Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Ingenieria del Software

VEF-Visor: Un profiler gráfico de MPI utilizando el framework de trazas VEF

Alumno:
D. Raúl Lumbreras Thau

Tutor:
D. Francisco José Andújar Muñoz

Agradecimientos

Me gustaría agradecer a mi pareja y familia por el constante apoyo, motivación y confianza depositada durante el grado. Agradecer también a mi tutor por la paciencia, apoyo y guía durante la realización de este proyecto.

Resumen

VEF Traces es un framework de código abierto que modela el tráfico MPI de aplicaciones reales sobre simuladores de redes de interconexión HPC (High Performance Computing). Las herramientas proporcionadas por el framework se dividen en *VEF-Prospector* y *VEF-TraceLib*; la primera de estas encargada de generar trazas VEF, que contienen toda la información de las comunicaciones de la aplicación, y la segunda, que se encarga de replicar las comunicaciones a partir de las trazas. Estas aplicaciones son utilizadas en la actualidad en proyectos de investigación nacionales y europeos, destacando entre ellos los proyectos europeos RED-SEA y DEEP-SEA.

El funcionamiento del framework comienza con la generación de las trazas, tras ejecutar el comando vmpirun, se ejecuta el programa MPI a través de las bibliotecas del framework generando así unos ficheros que posteriormente se mezclarán para obtener el modelo final del tráfico.

La casuística que soluciona este proyecto está referida a este último fichero. El modelo final presentaba un problema de legibilidad para los usuarios, dado que se trata de un solo fichero de texto plano que contiene todos los datos obtenidos por la ejecución. Este proyecto soluciona esto creando una interfaz visual capaz de leer e interpretar el contenido de las trazas, presentando a los usuarios una forma visual de analizar las comunicaciones de la aplicación MPI.

Abstract

VEF Traces is an open-source framework for modeling MPI communications on High-Performance Computing (HPC) interconnection network simulators. The tools provided by the framework can be divided into VEF-Prospector and VEF-TraceLib; the former is responsible for capturing the MPI communications and generating the VEF traces, and the latter is responsible for the replication of the communications on the interconnection network simulator. These applications are currently used in national and European research projects, with notable mentions including the european projects RED-SEA and DEEP-SEA.

The operation of the framework begins with the generation of the traces, after executing the vmpirun command, the MPI program is executed through the libraries of the framework thus generating files that later will be processed to obtain the final traffic model.

The problem with this framework is that the final generated model has poor readability for users. This project addresses this issue by creating a visual interface capable of reading and interpreting the content of the resulting file, providing users with visual and interactive tools for analyzing the communications of the MPI application.

Índice general

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

1.1. Contexto

VEF Traces [14] es un framework que reúne una serie de herramientas que permiten a los desarrolladores obtener modelos de carga de tráfico de aplicaciones MPI, para su posterior replicación en simuladores de redes de interconexión de altas prestaciones. Las trazas VEF generadas contienen el total de las comunicaciones que la aplicación MPI ha realizado durante su tiempo de ejecución, permitiendo así realizar modelos precisos y fiables.

Una de las herramientas destacables es VEF-Prospector [15], encargada de capturar las llamadas MPI de la aplicación y almacenarlas en ficheros de trazas con formato VEF. Esas trazas son luego procesadas por VEF-TraceLib [16], biblioteca encargada de leer las trazas y generar el modelo de tráfico de red.

VEF Traces se utiliza en varios proyectos de investigación a nivel europeo, como los proyectos RED-SEA [17] y DEEP-SEA [18]. El framework genera gran interés en proyectos que requieran computación de alto rendimiento (o por sus acrónimo en ingles, HPC, de *High Performance Computing*).

El problema que aborda este proyecto de fin de grado refiere a la legibilidad del modelo generado por el framework. Este se presenta a los usuarios como un fichero de texto plano, en el que cada una de sus líneas describe el comportamiento individual de cada una de las tareas que hayan sido registradas durante la ejecución. Para trazas que contengan miles de tareas, el formato resulta incómodo para su análisis, por lo que se ha desarrollado una interfaz gráfica que lee e interpreta el fichero de traza con el fin de brindar una mejor experiencia de usuario.

1.2. Motivación

El framework VEF Traces se desarrolló y publicó en el año 2015 y, sobre este, se han ido realizando diferentes iteraciones con el fin de mejorar y actualizar el funcionamiento.

El objetivo original de esta biblioteca era generar modelos de carga para investigadores en redes de interconexiones HPC, por lo que el sistema no se pensó inicialmente para desarrolladores. Durante el tiempo de uso del framework, se ha detectado la necesidad de crear un profiler que brinde apoyo a los desarrolladores de aplicaciones HPC en MPI, necesidad que se cubre con el desarrollo de este trabajo de fin de grado.

1.3. Objetivos

El principal objetivo de este TFG, es desarrollar un programa que muestre una representación visual del tráfico generado por una aplicación MPI tras su instrumentalización con la biblioteca VEF-Prospector. Los objetivos secundarios son los siguientes:

- Integrar el nuevo visor gráfico, VEF-Visor, dentro del framework.
- Utilizar C como lenguaje para el desarrollo del programa, ya que es el lenguaje en el que está escrito todo el framework.

1.4. Estructura de la memoria

Este documento se estructura de la siguiente forma:

Capítulo 1 Introducción: En este capítulo se describe de forma resumida el concepto del proyecto, la motivación tras su desarrollo, objetivos y estructura de la memoria.

Capítulo 2 Metodología y planificación: En este capítulo se describe cuál ha sido la metodología seguida en este proyecto, las fases en las que se ha dividido, planificación inicial, riesgos y presupuestos.

Capítulo 3 Fundamentos / Estado del Arte: En este capítulo se explicarán conceptos fundamentales sobre MPI, VEF Traces y GTK.

Capítulo 4 Tecnologías utilizadas: En este capítulo se detallarán cuáles han sido las herramientas y software utilizado para el desarrollo del proyecto.

Capítulo 5 Análisis y Diseño: En este capítulo se realizará un análisis exhaustivo sobre los requerimientos del proyecto y se presentará un diseño detallado de la solución propuesta.

Capítulo 6 Implementación y pruebas: En este capítulo se describirá la implementación del proyecto y los casos de prueba planteados para asegurar el correcto funcionamiento de este.

Capítulo 7 Conclusiones y mejoras: En este capítulo se detallarán las conclusiones finales y se realizarán propuestas de mejora a futuro.

Anexo A Manuales: En este anexo se incluyen la guía de instalación del proyecto y la guía de mantenimiento básico del mismo.

Anexo B Resumen de enlaces adicionales: En este anexo se incluyen enlaces útiles así como el repositorio en el que se aloja el proyecto.

Capítulo 2

Metodología y Planificación

2.1. Metodología

Debido al gran número de metodologías existentes para el desarrollo de un producto software, la elección de esta es algo complejo y a su vez determinante para cualquier proyecto. Dado que, en las etapas más tempranas de este proyecto, los únicos requisitos que estaban definidos eran los principales, se decidió realizar este proyecto usando un enfoque de diseño iterativo [19]. Esta metodología, cuyo origen se estima en la década de los 30, ve una de sus primeras aplicaciones dentro del desarrollo software en el "Project Mercury" [20] [21].

Se ha considerado esta metodología debido a que, cuando el desarrollo dio comienzo solo se contaba con una idea general sobre el producto final, permitiendo que con esta forma de trabajo, el proyecto avance de forma constante, entrando de forma cíclica en una etapa de requisitos, análisis, implementación, pruebas y evaluación.

Las ventajas del desarrollo iterativo que más destacan en este proyecto son [22]:

- El feedback que se obtiene durante el desarrollo es constante, permitiendo que el cliente tenga un impacto continuo sobre el producto.
- El impacto de los riesgos se reduce, ya que durante los ciclos de desarrollo se detectan muchos de los riesgos antes de que lleguen a manifestarse completamente.
- Este enfoque permite una mayor flexibilidad, adaptándose mejor a los cambios que pueden sufrir los requisitos durante las etapas del desarrollo.

En cuanto a las desventajas, nos encontramos con las siguientes:

- El desarrollo resulta más costoso en tiempo, dado que cada ciclo implica realizar todas las fases que lo componen.

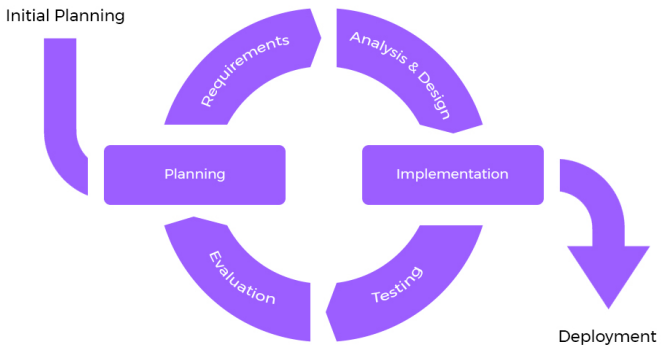

Figura 2.1: Diagrama desarrollo iterativo [1]

- La definición de los riesgos es más complicada al no contar con un plan inicial firme sobre el que realizar el análisis.
- Por la misma naturaleza de la metodología, dar fin a este tipo de proyectos resulta más complejo, ya que al no contar con unos requisitos iniciales estrictos, el producto puede entrar en un bucle de cambios constantes.

2.2. Planificación

El trabajo de fin de grado tiene una carga de trabajo estimada en unos 12 ECTS, siendo cada uno de estos equivalente a 25 horas. Con este cálculo, estimamos que la realización del proyecto debería ser aproximadamente de unas 300 horas [23].

El proyecto está estimado para desarrollarse durante los meses de Octubre de 2023 a Enero de 2024. Se ha dividido la carga de trabajo en ciclos de desarrollo de 20 horas cada uno, repartidos en 4 horas diarias de lunes a viernes. Esto se repetirá durante las primeras 10 semanas, en las cuales se dará lugar el desarrollo de la aplicación. Tras ello, se dedicarán las 6 semanas restantes a la redacción y documentación de esta memoria, dando un total estimado de 320 horas de trabajo.

Tarea	**Duración**	**Comienzo**	**Fin**
Documentación inicial			
VEF Traces	12 horas	02/08/2023	04/08/2023
Estudio de mercado	4 horas	05/08/2023	05/08/2023
Instalación de software	4 horas	06/08/2023	06/08/2023
Iteración desarrollo			
Planificación	2 horas	Lunes	Lunes
Requisitos	2 horas	Lunes	Lunes
Análisis y diseño	4 horas	Martes	Martes
Implementación	8 horas	Miércoles	Jueves
Testing y evaluación	4 horas	Viernes	Viernes
Finalización del proyecto			
Redacción y entrega del TFG	120 horas	18/12/2023	26/01/2024

Tabla 2.1: Planificación del proyecto

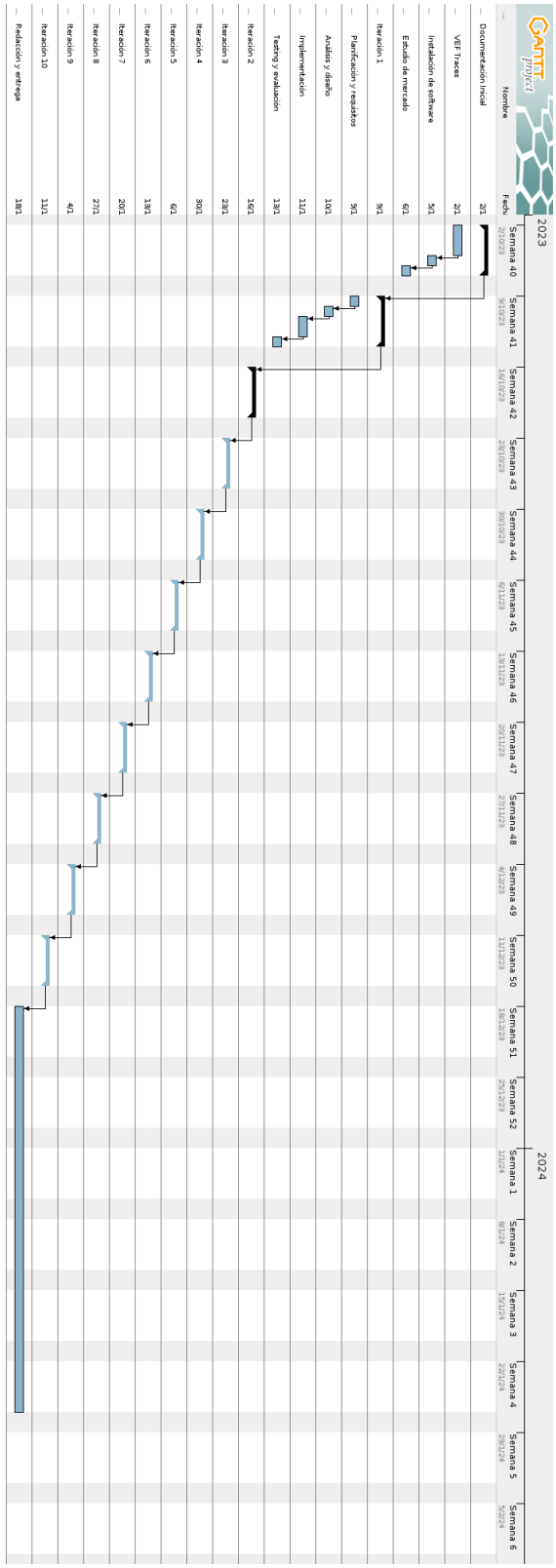

Figura 2.2: Diagrama de Gantt

2.3. Análisis de riesgos

La planificación de riesgos es un elemento crucial dentro de una buena gestión de proyectos, su objetivo es identificar, analizar y valorar los posibles problemas que podamos encontrar durante el tiempo de vida del proyecto y puedan afectar a su éxito; dentro de la planificación de riesgos también se deben crear estrategias para su prevención, mitigación o respuesta. A continuación, se muestra una lista de todos los riesgos planteados que se ha considerado puedan afectar a la consecución del proyecto. El formato seguido para cada riesgo es el siguiente:

- **ID:** identificador único asociado a cada riesgo.
- **Nombre:** nombre breve pero descriptivo, para facilitar la identificación del riesgo.
- **Descripción:** descripción detallada, incluyendo información relevante como el origen o alcance.
- **Categoría:** categoría en la que se puede englobar el riesgo (hardware, software, formación, personal, etc.).
- **Vulnerabilidad:** posible fallo o debilidad en el sistema, proceso o activo que posibilita la manifestación del riesgo.
- **Amenaza:** es el suceso que puede provocar el riesgo.
- **Probabilidad:** probabilidad con la que puede ocurrir la amenaza que provocaría el riesgo.
- **Impacto:** efecto que tendría la manifestación del riesgo.
- **Riesgo Total:** es el resultado entre la gravedad de la manifestación del riesgo y la probabilidad de que este ocurra.
- **Acciones de mitigación:** medidas preventivas que se pueden aplicar para minimizar la probabilidad de que el riesgo se manifieste, o reducir su impacto en caso de que ocurra.
- **Acciones correctivas:** medidas correctivas aplicadas tras la manifestación del riesgo, para recuperar el correcto desarrollo del proyecto y prevenir que el riesgo se repita.

		Probabilidad		
		BAJA	**MEDIA**	**ALTA**
Impacto	**BAJO**	BAJO	BAJO	MEDIO
	MEDIO	BAJO	MEDIO	ALTA
	ALTO	MEDIO	ALTA	ALTA

Figura 2.3: Tabla de cálculo del riesgo total

RSK01	Falta de tiempo
Descripción	La estimación inicial de tiempo resulta insuficiente.
Categoría	Formación.
Vulnerabilidad	Los plazos de entrega se incumplen.
Amenaza	La inexperiencia realizando la planificación de un proyecto puede provocar malas estimaciones.
Probabilidad	Media.
Impacto	Bajo.
Riesgo Total	Bajo.
Acciones de mitigación	
Consultar al Tutor y mejorar la planificación. Crear un prototipo del proyecto con cierta funcionalidad sin implementar.	
Acciones correctivas	
Redefinir el alcance del proyecto.	

Tabla 2.2: Riesgo 1: "Falta de tiempo"

RSK02	Cambios en el alcance
Descripción	Un nuevo enfoque en el proyecto, provocado por nuevas ideas o alguna problemática.
Categoría	Software.
Vulnerabilidad	Ambigüedad en los límites del proyecto.
Amenaza	Se pueden manifestar retrasos en el plazo de entrega y aumentos de costes.
Probabilidad	Media.
Impacto	Medio.
Riesgo Total	Medio.
Acciones de mitigación	
Implementar una metodología para el desarrollo adecuada. Mantener un proceso de comunicación adecuado entre el Tutor y el alumno.	
Acciones correctivas	
Aumentar los recursos asignados al proyecto y/o corregir la planificación de manera correspondiente.	

