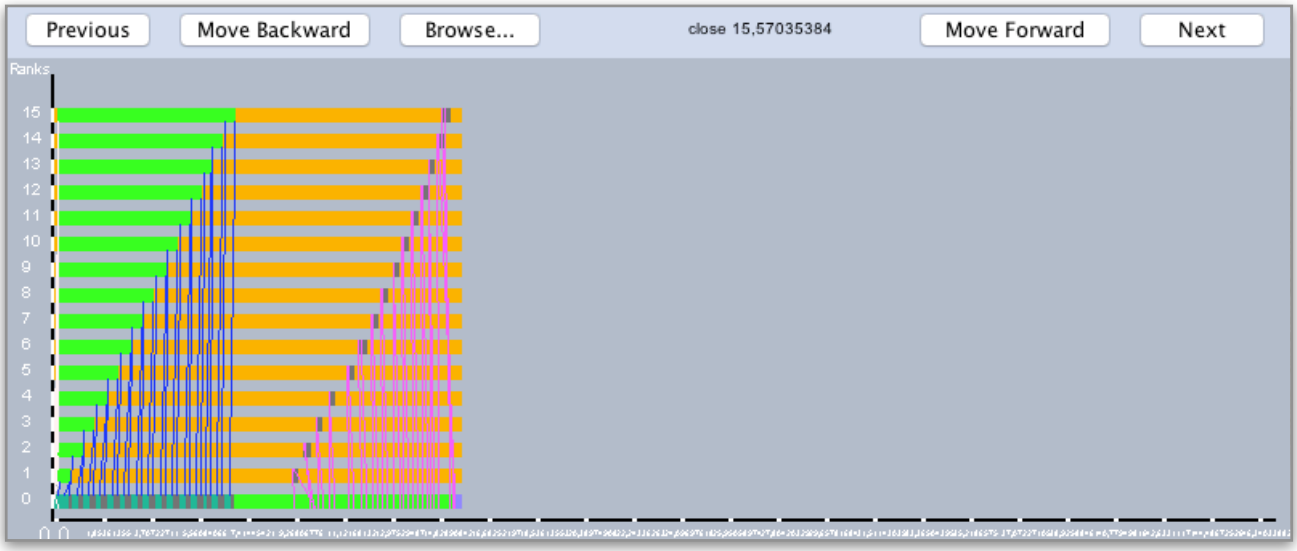

Figura 4.15 Representación gráfica de ejecución de 16 procesos con red de 1Gb

Observamos como en este caso, las redes de 100Mb y 1Gb tienen diferencias de tiempo de menos de medio segundo. Por lo que no es determinante usar una u otra configuración.

Figura 4.16 Comparación de ejecuciones en entornos de 100Mb y 1Gb

4.3.4.Entorno real

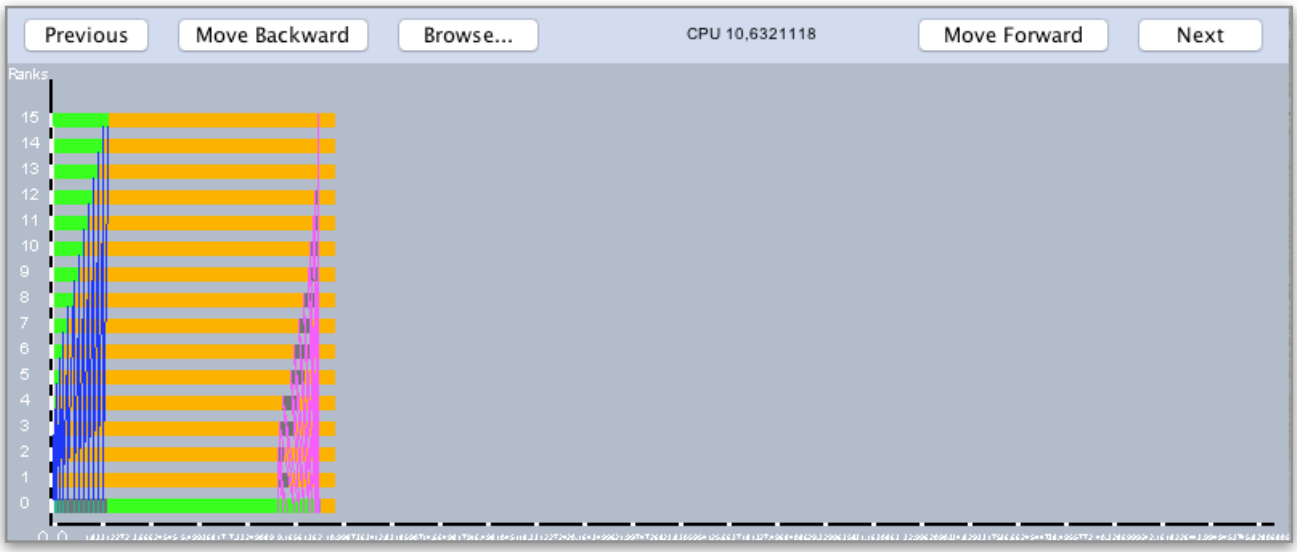

Figura 4.17 Ejecución de aplicación con 16 procesos en entorno real.

Al ver la ejecución de 16 procesos en entorno real observamos que el margen de mejora entre una red de 1Gb y un entorno local es mínimo ya que en entorno real el tiempo es de 10 segundos y en una red de 1G, el tiempo es de 15 segundos.

5. Conclusiones y Trabajo futuro

En esta sección se presentan las conclusiones obtenidas tras el desarrollo del proyecto, así como el trabajo futuro.