Tabla 2.3: Riesgo 2: "Cambios en el alcance"

RSK03	Pérdida de progreso
Descripción	Pérdida de los archivos relacionados con el TFG.
Categoría	Hardware.
Vulnerabilidad	Algún fallo en los discos duros ocasiona la pérdida de datos.
Amenaza	Se pierden todos los avances realizados del proyecto.
Probabilidad	Baja.
Impacto	Alto.
Riesgo Total	Medio.
Acciones de mitigación	
Utilizar software que permita almacenar el proyecto en la nube. Realizar copias de seguridad recurrentemente en discos duros externos.	
Acciones correctivas	
Aumentar el plazo de entrega del proyecto. Buscar maneras alternativas para la recuperación de los datos.	

Tabla 2.4: Riesgo 3: "Pérdida de progreso"

RSK04	Enfermedad
Descripción	El alumno contrae una enfermedad durante el plazo de desarrollo del TFG.
Categoría	Personal.
Vulnerabilidad	La enfermedad del alumno impide la realización de avances en el proyecto.
Amenaza	Retrasos en los plazos de entrega.
Probabilidad	Baja.
Impacto	Bajo.
Riesgo Total	Bajo.
Acciones de mitigación	
Mantener un correcto cuidado de la salud durante el tiempo de desarrollo del proyecto.	
Acciones correctivas	
Aumentar el plazo de entrega del proyecto. Redefinir el alcance del proyecto.	

Tabla 2.5: Riesgo 4: "Enfermedad"

RSK05	Falta de recursos o información
Descripción	Se pueden presentar dificultades a la hora de encontrar información relevante al proyecto.
Categoría	Formación.
Vulnerabilidad	Inexperiencia a la hora de sentar bases solidas en un proyecto de más alcance.
Amenaza	Una mala información puede provocar la pérdida de calidad en el resultado final.
Probabilidad	Media.
Impacto	Bajo.
Riesgo Total	Bajo.
Acciones de mitigación	
Realizar una mayor labor de investigación para encontrar más y mejores fuentes. Utilizar portales de información contrastados y reputados.	
Acciones correctivas	
Utilizar diferentes fuentes.	

Tabla 2.6: Riesgo 5: "Falta de recursos o información"

RSK06	Plagio o fraude académico
Descripción	Se puede cometer plagio o fraude en caso de que el alumno utilice material sin dar la debida atribución o al presentar trabajo como suyo sin serlo.
Categoría	Personal.
Vulnerabilidad	Pérdida de validez y confianza en el proyecto.
Amenaza	Los casos de plagio o fraude pueden conllevar graves consecuencias, llegando incluso a la expulsión de la entidad universitaria.
Probabilidad	Baja.
Impacto	Alto.
Riesgo Total	Medio.
Acciones de mitigación	
Seguir adecuadamente el reglamento de plagio y fraude de la entidad universitaria. Utilizar herramientas antiplagio para asegurar que no se den problemas.	
Acciones correctivas	
Encontrar y corregir aquellas secciones que incurran en plagio o fraude académico.	

Tabla 2.7: Riesgo 6: "Plagio o fraude académico"

RSK07	Falta de comunicación con el Tutor
Descripción	Se pueden dar complicaciones a la hora de establecer una comunicación adecuada con el Tutor.
Categoría	Personal.
Vulnerabilidad	Pérdida de retroalimentación y ambiguedad en el alcance del TFG.
Amenaza	La falta de comunicación puede provocar retrasos en la realización del proyecto, asi como la pérdida de su calidad.
Probabilidad	Baja.
Impacto	Alto.
Riesgo Total	Medio.
Acciones de mitigación	
Establecer en común un calendario para facilitar citarse. Mantener el contacto por varias vías de comunicación.	
Acciones correctivas	
Retomar lo antes posible la comunicación y reactivar el intercambio de información.	

Tabla 2.8: Riesgo 7: "Falta de comunicación con el Tutor"

RSK08	Baja calidad
Descripción	El trabajo presentado no cumple con los estándares de calidad planteados en la planificación inicial.
Categoría	Personal.
Vulnerabilidad	Reducción en la nota final o imposibilidad en la presentación del TFG.
Amenaza	La baja calidad puede terminar requiriendo de un aumento en los costes para ponerle solución.
Probabilidad	Media.
Impacto	Medio.
Riesgo Total	Medio.
Acciones de mitigación	
Tener suficiente plazo durante la finalización del proyecto y su entrega.	
Acciones correctivas	
Establecer claramente los puntos de mejora y solucionarlos de forma efectiva.	

Tabla 2.9: Riesgo 8: "Baja calidad"

2.4. Estimación de presupuesto

La realización del análisis presupuestario es algo imprescindible en cualquier proyecto. Durante este, se contabilizarán factores como el coste de las herramientas empleadas, el personal del proyecto, el tiempo de duración del proyecto, etc. Este análisis también nos permitirá tener una mejor visión y comprensión del tamaño y alcance del proyecto.

A continuación se detallará un desglose de los costes estimados para la realización de este trabajo:

- **Software:** para la realización de este trabajo de fin de grado, no se ha empleado ningún software que implique un costo por su licencia.
- **Hardware:** el único componente de hardware necesario para la realización del proyecto ha sido el ordenado personal del alumno, cuyo coste incluyendo periféricos ronda los 1200€.
- **Energético:** debido a que todo el desarrollo se ha realizado empleando dispositivos electrónicos, deberemos considerar también el gasto de electricidad asociado a estos. Teniendo en cuenta que, en ningún momento del desarrollo se ha expuesto al ordenador a cargas de trabajo exhaustivas, asumimos que el consumo promedio ronda los 250 Wh. A un promedio de 0,17742€/ kWh [24] tenemos que:

Fase	Tiempo de Uso (h)	Total (€)
Documentación	20	0,8871
Desarrollo	200	8,871
Finalización	120	5,3226
Total	320	15,0807

Tabla 2.10: Estimación coste energético

- **Internet:** durante los meses en los que se desarrolla en proyecto, se debe contar con una conexión a internet constante. En este caso, la tarifa son 20€ mensuales, por lo que, en los 4 meses que dura el desarrollo el coste aproximado sería de 80€.
- **Personal:** para el cálculo de los gastos de personal, se asumirá que el alumno ejerce las labores de un desarrollador junior.

  El alumno realizará trabajo durante las 320 horas estimadas inicialmente y, para el cálculo del coste asociado, tomaremos el salario medio de un desarrollador junior en Valladolid, siendo este de 12,97 €/h [25].

Rol	Tiempo empleado (h)	Total (€)
Estudiante (junior)	320	4150,4
Total	-	4150,4

Tabla 2.11: Estimación coste personal

2.4.1. Presupuesto final

Una vez analizados todos los costes, el total estimado que supondría la realización de este proyecto sería el indicado en la tabla siguiente:

Concepto	Total (€)
Software	0
Hardware	1200
Energético	15,08
Internet	80
Personal	4150,4
Total	5445,48

Tabla 2.12: Estimación coste total

Capítulo 3

Fundamentos/Estado del Arte

Durante este capítulo, entraremos en detalle sobre los conceptos fundamentales y necesarios para la correcta comprensión de este proyecto.

3.1. MPI

MPI (Message Passing Interface) [26] es una especificación estándar para la programación paralela y distribuida, diseñada para facilitar la comunicación entre procesos en entornos de computación de alto rendimiento . Se encarga de definir la sintaxis y semántica de una biblioteca de paso de mensajes diseñada para aprovechar la existencia de múltiples procesadores de forma simultánea.

El desarrollo de MPI se remonta a la década de 1980 como respuesta a la creciente demanda de estándares para la programación paralela en sistemas de alto rendimiento. Con el avance de la computación paralela, MPI evolucionó hasta que, en 1994, se presenta la primera versión oficial del estándar, que sigue en desarrollo hasta el día de hoy [26].

MPI se basa en el modelo de comunicación entre procesos, donde diferentes procesos en ejecución realizan un intercambio de mensajes para coordinar y realizar tareas. Estos procesos pueden residir dentro de una misma máquina o estar distribuidos dentro de múltiples nodos en un clúster.

La interfaz define puntos de comunicación mediante los cuales los procesos pueden realizar intercambios de información. Estos puntos incluyen funciones para el envío y recepción de mensajes, sincronización de procesos y gestión de grupos de procesos.

Siguiendo el modelo SPMD (Single Program, Multiple Data), el usuario deberá escribir su aplicación como un proceso secuencial del que se lanzarán varias instancias que se comunican y cooperan entre sí [13]. Los diferentes modos de comunicación incluidos dentro de MPI pueden diferenciarse en dos grupos:

- **Punto a punto:** involucran únicamente a dos procesos, y estas pueden realizarse de forma bloqueante o no bloqueante, es decir, deteniendo uno de los procesos hasta que la operación solicitada finalice o permitiendo que el proceso continué su ejecución con normalidad; también pueden realizarse de forma síncrona o asíncrona.

- **Colectiva:** una operación colectiva debe ser invocada por todos los procesos de un grupo, aunque los roles que jueguen no tienen por qué ser los mismos. La mayoría de estas operaciones requieren la designación de un proceso como raíz de la operación.

MPI permite a los desarrolladores organizar los procesos en grupos dentro de las aplicaciones para facilitar la coordinación y administrar eficientemente la comunicación entre subconjuntos específicos de procesos.

Todas las funciones MPI comienzan por el literal "MPI_". Por ejemplo, si queremos implementar una función de envío de mensaje entre un proceso y otro, utilizaríamos la función "MPI_Send". Las secciones del programa que implementen la interfaz, deben comenzar obligatoriamente con la función "MPI_Init" y terminar con la función "MPI_Finalize" [27].

Dentro de las comunicaciones MPI, se utilizan tipos de datos propios. Esto se debe a que los procesadores que participen en una aplicación MPI no tienen por qué ser todos iguales, lo que podría provocar diferencias en la representación de los mismos datos en diferentes máquinas. Por este motivo, MPI realiza transformaciones de sintaxis que posibilitan la comunicación entre los proceso en entornos heterogéneos. En la Tabla 3.1 se pueden observar los principales tipos de datos primitivos de MPI y su equivalente en C. Destacar también que el programador podrá definir tipos MPI derivados en función de las necesidades particulares del proyecto.

Tipos MPI	**Tipos C**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	Sin equivalente

Tabla 3.1: Tipos de datos MPI [13]

3.2. VEF Traces

VEF Traces [28] es un framework *open source* que incluye una serie de herramientas utilizadas para facilitar la creación de modelos de comunicaciones en aplicaciones MPI y su posterior reproducción en un simulador de redes de interconexión HPC. Podemos separar la funcionalidad del framework en dos herramientas principales, VEF-Prospector y VEF-TraceLib, la primera encargada de capturar el tráfico de red y generar las trazas necesarias y la segunda encargada de reproducir el tráfico generado por la herramienta anterior, aunque entraremos en más detalle de cada una de ellas a continuación.

3.2.1. VEF-Prospector

VEF-Prospector [15] es el paquete utilizado para generar trazas en formato VEF de una aplicación MPI. La aplicación entra dentro de la categoría de herramientas de instrumentación intermedia, esto significa que la librería se coloca entre la aplicación MPI y la propia librería MPI, interceptando las llamadas y recogiendo los diferentes datos necesarios para la creación del modelo.

De manera más detallada, VEF-Prospector captura todas las llamadas a las funciones MPI, interactuando con el driver MPI usando la interfaz de *profiling* PMPI (Profiling MPI), que se encarga de ejecutar finalmente las llamadas. Mientras, VEF-Prospector genera una serie de ficheros binarios temporales que contienen los datos asociados a cada una de las comunicaciones.

Podemos diferenciar estos archivos temporales en dos tipos:

- **Ficheros .veft:** Contienen todas las llamadas MPI realizadas por un proceso MPI, y la información más relevante sobre cada llamada, como marcas temporales de inicio y fin, el proceso de destino, el tamaño y tipo de comunicación MPI, etc.
- **Ficheros .comm:** Contienen todos los comunicadores MPI utilizados por un proceso MPI durante la ejecución de la aplicación.

Tras generar esta primera versión de las trazas, se realizar la mezcla de todas ellas para generar la traza final. De la realización de esta tarea se encarga la herramienta VEF Mixer, que realiza una lectura de todos los archivos temporales y los ordena, generando un fichero que contiene toda la información requerida para reproducir el modelo de comunicaciones de la aplicación MPI. En la figura 3.1 se puede observar un diagrama de la obtención de las trazas.

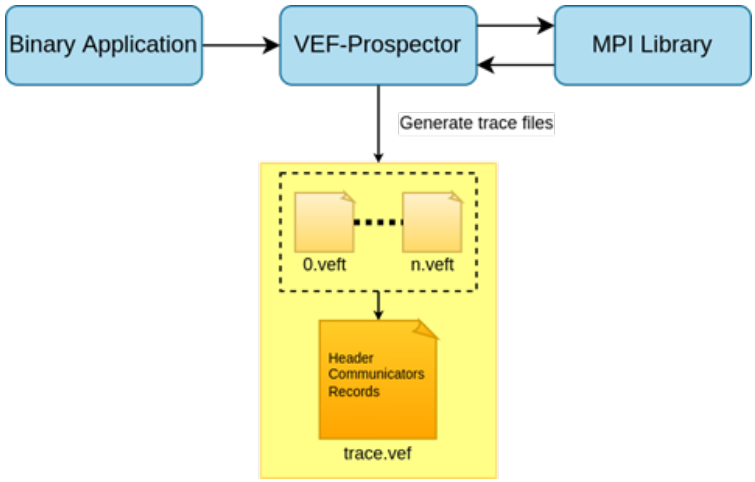

Figura 3.1: Procedimiento de adquisición de trazas VEF [2]

Dentro de VEF-Prospector también se incluyen más herramientas, explicadas a continuación:

- **vef_tmp_reader:** permite a los usuarios la lectura de los ficheros temporales.
- **vef2_updater:** actualiza las trazas VEF del formato VEF2 al formato VEF3.
- **prospector_debugger:** empleado únicamente por desarrolladores del framework, no es necesario su uso para obtener las trazas.
- **repasaVEF:** realiza la lectura de una traza VEF y describe su tráfico en diferentes ficheros.
- **vmpirun:** es el script empleado para ejecutar aplicaciones MPI utilizando la librería, se encarga de precargar la librería de instrumentalización y, posteriormente, ejecutar el comando mpirun con la aplicación del usuario.

3.2.2. Formato de trazas VEF

Una traza VEF [14] contiene toda la información necesaria para modelar la comunicación generada durante la ejecución de una aplicación MPI. Cada traza se compone de numerosas entradas, cada una de ellas representando una sola comunicación. Se pueden distinguir tres tipos de entradas dentro de las trazas VEF:

- **Cabecera:** representan la primera línea dentro del fichero de trazas e incluye información general sobre la ejecución de la aplicación MPI, siendo su formato el siguiente:

VEF *nNodes nMsgs nCOMM nCollComm nLocalCollComm noRecvDep clock*

Se procede a explicar el significado de cada uno de estos términos:

- **VEF3** indica el formato de la traza VEF.
- *nNodes* es el número total de tareas MPI.
- *nMsgs* es el número total de comunicaciones punto a punto.
- *nCOMM* es el número total de comunicadores.
- *nCollComm* es el número total de comunicaciones colectivas.
- *nLocalCollComm* es el número total de comunicaciones colectivas locales, es decir, vistas desde el punto de vista de los procesos y no de la aplicación. Por ejemplo, una comunicación que involucre cuatro procesos implica una sola comunicación colectiva global, pero cuatro comunicaciones colectivas locales.
- *noRecvDep* es un parámetro en desuso mantenido por compatibilidad con versiones anteriores de las trazas.
- *clock* es el menor incremento del valor del reloj del sistema, medido en picosegundos.

- **Comunicadores:** un comunicador especifica las tareas MPI involucradas dentro de un intercambio de mensajes generado por una comunicación colectiva. Una tarea MPI puede aprovecharse de varios comunicadores para comunicarse con múltiples grupos de tareas. Se construyen basadas en el siguiente formato:

$$C0\ element_0\ [element_1 \dots element_{n-1}]$$

Se procede a explicar el significado de cada uno de estos términos:

- *C0* es el identificador del comunicador, compuesto por el carácter "C" seguido de un número natural único.
- $element_0\ [element_1 \dots element_{n-1}]$ son los identificadores de cada una de las tareas agrupadas dentro del comunicador.

- **Comunicaciones:** contienen los detalles sobre la comunicación generada por la tarea MPI y todas las relaciones de dependencia con el resto de tareas. Las entradas para comunicaciones punto a punto y colectivas se registran en un formato diferente:

  - Punto a punto: las comunicaciones punto a punto modelan el intercambio de mensajes individuales entre dos tareas MPI. Se construyen basadas en el siguiente formato:

$$ID\ src\ dst\ length\ Dep\ dTime\ IDdep$$

    Se procede a explicar el significado de cada uno de estos términos:

    - *ID* es el identificador único de mensaje. Es importante aclara que los identificadores no aparecen en orden creciente en la traza, pero todos los identificadores de las comunicaciones de una tarea MPI sí siguen un orden estrictamente creciente.
    - *src* es la tarea MPI de origen.

- *dst* es la tarea MPI de destino.
- *length* es el tamaño del mensaje en bytes. Si encontrásemos mensajes con tamaño cero, significaría que están modelando una comunicación del tipo "*MPI_Wait()*".
- *Dep* es el tipo de dependencia entre mensajes.
- *dTime* es el tiempo de dependencia del mensaje, medido en ciclos de reloj. El tiempo en segundos puede obtenerse utilizando este campo y periodo de reloj indicado en la cabecera.
- *IDdep* es el identificador del mensaje del que depende.

- Colectivas: las comunicaciones colectivas modelan el intercambio de varios mensajes entre grupos de tareas MPI. Se construyen basadas en el siguiente formato:

  ID comm op task sendBytes recvBytes Dep dTime IDdep

  Se procede a explicar el significado de cada uno de estos términos:

  - *ID* es el identificador único la comunicación colectiva. Para diferenciarlos de los identificadores de mensajes punto a punto, siempre comienzan con el carácter "G".
  - *comm* es el identificador del comunicador.
  - *op* es el tipo de comunicación colectiva realizada.
  - *task* es el identificador de la tarea MPI.
  - *sendBytes* es número de bytes enviados en la comunicación.
  - *recvBytes* es el número de bytes recibidos en la comunicación.
  - *Dep* es el tipo de dependencia.
  - *dTime* es el tiempo de dependencia del mensaje, medido en ciclos de reloj.
  - *IDdep* es el identificador del mensaje del que depende.

Para ambos tipos de comunicaciones, resulta imprescindible explicar el concepto de dependencia. Las trazas VEF son trazas *auto-relacionadas*. Es decir, la ejecución de una comunicación siempre depende de que se haya completado una comunicación anterior, ya sea el envío de un mensaje previo, la recepción de un mensaje o completar una comunicación colectiva. Así, el campo *Dep* indica el tipo de dependencia la actual comunicación tiene con la comunicación anterior indicada por *IDdep*, y la comunicación actual comenzará su ejecución *dTime* después de que se satisfaga la dependencia con la comunicación *IDdeo*

A continuación, se especifican los valores y tipos de dependencia que puede reflejar el campo *Dep*:

- 0 - Mensajes independientes: estos corresponden a las primeras llamadas MPI de la aplicación, estos mensajes no dependen de ningún otro.
- 1 - Dependencia de envío: estos mensajes deben ser procesador cuando una determinada comunicación punto a punto sea enviada por la propia tarea MPI.
- 2 - Dependencia de recibo: estos mensajes deben ser procesados cuando una determinada comunicación punto a punto sea recibida por la propia tarea MPI.

- 3 - Dependencia de comunicación colectiva: estos mensajes deben ser procesados cuando una determinada comunicación colectiva ejecutada por la propia tarea MPI haya terminado.
- 4 - Mensajes independientes y disparadores: estos mensajes se comportan de manera análoga a los "Mensajes independientes", pero son a mayores un disparador. Un mensaje es un disparador cuando el receptor tiene un mensaje que depende de la recepción de este mensaje.
- 5 - Envío y disparador: estos mensajes tienen dependencia de envío y un mensaje disparador.
- 6 - Recibo y disparador: estos mensajes tienen dependencia de recibo y un mensaje disparador.
- 7 - Dependencia de comunicación colectiva y disparador: estos mensajes tienen dependencia de comunicación colectiva y un mensaje disparador.

3.2.3. VEF-TraceLib

VEF-TraceLib [16] es el paquete encargado de reproducir el modelo de carga del tráfico de red en cualquier simulador de redes de interconexión HPC. Esta librería es capaz de generar una única simulación a partir de múltiples trazas, al igual que puede gestionar la asignación de tareas a diferentes nodos finales y gestionar la comunicación entre la librería y el simulador.

La gestión de las comunicaciones MPI se realiza de forma transparente, es decir, el simulador debe comunicarse con la librería para que esta le proporcione los mensajes almacenados en la traza. De forma detallada, el software de simulación obtendrá los mensajes de VEF-TraceLib, se encargara de inyectarlos en la red y delegará el control a la librería cuando los mensajes se hayan recibido en los nodos finales. Para cada uno de los mensajes generados, la librería solamente provee al simulador con las interfaces de red de origen y destino, la longitud del mensaje y su identificador, pero el simulador no obtiene los datos sobre el tipo de comunicación que ha generado el envío del mensaje (punto a punto o colectiva), la tarea y aplicación que ha generado el mensaje (en el nodo conectado a la interfaz de red pueden existir múltiples tareas de diferentes aplicaciones), o información sobre si una tarea se ha detenido esperando a la finalización de otra, delegando toda la gestión de los mensajes a nivel de aplicación a VEF-TraceLib.

VEF-TraceLib también incluye una serie de adicciones interesantes para los desarrolladores, como la posibilidad de simular múltiples trazas de forma simultanea, un sistema de mapeo de tareas/interfaz de red totalmente flexible, o la posibilidad de implementar sus comunicaciones colectivas propias.

Dentro de las numerosas aplicaciones que podemos encontrar para la librería podemos destacar algunas de las más relevantes:

- Comparación del rendimiento de diferentes redes que implementen el mismo modelo de tráfico.

- Analizar el impacto de las modificaciones realizadas a los modelos de red.
- Analizar el impacto en el rendimiento de diferentes distribuciones de las tareas MPI en los nodos de computo.
- Comparación del rendimiento de diferentes algoritmos de comunicación.

3.3. GTK

GTK (GIMP Toolkit) [29] es una biblioteca de herramientas para crear interfaces gráficas de usuario en aplicaciones. Desarrollada inicialmente para el proyecto GIMP (GNU Image Manipulation Program) [30], GTK se ha convertido en una librería altamente versátil utilizada en una gran variedad de proyectos, incluyendo entornos de escritorio como GNOME [31] o aplicaciones como la propia GIMP o Inkscape [32].

Se ha considerado añadir una sección sobre GTK ya que ha sido la librería sobre la que se ha construido la aplicación. Esto se debe a que una de los requisitos del proyecto fue que su código se escribiera en el lenguaje de programación C, un lenguaje que no cuenta con mucho desarrollo en el ámbito de las interfaces gráficas y, la librería de GTK, permite una integración completa con aplicaciones C. El requisito sobre el lenguaje de desarrollo proviene de que el proyecto completo se encuentra escrito en este lenguaje y era necesario que las labores de mantenimiento pudieran realizarse por desarrolladores de C.

3.3.1. Componentes básicos de GTK

En GTK, los elementos visuales de la interfaz reciben el nombre de “widgets”. Estos “widgets” pueden ser desde botones, cajas de texto, ventanas, barras de progreso, etc. En general, casi cualquier elemento que componga la interfaz final será o estará compuesto por widgets. Estos se crean desde el código de la aplicación y se organizan de forma jerárquica. A continuación se muestra la creación de un nuevo widget para un botón:

```
GtkWidget *genericButton = gtk_button_new_with_label("genericButton");
```

Los elementos de la interfaz se agrupan utilizando un tipo especial de “widget” llamados contenedor. Estos elementos permiten gestionar la posición de los diferentes “widgets” que contengan, usando el contenedor “GtkBox” podemos organizarlos por filas o columnas, mientras que usando “GtkGrid” podemos organizarlos por casillas de una cuadrícula. A continuación se muestra la creación de un nuevo contenedor del tipo “GtkBox”:

```
GtkWidget *genericBox = gtk_box_new(GTK_ORIENTATION_HORIZONTAL, 0);
```

Para realizar la conectividad entre los diferentes elementos que componen la interfaz, GTK se apoya en el uso de señales y “callbacks”. Cuando los “widgets” reciben cierto tipo

de eventos (clics de ratón, cambios en el texto, etc) emiten una señal. Esta señal se asocia con anterioridad a un "callback", el cual invocará la función deseada para dar respuesta a la señal. A continuación se muestra un ejemplo que enlaza una señal de clic en un botón con una función:

```
g_signal_connect(genericButton, "clicked", G_CALLBACK(function), app);
```

Es importante hablar también del ciclo de eventos. GTK implementa un ciclo interno en el que, tras realizar la carga inicial de la interfaz, se mantiene a la espera para poder reaccionar en tiempo real a las entradas que reciba de los usuarios. Esto permite que la interfaz tenga una respuesta fluida a los eventos que se produzcan durante la ejecución de la aplicación.

Por último, mencionar que todas las aplicaciones GTK cuentan con una función main que, como mínimo, crea la ventana principal y comienza el ciclo de eventos anteriormente mencionado. A continuación se muestra un ejemplo de función main que cumpliría con la funcionalidad mínima de una aplicación GTK.

```
int main(int argc, char **argv) {
    GtkApplication *app;
    int status;

    app = gtk_application_new("uva.example", G_APPLICATION_FLAGS_NONE);
    g_signal_connect(app, "activate", G_CALLBACK(activate), NULL);
    status = g_application_run(G_APPLICATION(app), argc, argv);
    g_object_unref(app);

    return status;
}
```

El cierre de la aplicación se realiza cuando se recibe la señal "destroy" sobre la ventana principal, comenzando el proceso de cierre ordenado de la aplicación y de cualquier proceso asociado a esta.

3.3.2. Cairo Graphics

Cairo [33] es una biblioteca de software libre para gráficos 2D con soporte para múltiples dispositivos de salida. Esta biblioteca se encuentra escrita en su totalidad en C, pero es capaz de dar soporte a múltiples lenguajes.

Se ha considerado importante añadir una sección sobre Cairo ya que, a través de GTK, es la biblioteca que ha permitido realizar la impresión de las trazas VEF en la interfaz gráfica, empleando un "widget" del tipo "GTK_DRAWING_AREA".

El funcionamiento básico de Cairo junto con GTK es el siguiente: tras instanciar el "widget" anteriormente mencionado, se debe conectar a una función de dibujado.

```
gtk_drawing_area_set_draw_func(GTK_DRAWING_AREA(drawingArea),
    draw_function, NULL, NULL);
```

Esta función será llamada cada vez que GTK necesite dibujar los contenidos del "widget" en la pantalla y deberá contener toda la lógica que sea necesaria para realizar esta impresión. Dicho de otra manera, realizaremos una conexión entre la señal de dibujado del "widget" y el "callback" será la función que designemos. Los eventos que pueden desencadenar una llamada a la señal de dibujado son numerosos, mostrar el "widget", realizar un redimensionado de este, invocarlo manualmente, etc.

En el área de dibujo, Cairo sitúa un puntero en las coordenadas (0,0), al cual se le deberán dar instrucciones para realizar la imagen que se necesite. Las instrucciones más básicas son:

- *cairo_move_to*: encargada de mover el puntero de dibujado a las coordenadas indicadas por parámetros.
- *cairo_line_to*: encargada de realizar una línea recta desde la posición del puntero hasta las coordenadas indicadas por parámetros.

Con este par de instrucciones podemos explicar el funcionamiento elemental de Cairo, primero utilizamos instrucciones para situar el puntero en las coordenadas adecuadas de la zona de dibujado y luego utilizamos instrucciones de dibujo para realizar la impresión.

A mayores de *cairo_line_to()*, existen otra serie de instrucciones de dibujo que permiten realizar formas más complejas, como podrían ser *cairo_rectangle()*, *cairo_curve_to()*, *cairo_arc()*, etc. También es importante indicar que un gran número de estas instrucciones cuentan con su versión relativa, la cuál nos permite el uso de coordenadas relativas a la posición del puntero en vez de absolutas, por ejemplo, *cairo_rel_line_to()*.

También existen instrucciones para el dibujo de texto (*cairo_text_path()*) o caracteres personalizados (*cairo_show_glyphs()*).

El puntero almacena todo este conjunto de instrucciones a mayores del resto de propiedades necesarias para el dibujado, como podrían ser el color de la línea, el grueso de esta o la fuente del texto en caso de que correspondiera.

Una vez se hayan indicado las instrucciones de dibujo correspondientes, debemos realizar la llamada a las funciones que realizarán la impresión por pantalla. Estas reciben como parámetro el puntero que ha almacenado toda la información del dibujo y son las encargadas de colorear los píxeles indicados en la zona de dibujado. La función de impresión más elemental es (*cairo_stroke()*).

Un ejemplo de una función de dibujado que imprima por pantalla un cuadrado vacío de dimensión 10 en las coordenadas (20,30) y un círculo con relleno de radio 15 en las coordenadas (50, 50) sería el siguiente:

```
static void draw(GtkDrawingArea *drawingArea, cairo_t *puntero,
    int width, int height) {
        //Damos valor a las propiedades de grueso y color de línea
        cairo_set_line_width(puntero, 2);
        cairo_set_source_rgb(puntero, 0.8, 0.8, 0.8);

        // Dibujar un cuadrado en las coordenadas (20, 30)
        cairo_rectangle(puntero, 20, 30, 10, 10);
        cairo_stroke(puntero);

        // Dibujar un círculo en las coordenadas (50, 50)
        cairo_arc(puntero, 50, 50, 15, 0, 2 * G_PI);
        cairo_fill(puntero);
}
/* ... */
static void activate(GtkApplication *app, gpointer user_data) {
/* ... */
        GtkWidget *drawingArea = gtk_drawing_area_new();
        //Enlazamos el área de dibujado con su función
        gtk_drawing_area_set_draw_func(GTK_DRAWING_AREA(drawingArea),
            draw, NULL, NULL);
/* ... */
}
```

Capítulo 4

Tecnologías utilizadas

Durante este capítulo, realizaremos un resumen del software que se ha utilizado durante toda el periodo de desarrollo de este proyecto de fin de grado.

4.1. C

C [34] es un lenguaje de programación de propósito general con características de economía de expresión, control de flujo y estructuras de datos modernas, y un conjunto rico de operadores. C no es un lenguaje de "muy alto nivel", ni uno "grande", y no está especializado en ninguna área particular de aplicación, pero su falta de restricciones y su generalidad lo hacen más conveniente y efectivo para muchas tareas que los lenguajes supuestamente más potentes.

C fue originalmente diseñado e implementado por Dennis Ritchie en el sistema operativo UNIX [35] en la computadora DEC PDP-11. El sistema operativo, el compilador de C y esencialmente todos los programas de aplicaciones de UNIX están escritos en C. También existen compiladores de producción para varias otras máquinas, incluyendo el IBM System/370, el Honeywell 6000 y el Interdata 8/32. Sin embargo, C no está vinculado a ningún hardware o sistema en particular, y es fácil escribir programas que se ejecutarán sin cambios en cualquier máquina que admita C.

Se ha utilizado como lenguaje de desarrollo para este proyecto. La Figura 4.1 muestra el logo del lenguaje C.

Figura 4.1: Logo de C [3]

4.2. Visual Studio Code

Visual Studio Code [36] es un editor de código fuente ligero pero eficaz que se ejecuta en el escritorio y está disponible para Windows, macOS y Linux. Incluye compatibilidad integrada con JavaScript, TypeScript y Node.js, y cuenta con un amplio ecosistema de extensiones para otros lenguajes y entorno de ejecución (como C++, C#, Java, Python, Go, .NET).

Se ha utilizado durante toda la etapa de desarrollo de este proyecto como principal editor de código. La Figura 4.2 muestra el logo de Visual Studio Code.

Figura 4.2: Logo de Visual Studio Code [4]

4.3. CMake

CMake [37] es una herramienta de código abierto que facilita el proceso de construcción, configuración y generación de proyectos de software. Su función principal es proporcionar una manera independiente del sistema operativo para describir el proceso de construcción de un programa, lo que significa que los desarrolladores pueden escribir instrucciones de construcción en un formato específico de CMake y luego generar archivos de configuración nativos para plataformas como Make, Visual Studio o Xcode, entre otros.

Esta herramienta ya se encontraba integrada dentro del proyecto VEF-Prospector, por lo que se ha utilizado para realizar la integración de la aplicación VEF-Visor dentro del mismo. La Figura 4.3 muestra el logo de CMake.

Figura 4.3: Logo de CMake [5]

4.4. GanttProject

GanttProject [38] es una aplicación de código abierto diseñada para la gestión de proyectos y la creación de diagramas de Gantt, una herramienta visual que ayuda a planificar y seguir el progreso de las tareas a lo largo del tiempo en un proyecto. Esta herramienta se utiliza comúnmente en campos como la gestión de proyectos, la ingeniería, la construcción, y otros entornos donde es esencial una planificación estructurada y un seguimiento eficiente de las actividades.

Se ha utilizado para la realización del diagrama de Gantt (ver Figura 2.2) durante la etapa de planificación de este proyecto. La Figura 4.4 muestra el logo de GanttProject.