5.1.Conclusiones

A lo largo del desarrollo de este proyecto se han cumplido los distintos objetivos propuestos al inicio del mismo, afrontando distintos obstáculos y buscando las mejores soluciones para cumplirlos de forma satisfactoria.

El principal objetivo, que consiste en simular y representar gráficamente las aplicaciones MPI desarrolladas por el usuario, se ha conseguido gracias al cumplimiento de los objetivos secundarios. Estos objetivos han sido:

- Desarrollo de una biblioteca que registre las llamadas invocadas por cada proceso en una aplicación MPI.
- Adaptar el simulador SIMCAN a la nueva entrada de datos necesaria para generar la simulación.
- Desarrollo de una aplicación que, principalmente, represente gráficamente los resultados obtenido a partir de ejecuciones reales o simuladas de programas MPI.

Estos objetivos se han cumplido creando una aplicación sencilla de usar, con una interfaz cercana al usuario, sin necesidad de complementos o instalaciones engorrosas que hagan que el usuario pierda el interés por la aplicación.

Como se comentó en el apartado 1.2, el principal alcance del proyecto es el uso académico del sistema propuesto. De esta forma si finalmente se consigue dar un uso académico a esta aplicación, habrá sido un absoluto éxito personal, ya que sentiría que una pequeña parte de mi dedicación en estos años ha servido para ayudar a futuros alumnos a “amenizar” el aprendizaje de esta tecnología.

Tras finalizar este proyecto podemos concluir que el software desarrollado es viable para su uso académico ya que ha sido posible emplear varias aplicaciones MPI, como cálculo distribuido de matrices o una de las prácticas desarrolladas en la asignatura de Programación de Sistemas Distribuidos la cual se imparte en el Grado de Ingeniería de Computadores de la Facultad de Informática en la Universidad Complutense de Madrid. Se han desarrollado varios entornos distribuidos y ha sido posible comparar el comportamiento de las aplicaciones tanto en los entornos simulados como en un entorno local.

5.2.Trabajo futuro

Existen varias mejoras posibles en la aplicación que pueden añadirse en un futuro. A continuación comentamos algunas de las posibles mejoras que se pueden plantear en un futuro.

- Transformación automática de aplicaciones MPI en C. Implementar un conversor con el objetivo de poder adaptar aplicaciones MPI originales a aplicaciones que hacen uso de la biblioteca *TraceLib* de manera automática. Con esta mejora facilitaríamos la funcionalidad de carga de aplicaciones MPI ya que los usuarios no necesitarían conocer el funcionamiento de la biblioteca, tan solo cargar su programa MPI original.

- Análisis de Aplicaciones científicas. Sería interesante llevar a cabo experimentos de aplicaciones que requieren ser ejecutadas en sistemas de alto rendimiento. Como por ejemplo aplicaciones relacionadas con cálculos de números primos, procesamiento de imágenes de gran tamaño, etc.

5. Conclusions and future work

This section presents the conclusions obtained after the development of the project, as well as the proposed future work to continue it.

5.1.Conclusions

Throughout the development of this project, the different objectives proposed at the beginning of the project have been met, facing different obstacles and searching for the best solutions to satisfactorily fulfill them.

The main objective, which consists in simulating and graphically representing the applications developed by the user, has been achieved thanks to the fulfillment of the secondary objectives. These objectives have been:

- Development of a library that records the calls invoked by each process in an MPI application.
- Adapt the SIMCAN simulator to represent the trace file generated in the previous fhase.
- Development of an application that, mainly, graphically represents the results obtained from real or simulated executions of MPI programs.

These objectives have been met by creating a simple application to use, with an interface close to the user, without the need for add-ons or cumbersome installations that make the user lose interest in the application.

As discussed in section 1.2, the main scope of the project is the academic use of the proposed system. Whence if this app is used in the future for academic purpose, it will have been an absolute personal success, since you would feel that a small part of my dedication in these years has served to help future students to "entertain" the learning of this technology.

After completing this project we can conclude that the software developed is viable for academic use since it has been possible to use several MPI applications, such as distributed matrix calculation or one of the practices developed in the Distributed Systems Programming subject taught in the Degree in Computer Engineering at the Computing Faculty of the Complutense University of Madrid. Several distributed environments have been developed and it has been possible to compare the behavior of applications in both simulated environments and in a local environment.

5.2.Future Work

There are several improvements in the application that can be included in the future.

- Automatic transformation of MPI applications in C. Implement a converter in order to be able to adapt original MPI applications to applications that use the *TraceLib* library automatically. With this improvement we would facilitate the load functionality of MPI applications since users would not need to know the functioning of the library, just load their original MPI program.

- Analysis of scientific applications. It would be interesting to demonstrate experiments of applications that need to be executed in high performance systems. As for example applications related to calculations of prime numbers, processing of large images, etc.

Bibliografía

Documentación MPI:

https://es.scribd.com/document/54101857/Fundamentos-MPI

https://www.open-mpi.org

http://lam-mpi.miscellaneousmirror.org

https://www.mpich.org

Documentación MPE:

http://www.mcs.anl.gov/research/projects/perfvis/

Documentación SIMCAN:

http://antares.sip.ucm.es/cana/simcan/about.html

https://www.omnetpp.org

Documentación SIMGRID:

http://simgrid.gforge.inria.fr

Documentación C:

http://www.stack.nl/~dimitri/doxygen/index.html

Documentación Bash:

http://www.stack.nl/~dimitri/doxygen/index.html

Documentación JSON:

http://www.stack.nl/~dimitri/doxygen/index.html

Documentación Maven:

https://maven.apache.org/guides/

Documentación JAVA:

https://www.oracle.com/technetwork/java/javase/documentation/index.html

https://docs.oracle.com/javase/tutorial/uiswing/components/index.html