Figura 4.4: Logo de GanttProject [6]

4.5. Lucidchart

Lucidchart [39] es una herramienta en línea para la creación de diagramas y visualización de información que se utiliza comúnmente en entornos empresariales y educativos. Permite a los usuarios diseñar una amplia variedad de diagramas, desde organigramas y diagramas de flujo hasta mapas mentales y esquemas de red. Su enfoque colaborativo y basado en la nube facilita la creación y edición conjunta de diagramas, lo que resulta beneficioso para equipos de trabajo distribuidos.

Se ha utilizado para la realización de todos los diagramas utilizados durante la etapa de análisis y diseño de este proyecto (ver Figuras 5.1, 5.2 y 5.3). La Figura 4.5 muestra el logo de Lucidchart.

Figura 4.5: Logo de GanttProject [7]

4.6. GTK

GTK (GIMP Toolkit) [29] es una biblioteca de herramientas para crear interfaces gráficas de usuario en aplicaciones. Para más detalle sobre esta herramienta, consultar la Sección 3.3.

Se ha utilizado como biblioteca principal para la realización de la interfaz gráfica que compone la aplicación de este proyecto. La Figura 4.6 muestra el logo de GTK.

Figura 4.6: Logo de GTK [8]

4.7. Cairo

Cairo [33] es una biblioteca de software libre para gráficos 2D con soporte para múltiples dispositivos de salida. Para más detalle sobre esta herramienta, consultar la Sección 3.3.2.

Se ha utilizado como biblioteca de apoyo junto a GTK para realizar la impresión de las trazas dentro de los componentes de la interfaz. La Figura 4.7 muestra el logo de Cairo.

Figura 4.7: Logo de Cairo [9]

4.8. Overleaf

Overleaf [40] es una plataforma en línea colaborativa para la creación, edición y compilación de documentos científicos y académicos, especialmente diseñada para documentos

escritos en LaTeX [41]. LaTeX es un sistema de composición de texto de software libre, ampliamente utilizado en la comunidad académica para la creación de documentos técnicos, artículos científicos, tesis y otros tipos de textos que requieren una estructura y formato específicos.

Se ha utilizado como herramienta principal para la redacción de esta memoria. La Figura 4.8 muestra el logo de Overleaf.

Figura 4.8: Logo de Cairo [10]

4.9. Discord

Discord [42] es una plataforma en línea de mensajería instantánea, chat de voz y videollamada. Permite la gestión de comunidades o grupos a través de canales, admitiendo la subida de ficheros y facilitando la coordinación y comunicación entre los integrantes. Discord está disponible como aplicación para los principales sistemas operativos, aunque cuenta también con una versión web.

Se ha utilizado como herramienta principal de comunicación con el tutor durante la realización de este proyecto, empleando el chat de texto para comunicaciones más rápidas o la videollamada para realizar reuniones periódicas. La Figura 4.9 muestra el logo de Discord.

Figura 4.9: Logo de Discord [11]

4.10. GitLab

GitLab [43] es una plataforma completa de gestión del ciclo de vida de desarrollo de software (Systems Development Life Cycle) que integra funcionalidades de control de versiones, seguimiento de problemas, integración continua, despliegue continuo y colaboración en un solo lugar.

Se ha utilizado como como repositorio principal para alojar la aplicación desarrollada en este proyecto, ya que las herramientas de VEF Traces se encuentran alojadas en esta plataforma. La Figura 4.10 muestra el logo de GitLab.

Figura 4.10: Logo de GitLab [12]

Capítulo 5

Análisis y Diseño

5.1. Análisis

Durante el periodo de análisis del proyecto, se ha realizado un estudio acerca de las funcionalidades que debería implementar VEF-Visor. Estas funcionalidades se dividen entre requisitos funcionales y no funcionales. Basado en esos requisitos, se plantea también un diagrama de actividad, mostrado en la Figura 5.1, que describe el flujo de las interacciones entre el usuario y el sistema.

Notar que, al ser este un proyecto que se va a realizar en su totalidad en el lenguaje de programación C, el cual no es un lenguaje de programación con orientación a objetos, no se han incluido muchos de los diagramas UML (Unified Modeling Language) que de otra manera serían habituales en la etapa de análisis de un proyecto, ya que carecerían de sentido en este contexto.

5.1.1. Requisitos Funcionales

A continuación, se describen los requisitos funcionales identificados durante la etapa de análisis:

- **RF1.** La aplicación deberá ser capaz de mostrar una representación visual de las comunicaciones MPI capturadas en los ficheros de profiling generados por la biblioteca VEF-Prospector.
- **RF2.** La aplicación deberá ser capaz de abrir los ficheros de profiling mediante la interfaz.
- **RF3.** La aplicación deberá mostrar las comunicaciones MPI diferenciadas por tareas MPI individuales.

- **RF4.** La aplicación deberá permitir realizar "zoom" sobre la representación visual de las comunicaciones MPI.
- **RF5.** La aplicación deberá permitir mostrar un momento temporal específico, siempre que esté comprendido en el rango de tiempo de ejecución de la aplicación MPI.
- **RF6.** La aplicación deberá poder mostrar los detalles sobre una comunicación MPI concreta cuando el usuario realice un clic de ratón sobre su representación visual.
- **RF7.** La aplicación deberá mostrar en pantalla el nombre de la aplicación leída, sus instantes de inicio y fin y la duración de la traza.
- **RF8.** La aplicación deberá permitir la lectura de una nueva traza sin la necesidad de su reapertura.

5.1.2. Requisitos no Funcionales

A continuación, se describen los requisitos no funcionales identificados durante la etapa de análisis:

- **RNF1.** La aplicación deberá ser desarrollada en el lenguaje de programación C.
- **RNF2.** La aplicación deberá poder trabajar con cualquier fichero de trazas, independientemente del tamaño de este.
- **RNF3.** La aplicación se desarrollará de manera que se faciliten las futuras labores de mantenimiento.
- **RNF4.** La aplicación deberá estar integrada dentro de la compilación de VEF-Prospector.
- **RNF5.** La aplicación deberá ser integrada en el repositorio git de VEF-Prospector [15].
- **RNF6.** El tiempo de respuesta de la aplicación no deberá exceder los 10 segundos en ninguno de los momentos de lectura de ficheros.
- **RNF7.** La aplicación deberá diferenciar los tipos de eventos representados mediante el uso de colores diferentes.
- **RNF8.** Cuando la aplicación muestre algún instante de tiempo, este deberá estar normalizado utilizando como base el tiempo de inicio global de la traza.

5.1.3. Diagrama de actividad

El diagrama de actividad que se muestra en la Figura 5.1 representa el flujo de respuestas e interacciones entre la aplicación y el usuario durante un uso habitual de la aplicación, es decir, la apertura de un fichero de trazas VEF, la interacción con la representación de las trazas y el posterior cerrado de la aplicación.

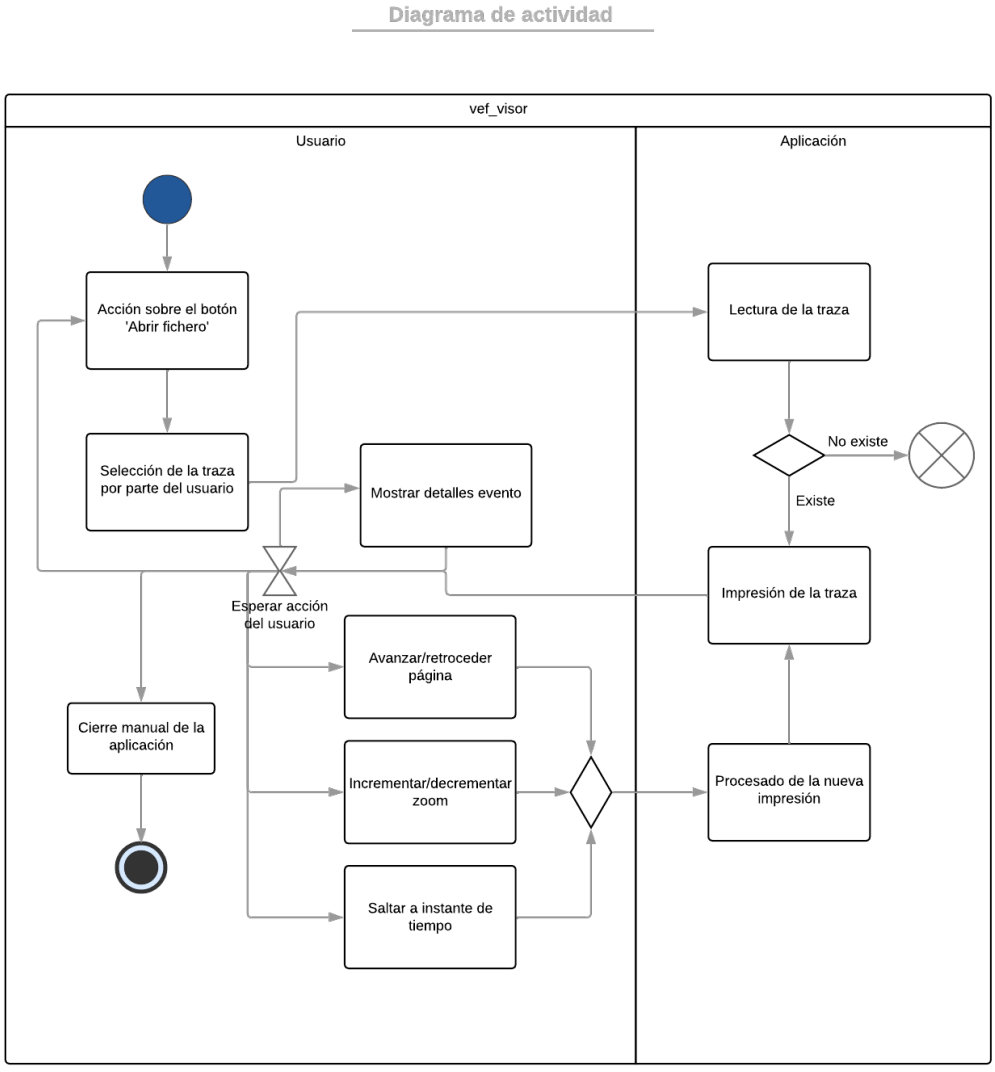

Figura 5.1: Diagrama de actividad

5.2. Diseño

En esta sección, se realizará la explicación de las decisiones de diseño tomadas durante el desarrollo del proyecto para poder cumplir con los requisitos identificados durante la etapa de análisis. Se explicará la arquitectura del sistema apoyándose en el diagrama de clases (ver Figura 5.2) y las dependencias de la aplicación mediante el diagrama de paquetes (ver Figura 5.3).

5.2.1. Diagrama de clases

Se ha empleado el diagrama de clases (ver figura 5.2) para representar la relación que tiene la aplicación con el resto de "clases" incluidas en VEF-Prospector.

Para la realización de este diagrama, se ha tomado la libertad de realizar la equivalencia entre las clases de un lenguaje con orientación a objetos y los archivos de cabecera en C; aunque en la práctica no cumplen con el mismo rol, resulta útil para poder realizar una mejor explicación de la aplicación.

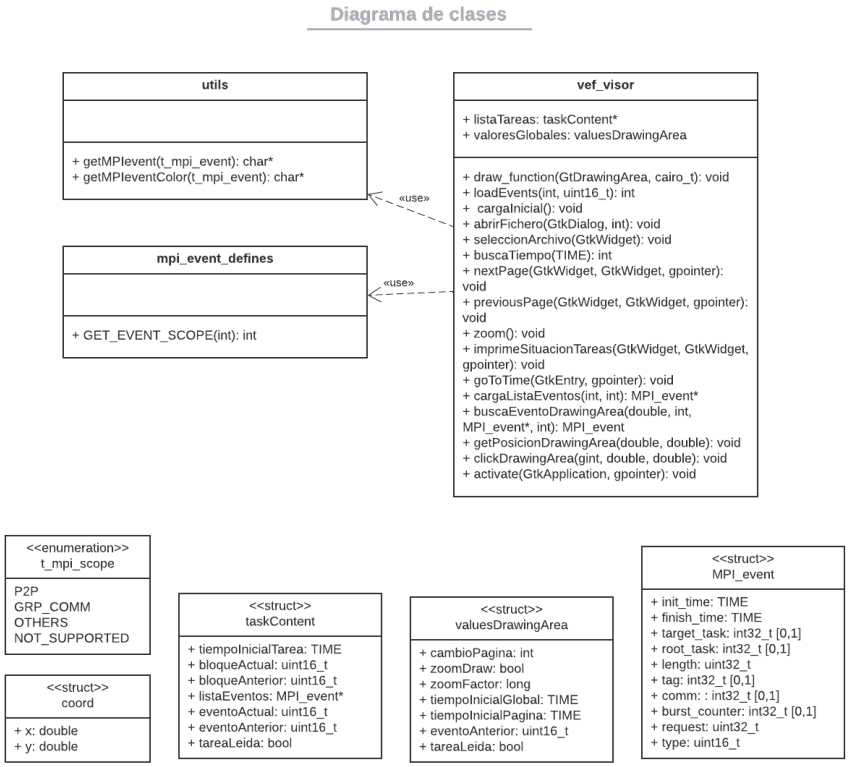

Figura 5.2: Diagrama de clases

Podemos observar como la "clase" *vef_visor* emplea diversas funciones repartidas entre las "clases" *utils* y *mpi_event_defines*. También se han descrito los diferentes "structs" y "enums" que se han creado para la realización de la aplicación o resultan relevantes para esta.

En las dos "clases", ambas incluidas en la aplicación principal, se puede ver como las funciones utilizadas son referidas a los eventos MPI, en este caso, a la obtención del tipo preciso de evento o a la obtención del tipo de mensaje (punto a punto o colectivo).

Además, se han detallado todas las operaciones dentro de *vef_visor*, junto con los parámetros de cada función y los valores de retorno.

5.2.2. Diagrama de paquetes

El diagrama de paquetes, mostrado en la Figura 5.3, permite observar la estructura interna dentro de la suite VEF-Prospector, en la cual la aplicación desarrollada en este proyecto se encuentra en un paquete diferente al de los ficheros de cabecera que se incluyen para obtener las funcionalidades antes explicadas.

También se indica la dependencia de la aplicación con la biblioteca GTK, que al ser una biblioteca de alto nivel se ha considerado representarla en un paquete externo al sistema.

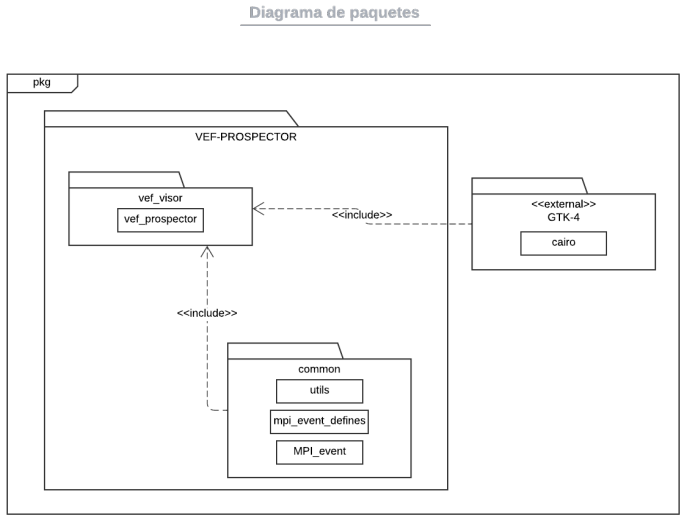

Figura 5.3: Diagrama de paquetes

Capítulo 6

Implementación y pruebas

En este capítulo, entraremos en detalle sobre la manera en la que se han implementado las diferentes funcionalidades de la aplicación, el uso que se ha dado a la biblioteca GTK [44] y los diferentes cursos de ejecución diseñados para el correcto funcionamiento del sistema.

También hablaremos sobre las pruebas a las que se ha sometido a la aplicación para garantizar su buen funcionamiento.

6.1. Funcionamiento general de la aplicación

Durante esta sección, explicaremos cuál es la funcionalidad general de la aplicación y como se distribuyen los componentes dentro de la pantalla.

6.1.1. Funcionalidad implementada

La aplicación VEF-Visor, es una aplicación de lectura y visualización de trazas VEF [14]. Para lograr este objetivo, la aplicación tiene que ser capaz de procesar los diferentes ficheros de trazas generados, procesarlos y mostrar una representación visual de ellos por pantalla.

El sistema está preparado también para navegar por las trazas, esto se debe a que en numerosos casos, las trazas que se deben representar alcanzan unas magnitudes de tiempo que imposibilitan que se vean representadas de un solo vistazo, por lo que permitir al usuario decidir el rango temporal que se muestra por pantalla resultaba imperativo.

Para ello, las trazas se dividen por franjas de tiempo, en las que el sistema muestra los eventos que hayan ocurrido durante ese rango. El usuario puede, utilizando botones, avanzar y retroceder entre esas franjas de tiempo para garantizar que pueda consultar la traza al completo.

Con el fin de aprovechar al máximo el espacio de representación de las franjas temporales, la zona en la que se imprimen los eventos de las trazas cuenta con una funcionalidad de scroll vertical e horizontal. Esto no solo permite que cada vez que se dibuje en la pantalla la cantidad de información mostrada sea máxima, sino que sin esta funcionalidad, las trazas que cuenten con una gran cantidad de tareas diferentes serían casi imposibles de representar, ya que la distancia entre ellas sería demasiado reducida y volvería su labor de análisis demasiado complicada.

Si en cambio el usuario quiere analizar un instante de tiempo particular dentro de la duración total de la traza, puede utilizar la funcionalidad de *Go to time*, que le permitirá avanzar directamente a ese instante de tiempo sin la necesidad de avanzar o retroceder páginas manualmente.

También se permite al usuario realizar zoom, es decir, aumentar o reducir el valor de estas franjas temporales. Esta funcionalidad, permite que el usuario pueda ver en mejor detalle eventos que hayan ocurrido en instantes de tiempo muy cercanos, permitiendo realizar un mejor análisis de la traza.

Para la realización de una correcta labor de análisis, el usuario no solo ha de contar con una representación visual de los eventos que hayan ocurrido, también es importante que pueda acceder a la información de estos eventos que no se puede representar mediante una imagen. Si el usuario hace clic derecho con el ratón sobre cualquiera de los eventos de la traza, se mostrará en la posición del cursor una ventana flotante que contiene toda la información disponible sobre el evento.

La aplicación también muestra un pequeño resumen de la información general de la traza, información como la duración total de esta o sus instantes de inicio y fin.

En cualquier momento de la ejecución de VEF-Visor, el usuario podrá decidir consultar un nuevo fichero de trazas. La aplicación está diseñada para adaptarse y poder cambiar de la antigua representación visual de la traza a la nueva, sin que ello afecte a la experiencia del usuario.

La Figura 6.1 muestra el estado de la aplicación en el momento de su apertura, aún con ficheros sin cargar. Se puede observar como se muestra el nombre en la parte central y como la única acción disponible pare el usuario es la apertura de un nuevo fichero de trazas.

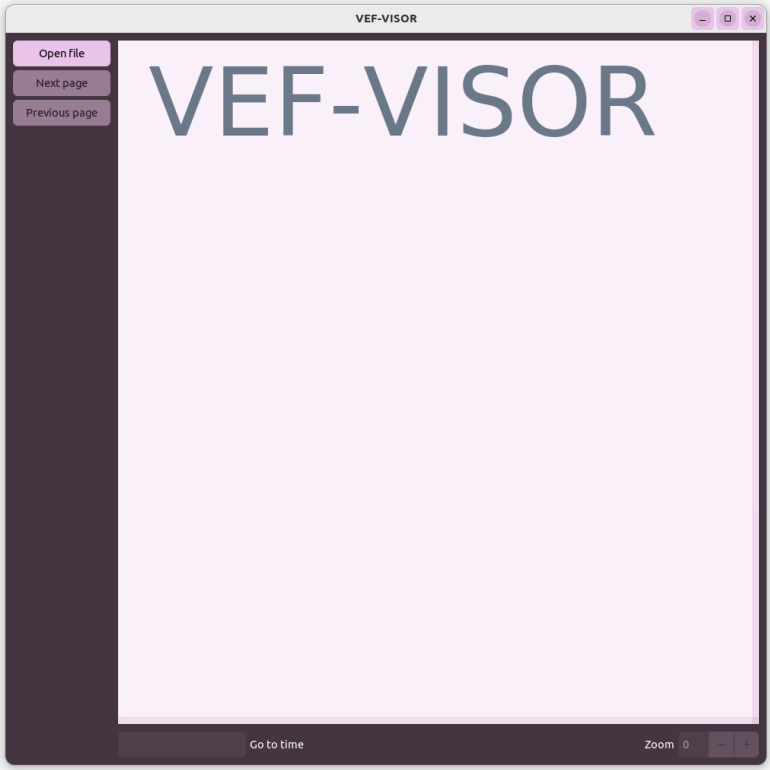

Figura 6.1: VEF-Visor antes de la apertura de la traza

6.1.2. Distribución de los componentes

Una de las partes más importantes cuando se crea una aplicación que cuente con una interfaz de usuario, es la distribución de los diferentes componentes de manera que resulten claros e intuitivos para los usuarios que den uso al sistema.

Como se puede observar en la Figuras 6.1 y 6.2, la parte central de la aplicación se encuentra ocupada por el área de representación de las trazas, ocupando esta la mayoría del espacio de la ventana. Esto se debe a que se trata del componente más importante de la aplicación y, a mayores, las dimensiones que tenga resultan relevantes, ya que cuanto más espacio de ventana pueda acaparar, mayor cantidad de eventos se podrán mostrar de forma simultánea.

A la izquierda de la zona de dibujado, encontramos el área en el que se agrupan los botones, estos se ordenan de manera que la primera tarea que se debe realizar con la aplicación, la apertura de ficheros, sea la primera en la columna de botones y, los botones de avance y

retroceso de página, se muestren inmediatamente después. Como podemos ver en la Figura 6.2, en la parte inferior de los botones se muestra el resumen de los datos de la traza.

En la parte inferior de la zona de dibujado se encuentran los controles de navegación *Go to time* y *Zoom*, de manera que queden cercanos y cómodos para que el usuario pueda utilizarlas en cualquier momento durante la navegación. Por último, entre ambos controles, se encuentra el nombre del directorio en el que se alojan las trazas, ya que resulta cómodo para el usuario conocer en todo momento sobre qué ficheros se está trabajando.

La Figura 6.2 muestra el estado de la aplicación inmediatamente después de la apertura de un fichero de trazas. Se puede observar como la zona de dibujado ya no muestra el nombre de la aplicación, sino el instante inicial de la traza y los eventos que se encuentran en el rango de tiempo. También observamos como se muestran los valores globales de la traza en la parte izquierda y el nombre del directorio en la parte inferior.

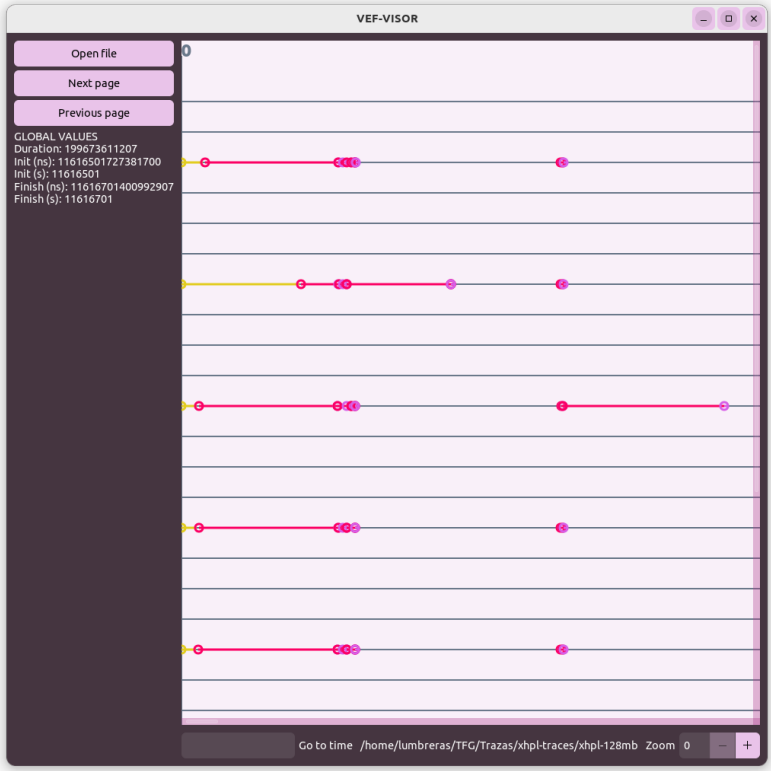

Figura 6.2: VEF-Visor tras la apertura de una traza

6.2. Apertura y lectura de las trazas

Durante esta sección, explicaremos la manera en la que la aplicación trabaja con los ficheros de trazas.

6.2.1. Formato de las trazas

Durante las primeras etapas del desarrollo del proyecto, se planteó cuál debía ser el formato de trazas a utilizar para alimentar al sistema; el debate se encontraba entre utilizar los archivos de traza sin procesar (con el formato *.veft*) o el archivo resultante de la mezcla de las trazas. Cada una de las opciones planteaba sus beneficios y perjuicios, mientras que la traza mezclada contaba con datos interesantes de representar como la tarea de destino de un evento, carecía de otros datos de relevancia para las trazas, como los tiempos absolutos de duración de los eventos (solo almacenan los tiempos relativos entre eventos). La decisión final fue utilizar las trazas sin procesar, ya que contaban con una serie de beneficios detallados a continuación:

- Menor tamaño: estas trazas son ficheros que contienen los datos de los eventos y tareas representados en formato binario, esto permite que la cantidad de información almacenada sea lo más eficiente en relación al espacio que ocupan en memoria.
- Tamaño estandarizado: al estar representadas en formato binario, cada evento almacenado en el fichero de traza ocupa una cantidad fija de espacio, frente a la representación en texto que es dependiente del número de caracteres que se deban representar. Esto facilita la lectura de las trazas y su procesado.
- Facilidad de representación: al encontrarse divididas por tareas, se facilita el dibujado de estas, ya que se pueden representar de forma secuenciada.
- Información más relevante: tras realizar la comparativa entre trazas, se determinó que la traza sin procesar almacenaba la mayor cantidad de información relevante para la aplicación.
- Integración con VEF-Prospector: los programas que componen VEF-Prospector están desarrollados para poder trabajar con este tipo de trazas, evitando así la duplicidad de funcionalidades mediante la reutilización y el aprovechamiento de código.

6.2.2. División de las trazas

Una vez tomada la decisión sobre el formato de trazas que utilizaremos para alimentar a nuestra aplicación, se debe poner la mira sobre la manera en la que estas serán tratadas en código.

Como se ha mencionado con anterioridad, algunas de los ficheros de trazas con los que se puede llegar a trabajar, pueden representar tiempos de ejecución de horas, consecuentemente

teniendo un tamaño elevado. Para solventar este problema, se tomó la decisión de realizar la lectura de las trazas por bloques, es decir, establecer un número de eventos que se deberán leer y almacenar en memoria en todo momento, con el fin de no saturar la memoria del ordenador que tenga en ejecución la aplicación.

La creación de este bloque de eventos se ve enormemente favorecida por la decisión de usar los ficheros *.veft* en lugar de la traza final, ya que al conocer el tamaño exacto que ocupa cada evento, se puede realizar una operación de lectura sobre el fichero que solo recupere el número de eventos deseado. El código empleado para la lectura del bloque de eventos es el siguiente:

```
int tarea = open(absolutePath, O_RDONLY);
ssize_t bytesRead = pread(tarea, listaTareas[indiceFichero].listaEventos,
    READ_SIZE * sizeof(struct MPI_event), bloqueLectura *
    READ_SIZE * sizeof(struct MPI_event));
close (tarea);
```

6.2.3. Lectura de los ficheros

Debido a la decisión tomada sobre el formato de la traza, si solo alimentáramos al programa con un solo fichero *.veft*, este únicamente podría representar los eventos de la tarea que hubiéramos seleccionado. La manera de solventar este problema es realizar la lectura de todos los ficheros *.veft* existentes en el directorio. Para ello, cuando el usuario pulsa el botón de abrir fichero, se muestra una ventana como en la Figura 6.3, que permite al usuario navegar por el sistema de ficheros de su sistema. En esta ventana, el usuario deberá seleccionar un fichero cualquiera que se encuentre dentro del directorio que genera la ejecución de *vmpirun* (ver Sección 3.2.1). Tras ello, el programa realizará la lectura de todos aquellos ficheros que contengan en su nombre el patrón de texto `''^[0-9]+\.veft$''`, es decir, aquellos ficheros que comiencen por al menos un carácter numérico y tengan la extensión .veft. El código empleado para la lectura de los ficheros del directorio es el siguiente:

```
//Creamos el patrón de nombres que deberán seguir los archivos
regex_t regex;
const char *pattern = "^[0-9]+\\.veft$";
if (regcomp(&regex, pattern, REG_EXTENDED)) {
    printf("Error al compilar la expresión regular.\n");
    return;
}
//Cargamos el directorio padre del archivo
dp = opendir(folderPath);

/* ... */

//Creamos el nombre del fichero
char *nombreTarea[10];
```

```
itoa(indiceFichero, nombreTarea, 10);
strcat(nombreTarea, ".veft");
//Creamos la ruta absoluta del fichero
char *absolutePath[strlen(folderPath) + strlen(nombreTarea) + 1];
strcat(strcpy(absolutePath, folderPath), "/");
strcat(absolutePath, nombreTarea);
```

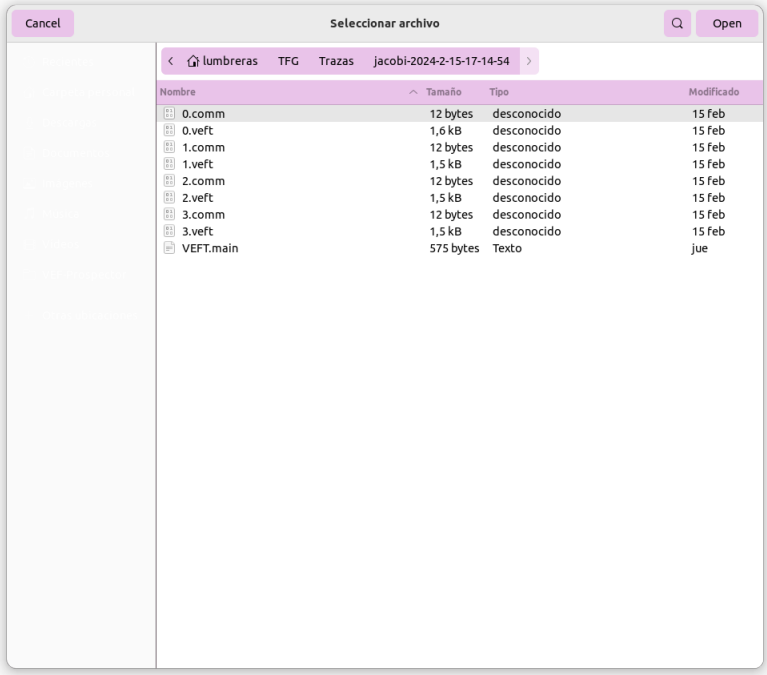

Figura 6.3: Ventana de selección de ficheros

6.2.4. Carga de los valores iniciales

Al haber descartado el formato procesado de las trazas, con ello hemos perdido también ciertos valores que se almacenaban en la cabecera (ver Sección 3.2.2). Para este sistema, los valores imprescindibles son el instante de inicio y fin de la traza, algo que no se almacena dentro de los ficheros .veft, pero sí que se almacena dentro del fichero *VEFT.main*, otro formato de fichero creado tras la ejecución de *vmpirun* que condensa valores globales sobre la traza y sus tareas, como el número total de tareas, así como la parte relevante para el visor, los instantes de inicio y fin de cada tarea. Estos se almacenan de dos maneras diferentes, como tiempos relativos y tiempos absolutos, escogiendo la segunda opción ya que es la utilizada durante la creación de los archivos *.veft*. En la Figura 6.4 podemos observar el contenido de

un fichero VEFT.main; en cada una de las líneas de tareas, leeremos el valor de los campos "TI" y "TF" y calcularemos el menor tiempo de inicio y mayor tiempo de fin.

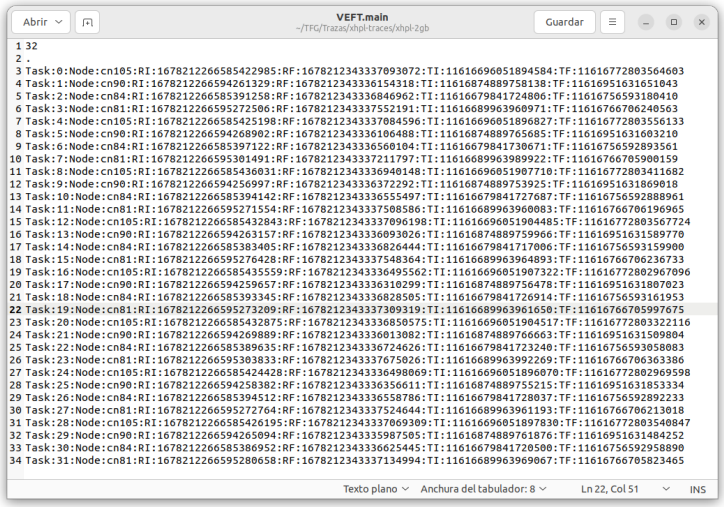

Figura 6.4: Fichero de trazas VEFT.main

El código encargado de realizar la carga inicial de valores es el siguiente:

```
/* ... */

tarea = fopen(mainPath, "r");
while ((bytesRead = getline(&linea, &len, tarea)) != -1) {
    //Buscamos la cadena de tiempo inicial y final por cada tarea
    char *inicio_TI = strstr(linea, "TI:");
    char *inicio_TF = strstr(linea, "TF:");

    if (inicio_TI != NULL && inicio_TF != NULL) {
        char valor_TI[50], valor_TF[50];
        sscanf(inicio_TI + 3, "%[^\n:]", valor_TI);
        sscanf(inicio_TF + 3, "%s[^\n:]", valor_TF);

        long ti = (long) strtol(valor_TI, NULL, 10);
        long tf = (long) strtol(valor_TF, NULL, 10);

        //Almacenamos el menor tiempo inicial y el mayor tiempo final
        if (ti < valoresGlobales.tiempoInicialGlobal)
            valoresGlobales.tiempoInicialGlobal = ti;
        if (tf > valoresGlobales.tiempoFinalGlobal)
            valoresGlobales.tiempoFinalGlobal = tf;
}   }
```

6.3. Impresión de las trazas

Durante esta sección, explicaremos la manera que tiene la aplicación de mostrar el contenido de los ficheros leídos y la lógica utilizada para su impresión dentro de la zona de dibujado.

6.3.1. Impresión de la cuadrícula

Como se puede observar en la Figura 6.5, la zona de impresión se encuentra dividida por líneas verticales y horizontales, formando una especie de cuadrícula que refuerza la información visual que obtienen los usuarios. Las líneas verticales representan el instante de tiempo que se imprimiría en esa coordenada precisa, mientras que las horizontales representan cada una de las tareas que se han encontrado en la traza.

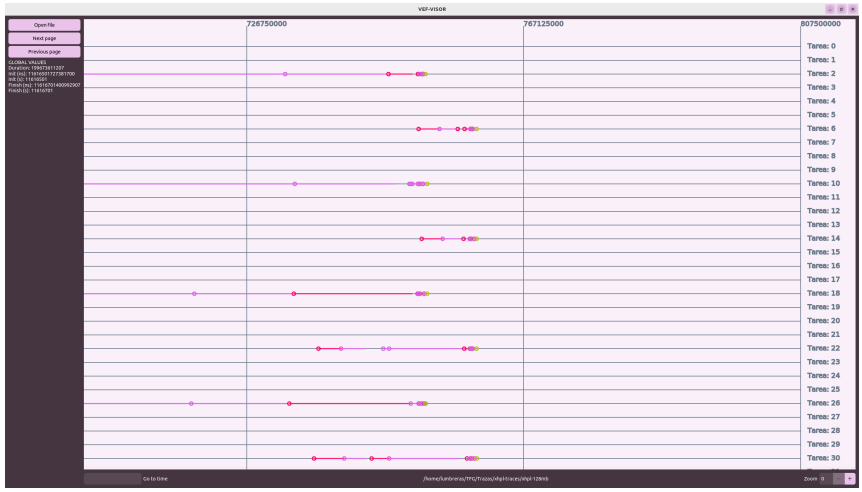

Figura 6.5: VEF-Visor mostrando el final de la página

Cuando se realiza el dibujado de la traza por cualquier motivo, todo lo que estuviera representado con anterioridad se pierde, por lo que las líneas de cuadrícula deben ser dibujadas de nuevo, hayan o no cambiado su valor.

Las líneas horizontales o líneas de trazas se mantienen estáticas durante toda la navegación por la traza, ya que el número total de tareas no varía en ningún momento. El código encargado de su dibujado es el siguiente:

```
for (int i = 0; i < totalTasks; i++) {
    //Colocamos el puntero en la posición inicial
    cairo_move_to(cr, 0, (DRAWING_AREA_MARGIN * (i + 2)));
    cairo_get_current_point(cr, &coord.x, &coord.y);
    cairo_line_to(cr, DRAWING_AREA_MAX_X, coord.y);
```

```
            //Imprimimos el número de tarea al final de la línea
            cairo_rel_move_to(cr, 20, 5);
            char* numeroTarea[50];
            sprintf(numeroTarea, "Tarea: %d", i);
            cairo_text_path(cr, numeroTarea);
        }
```

En cambio, aunque para las líneas verticales o líneas de tiempo, su posición se mantenga fija en la pantalla (se divide el total de la zona de dibujado en 20 segmentos), el valor temporal al que está asociada cada línea variará en función del estado de la navegación. Por ejemplo, si el usuario ha cambiado de página, se deberán mostrar los instantes temporales correspondientes a la siguiente franja temporal. El código encargado de su dibujado es el siguiente:

```
    for (int i = 0; i <= TIME_DIVISIONS; i ++) {
        long posicionLinea = (DRAWING_AREA_MAX_X * i) / TIME_DIVISIONS;
        cairo_move_to(cr, posicionLinea, 20);
        cairo_line_to(cr, posicionLinea, gtk_drawing_area_get_content_height(
            GTK_DRAWING_AREA(drawingArea)));
        cairo_move_to(cr, posicionLinea, 20);
        //Calculamos el tiempo de la línea
        tiempoFinal = valoresGlobales.tiempoInicialPagina +
            ((tiempoDibujado * i) / TIME_DIVISIONS);
        //Guardamos la primera posición de la pantalla
        if (i == 0)
            tiempoInicial = tiempoFinal;
        //Imprimimos el tiempo en la parte superior de cada línea
        char tiempo[30];
        sprintf(tiempo, "%ld", tiempoFinal - valoresGlobales.tiempoInicialGlobal);
        cairo_text_path(cr,tiempo);
    }
```

6.3.2. Impresión general

Una vez la aplicación ha representado la cuadrícula, procede a realizar la impresión de cada evento de forma individualizada.

Para esto, el programa recorre cada entrada de la lista de eventos de cada tarea y, tras analizar el tiempo de inicio y fin del evento, las divide en cuatro categorías:

- Impresión completa: esta categoría agrupa a los eventos que comienzan y terminan dentro del rango de impresión.
- Impresión anterior: esta categoría agrupa a los eventos que han comenzado en un rango anterior, pero terminan dentro del rango de impresión actual.

- Impresión siguiente: esta categoría agrupa a los eventos que comienzan en el rango de impresión actual pero terminan en un rango posterior.
- No impresión: esta categoría agrupa a los eventos que no se pueden representar en este rango de impresión.

Las tres primeras categorías producen un tipo diferente de impresión, mientras que la cuarta categoría marca el final de la impresión para esta franja temporal y provoca que se continúe con los eventos de la siguiente tarea.

Si durante la impresión de los eventos se llega al final del bloque de lectura de la tarea (ver Sección 6.2.2), la función de impresión está preparada para solicitar la lectura de un nuevo bloque de eventos, pudiendo continuar con la impresión de eventos hasta que estos se terminen o alcance el límite de lo que se puede representar en el rango. A continuación se muestra uno de los fragmentos de código encargado de la impresión de eventos:

```
cairo_rel_move_to(cr, 0, -10);
cairo_get_current_point(cr, &coord.x, &coord.y);
cairo_rectangle(cr, coord.x, coord.y, DRAWING_AREA_MAX_X - coord.x, 20);
cairo_rel_move_to(cr, 0, 10);
cairo_get_current_point(cr, &coord.x, &coord.y);
cairo_fill(cr);
```

6.3.3. Diferenciación por eventos

Para mejorar el entendimiento de las trazas, se decidió representar de forma diferente los eventos colectivos y los eventos punto a punto. Como podemos observar en la Figura 6.6, hay dos tipos de figuras diferentes, circunferencias seguidas por una línea horizontal y rectángulos.

El primer tipo de representación está asociado a los eventos correspondientes a comunicaciones punto a punto. Siempre que se intente mostrar un evento de este tipo se representará con una circunferencia indicando su inicio y una línea horizontal que se expande hasta su instante final. El segundo tipo de representación está asociado a los eventos correspondientes a comunicaciones colectivas, siendo un rectángulo que comienza y termina en los respectivos instantes de inicio y finalización de la comunicación.

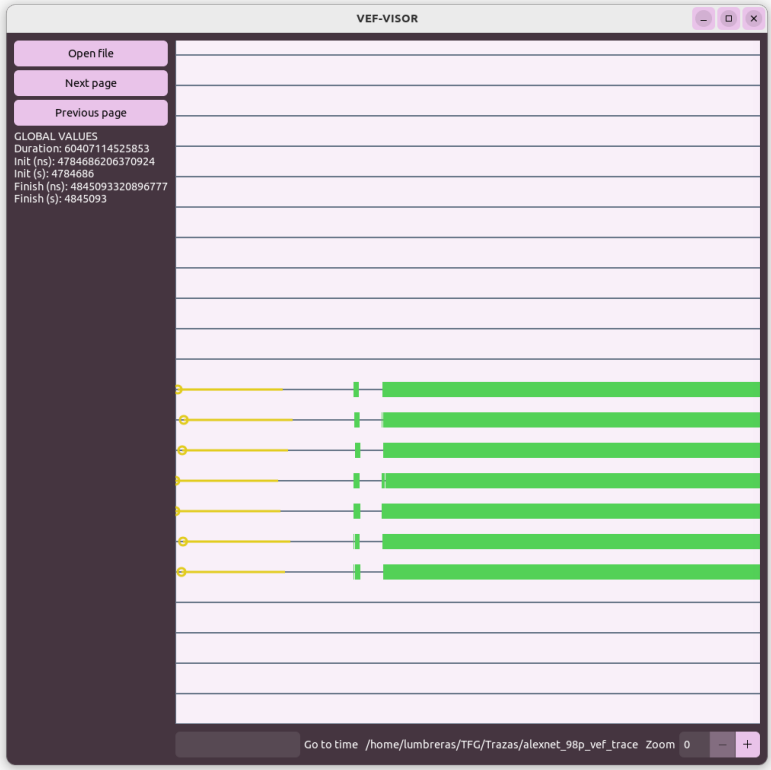

Figura 6.6: VEF-Visor mostrando eventos colectivos y punto a punto

Como también se puede observar en las Figuras 6.2, 6.5 y 6.6, los eventos representados también muestran una gama variada de colores. Esta es una decisión de diseño que ayuda a los usuarios a diferenciar de forma mas cómoda los eventos sin la necesidad de mostrar los detalles individualmente. Los diferentes colores se asocian a tipos generales de comunicaciones, siendo estos tipos los siguientes:

- Comunicaciones colectivas (MPI_Bcast, MPI_Reduce, MPI_Barrier, etc.), asociadas al color verde.
- Comunicaciones de envío (MPI_Send, MPI_Isend, etc.), asociadas al color rosa.
- Comunicaciones de recepción (MPI_Recv, MPI_Irecv, etc.), asociadas al color rojo.
- Comunicaciones de espera (MPI_Wait, MPI_WaitAll, etc.), asociadas al color amarillo verdoso.
- Comunicaciones de prueba (MPI_Test), asociadas al color azul turquesa .
- Inicio de la tarea (MPI_Init), asociadas al color amarillo.

- Finalización de la tarea (MPI_Finalize), asociadas al color azul.

6.4. Avance y retroceso de página

Durante esta sección, explicaremos la forma en la que la aplicación procesa los cambios de página, es decir, cuando se avanza o retrocede la franja temporal de eventos que se imprimen en la zona de dibujado.

6.4.1. Avance de página

El avance de página resulta algo trivial para el programa, ya que se aprovecha del funcionamiento habitual del algoritmo de dibujado. Los únicos cambios necesarios son actualizar el tiempo inicial de la página, recalcular valores que hayan podido cambiar y lanzar la función de dibujado. El código encargado de avanzar la página es el siguiente:

```
static void nextPage() {
    //Recalculamos el factor de conversión y el tiempo de dibujado
    long factorConversion = TIME_CONVERSION_UNIT/valoresGlobales.zoomFactor;
    TIME tiempoDibujado = DRAWING_AREA_MAX_X * factorConversion;
    //Marcamos el avance de página
    TIME nuevoTiempoInicial = valoresGlobales.tiempoInicialPagina +
        tiempoDibujado;
    if (!valoresGlobales.tiempoFinalAlcanzado) {
        valoresGlobales.tiempoInicialPagina = nuevoTiempoInicial;
        gtk_widget_queue_draw(GTK_DRAWING_AREA(drawingArea));
    }
}
```

6.4.2. Retroceso de página

En cambio, el retroceso de página resulta algo más complejo para la aplicación. En este caso, no es suficiente con cambiar el tiempo inicial de la página, ya que, si los eventos anteriores no se encuentran en el bloque de eventos actual cargado en memoria, la aplicación no mostraría nada por pantalla.

Para ello, se creó una función encargada de buscar un instante de tiempo preciso dentro de las trazas, tanto en los bloques de eventos cargados en memoria como en los ficheros de traza.

La función de búsqueda primero consulta el bloque de eventos cargado en memoria, comprobando si el rango temporal de cualquier evento cae dentro del tiempo objetivo de la búsqueda, en ese caso, marcará ese evento como el primer evento de la impresión. En caso

de que el tiempo objetivo no se encuentre en el bloque, pedirá la carga del bloque de eventos anterior o siguiente y repetirá el proceso hasta encontrarlo o hasta que no tenga más eventos en los que buscar. El código encargado de buscar instantes de tiempo es el siguiente:

```
for (int i = 0; i < totalTasks; i++) {
    /* ... */
    //Comprobamos si el tiempo que buscamos está en un bloque superior
    if (tiempoObjetivo < tiempoInicialBloque &&
        tiempoObjetivo >= valoresGlobales.tiempoInicialGlobal) {
        //Si nos encontramos en el primer bloque,
        //reiniciamos el índice de eventos
        if (listaTareas[i].bloqueActual == 0)
            listaTareas[i].eventoActual = 0;
        else if (loadEvents(i, listaTareas[i].bloqueActual - 1))
            i --;
    //Comprobamos si el tiempo que buscamos está en un bloque inferior
    } else if (tiempoObjetivo > tiempoFinalBloque &&
        tiempoObjetivo <= valoresGlobales.tiempoFinalGlobal) {
        if (loadEvents(i, listaTareas[i].bloqueActual + 1))
            i --;
    //Comprobamos si el tiempo que buscamos está dentro
    // del bloque de lectura actual
    } else if ((tiempoObjetivo >= tiempoInicialBloque) &&
        (tiempoObjetivo <= tiempoFinalBloque)) {
        int numEvento = 0;
        do {
            TIME tiempoInicial = listaTareas[i].listaEventos[numEvento].init_time;
            TIME tiempoFinal = listaTareas[i].listaEventos[numEvento].finish_time;
            //Cuando encontremos un evento que supere al tiempo objetivo
            // terminamos de buscar
            if ((tiempoInicial >= tiempoObjetivo) ||
                (tiempoFinal >= tiempoObjetivo)) {
                listaTareas[i].eventoActual = numEvento;
                break;
            }
            numEvento ++;
        } while (numEvento < READ_SIZE);
    } else
        return 0;
}
```

Una vez actualizados todos los bloques de eventos, cuando se llame a la función de dibujado, el programa es capaz de representar sin problema el retroceso de página.

6.5. Salto a tiempo

Para la implementación del salto a un instante temporal, se reutiliza en su totalidad la funcionalidad desarrollada en la Sección 6.4.2, ya que, la función también se desarrolló teniendo en mente que los tiempos de búsqueda pudieran ser posteriores al tiempo actual de bloque. Con ello, el código que se encarga de hacer el salto temporal queda de la siguiente manera.

```
static void goToTime() {
    //Recuperamos el valor del campo de texto
    char *ptr;
    long valorTiempo = strtol(gtk_entry_buffer_get_text
        (GTK_ENTRY_BUFFER(pageBuffer)), &ptr, 10);
    //Comprobamos que el tiempo sea positivo, menor que el tiempo total
    //y un valor numérico
    if(valorTiempo >= 0 && (valorTiempo + valoresGlobales.tiempoInicialGlobal)
        <= valoresGlobales.tiempoFinalGlobal  && ptr[0] == '\0') {
        if ((valorTiempo + valoresGlobales.tiempoInicialGlobal)
            >= valoresGlobales.tiempoInicialGlobal) {
            if (buscaTiempo((valorTiempo + valoresGlobales.tiempoInicialGlobal))) {
                valoresGlobales.tiempoInicialPagina =
                    valorTiempo + valoresGlobales.tiempoInicialGlobal;
                gtk_widget_queue_draw(GTK_DRAWING_AREA(drawingArea));
}   }   }   }
```

6.6. Gestión del zoom

Durante esta sección, explicaremos la forma en la que la aplicación gestiona los cambios en el valor de zoom.

6.6.1. Zoom inicial y factor de conversión

El valor de zoom se usa en toda la aplicación para calcular una variable denominada "factor de conversión". Esta variable almacena la cantidad de tiempo que se deberá representar por píxel en la zona de dibujado. Esto es necesario ya que los tiempos en las trazas VEF 3.2.2 se almacenan en nanosegundos, por lo que si representáramos un nanosegundo por píxel, las distancias de dibujado entre comunicaciones se volverían inmensas, volviendo imposible el uso del visor.

El valor del factor de conversión se calcula de la siguiente manera:

```
long factorConversion = TIME_CONVERSION_UNIT/valoresGlobales.zoomFactor;
```

Siendo `TIME_CONVERSION_UNIT = 500000`. Por lo que, a un valor de zoom 0, cada píxel del área de dibujado representaría 500000 nanosegundos.

A mayores, existe definida una variable en código que permite establecer un factor de zoom base que se añadirá siempre al valor que introduzca el usuario.

```
#define INITIAL_ZOOM_FACTOR 10
```

6.6.2. Cambios de zoom

Habiendo explicado como utiliza la aplicación el valor del zoom, es importante describir el comportamiento de la impresión cuando el usuario cambia el valor de zoom.

La función asociada al cambio de valor del zoom es muy sencilla:

```
static void zoom() {
    if (listaTareas) {
        //Recalculamos el valor del zoom
        valoresGlobales.zoomFactor = INITIAL_ZOOM_FACTOR +
            (gtk_spin_button_get_value_as_int(zoomSpinButton) * 10);
        //Indicamos que se ha realizado zoom
        valoresGlobales.zoomDraw = TRUE;

        gtk_widget_queue_draw(GTK_DRAWING_AREA(drawingArea));
}   }
```

Como se puede observar, su única función es asignar el nuevo valor de zoom a la variable global, indicar que se ha realizado el zoom poniendo a “true” la variable de “zoomDraw” y lanzar la función de dibujado.

La diferencia radica en la función de impresión de la traza; mientras que en cualquier otro evento que cause dibujado, la función debería comenzar sobre el evento actual, es decir, el ultimo evento al que estuviera apuntando la lista de eventos, cuando se realiza zoom, se debe imprimir la lista de eventos comenzando por el primer evento que esté actualmente representado.

Esto obliga a almacenar también los punteros al primer evento que se imprimiera en la pantalla, dentro de la variable `listaTareas[i].eventoAnterior`. Esto también genera otro problema, ya que si durante la impresión anterior, el programa tuvo que avanzar o retroceder en su bloque de lectura de eventos, el puntero no estará situado en el bloque de inicio, sino en el de finalización, por lo que también almacenamos el puntero al anterior bloque de eventos en la variable `listaTareas[i].bloqueAnterior`.

El código encargado de hacer estos ajustes es el siguiente:

```
//Si acabamos de hacer zoom,
//reiniciamos a los valores anteriores de impresión
if (valoresGlobales.zoomDraw) {
    //Si el bloque de lectura ha cambiado
    if (listaTareas[i].bloqueAnterior != listaTareas[i].bloqueActual) {
        //Cargamos el bloque anterior
        loadEvents(i, listaTareas[i].bloqueAnterior);
        //Indicamos el nuevo bloque
        listaTareas[i].bloqueActual = listaTareas[i].bloqueAnterior;
    }
    //Cargamos el evento anterior
    listaTareas[i].eventoActual = listaTareas[i].eventoAnterior;
}
```

Con esta condición, conseguimos que el resto de la función de impresión pueda realizar el dibujado con normalidad, teniendo los punteros de bloque y evento apuntando correctamente.

6.7. Click sobre los eventos

Durante esta sección, entraremos en los detalles de la manera en la que la aplicación es capaz de mostrar el detalle de los eventos sobre los que el usuario haga clic mientras navega por la traza.

6.7.1. Cálculo de las coordenadas

Uno de los mayores problemas enfrentados durante el desarrollo de esta funcionalidad proviene de la manera en la que GTK [44] gestiona los "widgets" de dibujado. Una vez se ha realizado la impresión, GTK no almacena de ninguna manera los diferentes eventos o acciones que han producido el dibujo resultante. Esto es un problema porque imposibilita que la biblioteca "recuerde" que en unas coordenadas concretas de la zona de dibujado, ha representado un evento, ni mucho menos almacene los datos de este evento.

La primera parte para abordar este problema, es conocer la posición sobre la que el usuario ha hecho clic, para esto, nos apoyamos en el evento "pressed", el cuál conectamos a la función que comienza el proceso de mostrar la ventana flotante con la información.

```
g_signal_connect(clickController, "p ressed", G_CALLBACK(clickDrawingArea), NULL);
```

Este evento envía en forma de parámetros las coordenadas del clic, pero de manera absoluta, es decir, independientemente de la posición del scroll vertical y horizontal.

Para solventar esto, debemos calcular la posición relativa del clic recogiendo los valores de las barras de scroll:

```
//Recuperamos la posición del scroll horizontal
GtkAdjustment* ajusteX = gtk_scrollable_get_hadjustment(scrollViewport);
double scrollX = gtk_adjustment_get_value(ajusteX);
//Si nos excedemos de la zona de la traza
if ((x + scrollX) > (DRAWING_AREA_MAX_X + 2))
    return;
//Recuperamos la posición del scroll vertical
GtkAdjustment* ajusteY = gtk_scrollable_get_vadjustment(scrollViewport);
double scrollY = gtk_adjustment_get_value(ajusteY);
```

Con estos valores obtenidos, podemos sumar ambos y obtener las coordenadas reales de clic del usuario en la zona de dibujado.

6.7.2. Funciones de búsqueda de eventos

Una vez obtenidas las coordenadas reales, debemos calcular la tarea a la que corresponden. Para ello, como conocemos la distancia entre cada línea de tarea y el margen superior de la zona de dibujado antes de la primera línea de tarea, realizamos un sencillo cálculo que nos devuelve la tarea correspondiente. Es importante indicar que se otorga al usuario un margen de clic de aproximadamente un 20 % en el eje vertical, ya que sino la información solo se mostraría cuando se pulsara en el píxel exacto de la línea, empeorando notablemente la experiencia de usuario.

La coordenada horizontal del clic representa el evento sobre el que se ha pulsado, en otras palabras, el instante de tiempo sobre el que consultar la información. Para conseguirlo, se ha creado una función que convierte las coordenadas horizontales de clic en el tiempo absoluto dentro de la aplicación y, tras realizar el cálculo, lanza una búsqueda del evento, que opera de forma similar a la función de búsqueda de tiempos descrita en la Sección 6.4.2, aunque con la diferencia de que esta no carga los valores en todas las listas de tareas, sino que devuelve el evento particular que se había solicitado.

6.7.3. PopOver

Una vez obtenido el evento sobre el que se ha hecho clic, se deben obtener toda la información almacenada en este. Esta labor se ve facilitada por el uso del "struct" `MPI_event`, existente con anterioridad dentro de VEF-Prospector [15].

```
struct MPI_event {
    TIME init_time; //time when the MPI_call starts; 8B
    TIME finish_time; // time when the MPI_call finish; 16B

    union {
        int32_t target_task; //Target task in p2p 20B
        int32_t root_task; //root in grpComm 20B
```

```
        };
        uint32_t lenght; //lenght in bytes 24 B

        union {
            int32_t tag; //tag for p2p  28 B
            int32_t comm; // comm por collectives  28 B
            int32_t burst_counter;
        };
        uint32_t request; //request idientifier 32B
        uint16_t type; //type of MPI event 34B
    };
```

Se prepara una cadena de texto que recopila toda la información existente dentro del evento y se actualiza el "widget" popOver con ella.

Lo último que queda es mostrar la ventana flotante en la posición del cursor, para ello, reutilizamos las coordenadas absolutas obtenidas con anterioridad y le indicamos al popOver que apunte a ellas y muestre el "widget" de la siguiente manera:

```
    //Si la cadena almacenada no es la de error
    if (eventPopOver_str[0] != 0) {
        //Asignamos el texto
        gtk_label_set_text(popOverLabel, eventPopOver_str);
        //Asignamos los padres del widget
        gtk_popover_set_child(eventPopOver, popOverLabel);
        gtk_widget_set_parent (eventPopOver, scrollViewport);

        //Calculamos la posición del cursor y
        //hacemos que el popover apunte a este
        GdkRectangle rect;
        rect.x = x;
        rect.y = y;
        rect.width = 0;
        rect.height = 0;
        gtk_popover_set_pointing_to(eventPopOver, &rect);
        //Mostramos el widget
        gtk_popover_popup(GTK_POPOVER(eventPopOver));
    }
```

En la Figura 6.7 podemos observar la aplicación mostrando la ventana flotante con la información del evento correspondiente al clic del usuario.

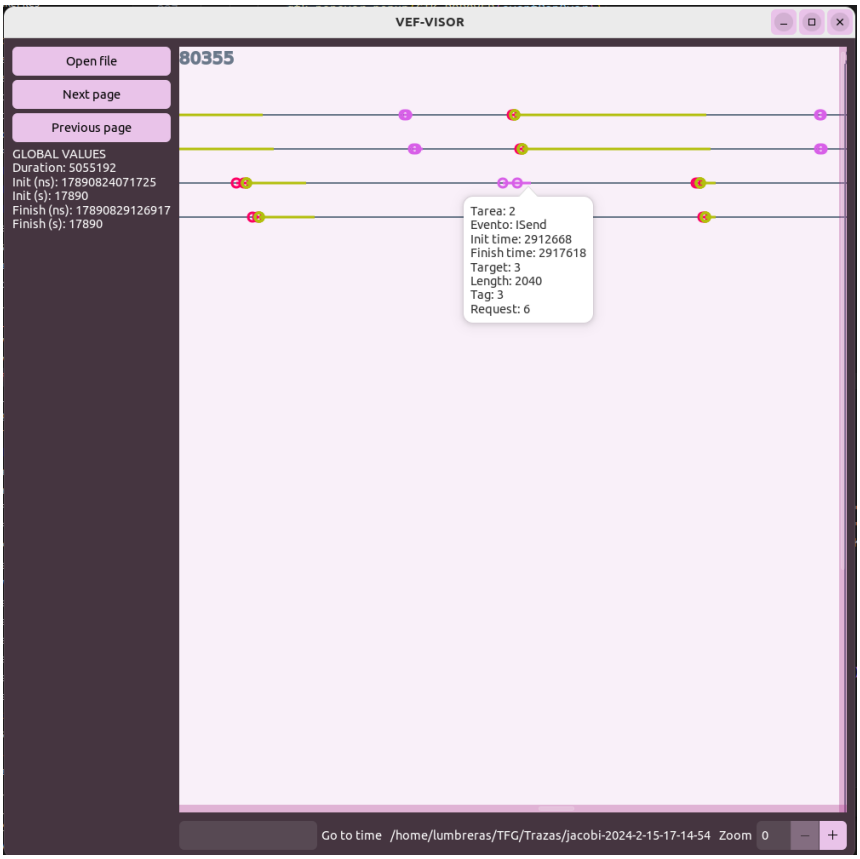

Figura 6.7: VEF-Visor mostrando los detalles de un evento

6.8. Uso de css para la gestión de colores

Una de las ventajas de GTK [44] a la hora del desarrollo de interfaces es que permite el uso de ficheros CSS (Cascading Style Sheets) para dar valores de estilo a sus diferentes componentes.

Para esta aplicación, se ha empleado esta funcionalidad con el fin de permitir una mejor mantenibilidad por parte de futuros desarrolladores. Se han separado los componentes de la aplicación en las siguientes categorías:

- button: estilo aplicado a todos los botones de la aplicación.
- scroll: estilo aplicado a las barras de scroll de la zona de dibujado.
- drawingArea: estilo aplicado a la zona de dibujado.

- window: estilo aplicado al resto de la ventana de la aplicación.

También se han creado estilos para la acción de pasar por encima de un botón o pulsarlo, con el fin de mejorar la respuesta de la aplicación a las acciones del usuario. En la Figura 6.8, se muestra un ejemplo de fichero de estilo que podría dar formato a la aplicación.


```
button {
    background-color: #E9C3E9;
    /* COLOR DE LETRA */
    color: BLACK;
}

button:hover {
    background-color: #AF46AF; /* Cambia el color cuando el mouse está sobre el botón */
}

button:active {
    background-color: #833483; /* Cambia el color cuando el botón está presionado */
}

window {
    background-color: #453540; /* Fondo para toda la aplicación */
    /* COLOR DE LETRA */
    color: WHITE;
}

.scroll {
    background-color: #AD388B;
}

.drawingArea {
    background-color: #F9F0F9; /* Fondo para la DrawingArea */
}
```

Figura 6.8: Ejemplo de fichero css para VEF-VISOR

6.9. Integracion con VEF-Prospector

Uno de los requisitos de la aplicación era su integración dentro del conjunto de herramientas proporcionado por VEF-Prospector [15]. El paquete de aplicaciones se compila a través del uso de CMake [37], una herramienta que permite la compilación y construcción de software independientemente del sistema operativo en el que se ejecute.

Para integrar VEF-Visor, se han realizado modificaciones sobre el archivo "CMakeLists.txt", un fichero que recoge las instrucciones de compilación, así como la gestión del resto de parámetros o configuraciones necesarias para garantizar el correcto funcionamiento de CMake.

El texto necesario para la compilación de VEF-Visor es el siguiente:

```
# VEF_VISOR #

find_package(PkgConfig REQUIRED)
pkg_check_modules(GTK4 REQUIRED gtk4)

set(CMAKE_C_STANDARD 11 ${COMMONDIR}/utils.c)
```

```
add_executable(vefVisor ${VISORDIR}/vef_visor.c ${COMMONSRC})
target_compile_options(vefVisor PRIVATE
    "-DSTYLE_FYLE=\"${CMAKE_INSTALL_PREFIX}/share/styles.css\"")

target_link_libraries(vefVisor PRIVATE m)

target_include_directories(vefVisor PRIVATE ${GTK4_INCLUDE_DIRS})
target_link_libraries(vefVisor PRIVATE ${GTK4_LIBRARIES})
INSTALL(FILES ${VISORDIR}/styles.css  DESTINATION share)
```

Podemos observar como se añade la condición de que la biblioteca GTK4 [44] se encuentre disponible entre las bibliotecas del usuario. También se resuelven las diferentes dependencias de VEF-Visor con el resto de aplicaciones de VEF-Prospector.

Por último, es importante destacar la existencia de dos variables que se definen en tiempo de compilación y afectan a la aplicación: "VERBOSE_MODE" y "STYLE_FILE".

"VERBOSE_MODE" es una variable de compilación existente con anterioridad dentro de la compilación de VEF-Prospector y cumple con una función bastante habitual, permite al desarrollador compilar el programa de forma "habladora", es decir, permitiendo que este realice impresiones por la terminal de ejecución para realizar labores de mantenimiento o desarrollo que, sin la existencia de esta opción, se mostrarían siempre a los usuarios.

"STYLE_FILE" es la variable que permite que los ficheros de estilo descritos en la sección 6.8 puedan ser accedidos por la aplicación independientemente del directorio sobre el que se esté ejecutando. Esto resulta importante porque, tras el proceso de instalación de la biblioteca mediante CMake, el comando de apertura de la aplicación se puede lanzar desde cualquier directorio del ordenador y, si la aplicación dependiera de encontrar el fichero de estilo dentro de la carpeta de instalación, provocaría que en estos casos no se mostrase correctamente.

6.10. Pruebas

Para garantizar el correcto cumplimiento de los requisitos planteados en las Secciones 5.1.1 y 5.1.2, se ha sometido a la aplicación a numerosas pruebas periódicas durante la etapa del desarrollo del proyecto.

6.10.1. Pruebas de integración

La finalidad de estas pruebas es verificar que los diferentes sistemas y componentes que forman la aplicación, se comportan de la manera esperada.

Estas pruebas se han realizado utilizando cuatro trazas creadas tras la ejecución de diferentes programas MPI, siendo estas muy variadas entre sí tanto en número de tareas, tipos

de comunicaciones, distribución de eventos a lo largo del tiempo de ejecución, tiempo total de duración, etc. Con estas trazas de ejemplo, se han considerado todos los casos límites que puedan afectar a la lectura, visualización y navegabilidad de las trazas, garantizando así que las pruebas realizadas sobre el sistema sean lo más completas posibles.

En las Tablas 6.1-6.11, se recogen los casos de prueba más relevantes a los que se ha sometido la aplicación. El formato seguido para cada uno de los casos de prueba es el siguiente:

- **CP:** identificador único asociado a cada riesgo.
- **Descripción:** descripción resumida del caso de prueba.
- **Objetivo:** sistema o componente sobre el que se realiza el caso de prueba.
- **Resultado esperado:** resultado que se debería obtener tras la ejecución del caso de prueba.
- **Resultado de la prueba:** resultado obtenido tras la ejecución de la prueba.

CP01	Apertura inicial
Descripción	En este caso se prueba que la aplicación se inicie de forma correcta.
Objetivo	Tras la ejecución inicial de la aplicación, esta se abre mostrando la pantalla de inicio y con el estado correcto en sus componentes.
Resultado esperado	La aplicación se abre correctamente.
Resultado de la prueba	Correcto.

Tabla 6.1: Caso de prueba 1: “Apertura inicial”

CP02	Selección de fichero
Descripción	En este caso se prueba que la ventana de selección de ficheros se muestre correctamente.
Objetivo	Tras pulsar el botón “Open file”, se muestra la ventana flotante de selección de ficheros.
Resultado esperado	La ventana se muestra correctamente.
Resultado de la prueba	Correcto.

Tabla 6.2: Caso de prueba 2: “Selección de fichero”

CP03	Apertura de fichero
Descripción	En este caso se prueba que los ficheros se abran correctamente.
Objetivo	Tras seleccionar un fichero de trazas en la ventana de selección de ficheros, se realiza la apertura y lectura de este.
Resultado esperado	El fichero se lee correctamente.
Resultado de la prueba	Correcto.

Tabla 6.3: Caso de prueba 3: "Apertura de fichero"

CP04	Visualización de la traza
Descripción	En este caso se prueba que la traza se muestre correctamente.
Objetivo	Tras la lectura de un fichero de traza, esta se muestra en la zona correspondiente de la pantalla.
Resultado esperado	La traza se muestra correctamente.
Resultado de la prueba	Correcto.

Tabla 6.4: Caso de prueba 4: "Visualización de la traza"

CP05	Navegación por la traza
Descripción	En este caso se prueba que la navegación básica por la traza sea correcta.
Objetivo	Cuando haya una traza representada, se puede navegar horizontal y verticalmente por su representación.
Resultado esperado	La navegación funciona correctamente.
Resultado de la prueba	Correcto.

Tabla 6.5: Caso de prueba 5: "Navegación por la traza"

CP06	Avance/retroceso de página
Descripción	En este caso se prueba que el avance o retroceso de la traza sea correcto.
Objetivo	Cuando haya una traza representada, se puede avanzar o retroceder en el momento temporal que se encuentra representado.
Resultado esperado	El avance/retroceso funciona correctamente.
Resultado de la prueba	Correcto.

Tabla 6.6: Caso de prueba 6: "Avance/retroceso de página"

CP07	Cambio de zoom
Descripción	En este caso se prueba que la realización del zoom sea correcta.
Objetivo	Cuando haya una traza representada, se puede modificar el valor del zoom y la representación se modificará de forma acorde.
Resultado esperado	El zoom funciona correctamente.
Resultado de la prueba	Correcto.

Tabla 6.7: Caso de prueba 7: "Cambio de zoom"

CP08	Salto de tiempo
Descripción	En este caso se prueba que el salto de tiempo sea correcto.
Objetivo	Cuando haya una traza representada y se inserte un valor de tiempo correcto en el campo "Go to time", la representación de la traza se modifica de forma adecuada.
Resultado esperado	La representación se actualiza correctamente.
Resultado de la prueba	Incorrecta. Para ciertos valores límite, se produce un cierre inesperado de la aplicación.

Tabla 6.8: Caso de prueba 8: "Salto de tiempo"

CP09	Clic sobre evento
Descripción	En este caso se prueba que la información mostrada de los eventos sea correcta.
Objetivo	Cuando haya una traza representada y se realice clic sobre la representación de un evento, se muestra una ventana flotante que contiene la información del evento.
Resultado esperado	La información se muestra correctamente.
Resultado de la prueba	Correcta.

Tabla 6.9: Caso de prueba 9: "Clic sobre evento"

CP10	Visualización de valores globales
Descripción	En este caso se prueba los valores globales de la aplicación se muestran de forma correcta.
Objetivo	Cuando haya una traza representada, se muestra un resumen de los datos generales de la traza en los espacios designados para ello.
Resultado esperado	La información se muestra correctamente.
Resultado de la prueba	Correcta.

Tabla 6.10: Caso de prueba 10: "Visualización de valores globales"

CP11	Tiempos de carga
Descripción	En este caso se prueba que los tiempos de carga de la aplicación sean correctos.
Objetivo	En cualquier momento que se produzca una lectura de ficheros, esta finaliza en un tiempo inferior a 10 segundos.
Resultado esperado	El tiempo de lectura es correcto.
Resultado de la prueba	Correcta.

Tabla 6.11: Caso de prueba 11: "Tiempos de carga"

6.10.2. Pruebas de aceptación

La finalidad de estas pruebas es verificar que el proyecto cumple con los requisitos y expectativas del cliente.

Durante las semanas de desarrollo de la aplicación y siguiendo la metodología aplicada para este proyecto (ver sección 2.1), se concertaron reuniones periódicas en las que se realizaron pruebas sobre la aplicación para garantizar la conformidad del cliente, es decir, el tutor de proyecto, sobre el resultado final de la aplicación.

Algunos de los cambios más relevantes resultantes de estas reuniones fueron los siguientes:

- En las primeras iteraciones, la aplicación representaba cada tipo diferente de comunicador de evento con un color distinto, a lo que se propuso agruparlos por categorías a fin de facilitar la comprensión de las trazas.
- Los tiempos que se muestran en la aplicación deberán estar normalizados sobre el instante de inicio, ya que los valores resultaban poco representativos.
- Los colores con los que se mostraban las barras de "scroll" en la zona de impresión de trazas eran demasiado similares al fondo de la aplicación, dificultando su uso.
- El nombre del fichero de trazas antes sustituía el nombre de la aplicación, ahora tiene designado su propio espacio en la interfaz.
- La pantalla que muestra el tramo final de la traza, antes dejaba mucha espacio en blanco en la zona de impresión, se modificó para que ocupase toda la extensión horizontal de la zona.

Estos cambios, algunos de ellos representados como requisitos, han permitido que este proyecto satisfaga de mucha mejor manera las necesidades del cliente.

Capítulo 7

Conclusiones y mejoras

7.1. Conclusiones

En el trabajo expuesto en esta memoria se ha realizado una aplicación capaz de leer e interpretar las trazas generadas por VEF Traces [28]. De esta manera, se ha cumplido con los objetivos planteados inicialmente en este proyecto, de la manera que se resume a continuación.

En primer lugar, se ha realizado una labor de estudio y comprensión sobre el framework VEF Traces, así como sobre los conceptos MPI [26] necesarios para la realización de este trabajo. También se ha realizado una labor de investigación sobre las herramientas apropiadas para las características particulares del proyecto, por ejemplo, el uso imperativo de C [34].

Se ha aprendido a utilizar una biblioteca para la creación de interfaces visuales como es GTK [29], así como se ha aprendido a desarrollar este tipo de aplicaciones en el lenguaje de programación C.

A continuación, se ha desarrollado la aplicación cumpliendo con los requisitos detallados durante la etapa de análisis, garantizando, mediante pruebas periódicas, que el producto resultante cubra de la mejor manera posible la necesidad planteada, una herramienta de "profiling" que pueda ser utilizada por desarrolladores de aplicaciones HPC en MPI.

Este trabajo ha planteado numerosos retos al alumno, ya que la limitación en cuanto al lenguaje de programación, ha supuesto una gran inversión de tiempo de estudio para familiarizarse con nuevas tecnologías que han sido imprescindibles para la consecución de este proyecto. Así mismo, el proyecto sobre el que está basado este trabajo de fin de grado, VEF Traces, ha implicado una extensiva labor de investigación y comprensión sobre este mismo con el fin de ofrecer una aplicación que cumpliese de la mejor manera los objetivos planteados.

7.2. Trabajo Futuro

Aunque el proyecto haya concluido de forma exitosa cumpliendo con la mayoría de sus objetivos, todavía cuenta con margen de mejora. Algunas de las mejoras que se podrían implementar son:

- Mejorar el tiempo de lectura y procesado de ficheros, ya que el algoritmo utilizado para ciertas funcionalidades podría ser optimizado, mejorando así el rendimiento general de la aplicación.
- Mejoras generales de la interfaz, por ejemplo, permitiendo que los valores de tiempo y número de traza dentro de la zona de dibujado sean siempre visibles para el usuario.
- Añadir una leyenda de colores dinámica, que muestre a los usuarios el color representativo de cada tipo de comunicaciones.
- Garantizar el funcionamiento del visor en otros sistemas operativos, ya que tanto el desarrollo como las pruebas de esta aplicación se han realizado únicamente en sistemas Linux.

Apéndice A

Manuales

A.1. Manual de descarga e instalación

A continuación se detallan los pasos necesarios para la instalación de VEF-Visor en cualquier máquina Linux. La mayoría de pasos de este manual se pueden encontrar en el repositorio original de VEF-Prospector [15], aunque se repetirán en esta sección por comodidad para el usuario.

1. Descarga del repositorio de VEF-Prospector mediante git:

   ```
   $ git clone https://gitraap.i3a.info/fandujar/VEF-TraceLIB.git
   $ git clone https://gitraap.i3a.info/fandujar/VEF-Prospector.git
   ```

2. Asegurar que las siguientes dependencias de paquetes se encuentran instaladas:

   - CMake ($\geq$ 2.6)
   - Distribución MPI (probado con OPEN MPI and MPICH2)
   - GTK4 ($\geq$ 4.6.9)

3. Tras obtener el repositorio, ejecutar:

   ```
   $ cd VEF-Prospector
   $ cmake . -DCMAKE_INSTALL_PREFIX="installation_folder" -DWITH_VEF_VISOR=yes
   $ make
   $ make install
   ```

   La especificación del directorio de instalación mediante *CMAKE_INSTALL_PREFIX* es opcional, siendo la ruta de instalación por defecto es *$HOME/local/vef_prospector*. *WITH_VEF_VISOR* especifica si se creará el ejecutable *VEFVisor*, permitiendo la compilación del paquete en sistemas sin entorno gráfico que carezcan de la biblioteca GTK4.

En caso de obtener algún error durante la instalación, remitir al repositorio del proyecto para una explicación más extensa.

Apéndice B

Resumen de enlaces adicionales

Los enlaces útiles de interés en este Trabajo Fin de Grado son:

- Repositorio del código: `https://gitraap.i3a.info/fandujar/VEF-Prospector`.

Bibliografía

[1] Joana Almeida. Iterative development: A starter's guide. https://distantjob.com/blog/iterative-development/, 2023-10-28. Accessed: 2024-1-4.

[2] RED-SEA. The vef traces framework. https://redsea-project.eu/the-vef-traces-framework/. Accessed: 2024-1-3.

[3] Wikimedia Foundation. Logo c. https://upload.wikimedia.org/wikipedia/commons/1/18/C_Programming_Language.svg. Accessed: 2024-1-15.

[4] Miriam Carlos. Logo visual studio code. https://icon-icons.com/icon/file-type-vscode/130084. Accessed: 2024-1-15.

[5] Miriam Carlos. Logo cmake. https://icon-icons.com/icon/cmake-logo/169379. Accessed: 2024-1-15.

[6] X. Logo ganttproject. https://pbs.twimg.com/profile_images/1221937029287698434/OwRAb9k1_400x400.jpg. Accessed: 2024-1-15.

[7] Lucidchart Company. Logo lucidchart. https://avatars.slack-edge.com/2022-07-26/3865608556737_8f4ae4a98b36ab6912b3_512.png. Accessed: 2024-1-15.

[8] Wikimedia Foundation. Logo gtk. https://upload.wikimedia.org/wikipedia/commons/thumb/7/71/GTK_logo.svg/712px-GTK_logo.svg.png. Accessed: 2024-1-15.

[9] Behdad Esfahbod Carl Worth. Logo cairo. https://www.cairographics.org/. Accessed: 2024-1-15.

[10] Overleaf. Logo overleaf. https://www.overleaf.com/for/partners/logos. Accessed: 2024-1-15.

[11] Discord Inc. Logo discord. https://discord.com/branding. Accessed: 2024-1-15.

[12] GitLab Inc. Logo gitlab. https://about.gitlab.com/press/press-kit/. Accessed: 2024-1-15.

[13] José Miguel Alonso. *Programación de aplicaciones paralelas con MPI (Message Passing Interface)*. Facultad de Informática UPV/EHU, 13/1/97. Accessed: 2024-1-9.

[14] Francisco J. Andújar, Javier Cano-Cano, Francisco J. Alfaro, and José L. Sánchez. Speeding up exascale interconnection network simulations with the vef3 trace framework. *Journal of Parallel and Distributed Computing*, pages 1–12, 2019. Accessed: 2024-1-9.

[15] F. J. Andújar et al. VEF-Prospector Git repository. `https://gitraap.i3a.info/fandujar/VEF-Prospector`, 2016. Accessed: 2024-1-9.

[16] F. J. Andújar et al. VEF-TraceLIB Git repository. `https://gitraap.i3a.info/fandujar/VEF-TraceLIB`, 2016. Accessed: 2024-1-9.

[17] EuroHPC. RED-SEA Project. `https://redsea-project.eu/`. Accessed: 2024-1-3.

[18] DEEP-Projects. DEEP-SEA Project. `https://deep-projects.eu/project/deep-sea/`. Accessed: 2024-1-3.

[19] RADIANT-DIGITAL. What is iterative design? `https://www.radiant.digital/what-is-iterative-design/`. Accessed: 2024-1-3.

[20] C. Larman and V. R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(06):47–56, jun 2003.

[21] NASA. Project mercury. `https://www.nasa.gov/project-mercury/`. Accessed: 2024-1-3.

[22] One Beyond. The pros and cons of iterative software development. `https://www.one-beyond.com/pros-cons-iterative-software-development/`, 2017-3-17. Accessed: 2024-1-4.

[23] José Manuel Marqués Corral. Proyecto/guía docente de la asignatura. `https://apps.stic.uva.es/guias_docentes/uploads/2023/545/46976/1/Documento.pdf`, 2023-9-15. Accessed: 2024-1-2.

[24] Selectra. Precio kwh. `https://tarifasgasluz.com/comparador/precio-kwh`. Accessed: 2024-1-4.

[25] Jooble. Jooble. `https://es.jooble.org/salary/programador/Valladolid#hourly`. Accessed: 2024-1-4.

[26] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 2015-6-4. Accessed: 2024-1-10.

[27] Universidad de Valladolid. Introducción a mpi. `https://informatica.uv.es/iiguia/ALP/materiales2005/2_2_introMPI.htm`. Accessed: 2024-1-8.

[28] Francisco J. Andújar, Juan A. Villar, Jose L. Sánchez, Francisco J. Alfaro, and Jesús Escudero-Sahuquillo. An open-source family of tools to reproduce mpi-based workloads in interconnection network simulators. *The Journal of Supercomputing*, pages 1–28, 2016. Accessed: 2024-1-9.

[29] GTK Development Team. Gtk. `https://www.gtk.org/`. Accessed: 2024-1-12.

[30] Michael Natterer and Jehan Pagès. Gimp: Gnu image manipulation program. `https://www.gimp.org/`. Accessed: 2024-1-12.

[31] GNOME Foundation. Gnome. https://www.gnome.org/. Accessed: 2024-1-12.

[32] Inkscape Project. Inkscape: Draw freely. https://inkscape.org/es/. Accessed: 2024-1-12.

[33] Behdad Esfahbod Carl Worth. Cairo: A vector graphics library. https://www.cairographics.org/. Accessed: 2024-1-12.

[34] B. W. Kernighan y D. Ritchie. C programming language. https://colorcomputerarchive.com/repo/Documents/Books/The%20C%20Programming%20Language%20(Kernighan%20Ritchie).pdf. Accessed: 2024-1-15.

[35] The Open Group. Unix: A standard of the open group. https://unix.org/. Accessed: 2024-1-15.

[36] Microsoft. Visual studio code. https://visualstudio.microsoft.com/es/. Accessed: 2024-1-15.

[37] Kitware. Cmake. https://cmake.org/. Accessed: 2024-1-15.

[38] BarD Software s.r.o. Ganttproject. https://www.ganttproject.biz/. Accessed: 2024-1-15.

[39] Lucidchart Company. Lucidchart. https://www.lucidchart.com/. Accessed: 2024-1-15.

[40] Overleaf. Overleaf. https://www.overleaf.com/. Accessed: 2024-1-15.

[41] LaTeX Team. Latex. https://www.latex-project.org/. Accessed: 2024-1-15.

[42] Discord Inc. Discord. https://discord.com/. Accessed: 2024-1-15.

[43] GitLab Inc. Gitlab. https://about.gitlab.com/. Accessed: 2024-1-15.

[44] GTK Development Team. Gtk – 4.0. https://docs.gtk.org/gtk4/. Accessed: 2024-1-12.

[45] Francisco J. Andújar, Juan A. Villar, Jose L. Sánchez, Francisco J. Alfaro, and Jesús Escudero-Sahuquillo. VEF Traces: A Framework for Modelling MPI Traffic in Interconnection Network Simulators. In *Proceedings of 2015 IEEE International Conference on Cluster Computing*, pages 841–848, Sep 2015. Accessed: 2024-1-9.

[46] AlonsoLP. Gtk (español). https://wiki.archlinux.org/title/GTK_(Espa%C3%B1ol), 2021-02-05. Accessed: 2024-1-11.

[47] Comité de Título. Trabajo de fin de grado: Guía del alumno. https://www.inf.uva.es/wp-content/uploads/2013/01/00-GuiaAlumnoTFG_2017.pdf, 2012-11-9. Accessed: 2024-1-2.