Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
Mención en Tecnologías de la Información

Paralelización de la aplicación VEF Mixer dentro del framework de trazas VEF

Alumno: Daniel López González

Tutor: Francisco José Andújar Muñoz

Agradecimientos

Me gustaría agradecer a mi familia y amigos por todo el apoyo y confianza durante los años de la carrera. También agradecer a mi tutor por la ayuda y seguimiento realizado durante todo el transcurso del TFG.

Resumen

VEF Traces es un framework open-source que modela tráfico MPI en simuladores de redes de interconexión HPC (High Performance Computing). El framework está formado por las herramientas *VEF-TraceLib* y *VEF-Prospector*, que contienen diferentes scripts y bibliotecas que proporcionan distintas utilidades. *VEF-Prospector* se encarga de la generación de las trazas VEF, mientras que *VEF-TraceLIB* interpreta estas trazas para generar la carga de trabajo de la red en cualquier simulador de red que utilice esta biblioteca. Estas aplicaciones se utilizan en varios proyectos de investigación financiados por España y por la Unión Europea. Por señalar los más importantes, podemos destacar los proyectos europeos RED-SEA y DEEP-SEA.

Para la creación de las trazas VEF, se utiliza en primer lugar la aplicación vmpirun. Ésta pone en marcha el comando mpirun, el cual efectúa la ejecución del programa MPI y además carga las bibliotecas de instrumentación. El siguiente paso es usar la aplicación *VEF Mixer*, que mezcla los archivos temporales generados por la biblioteca de instrumentación para obtener las trazas VEF.

El problema que se presenta en este trabajo recae en la generación final de las trazas en formato VEF. La aplicación *VEF Mixer* no está paralelizada, y por tanto, su finalización requiere un amplio espacio de tiempo, que se torna cada vez más grande cuantas más tareas conforman el programa MPI. Este proyecto aporta una solución al problema existente, paralelizando la aplicación utilizando OpenMP y consiguiendo una importante mejora en el rendimiento respecto a la aplicación secuencial.

Abstract

VEF-Traces is a open-source framework for modeling MPI traffic in interconnection network simulators. The framework comprises two main tools, *VEF-TraceLib* and *VEF-Prospector*, which contain different scripts and libraries that provide different utilities. *VEF-Prospector* handles the generation of VEF traces, and *VEF-TraceLib* interprets these traces to generate the network workload in any network simulator using this library. The VEF-traces framework is used in several research projects funded by the Spanish Government and the European Union, such as the RED-SEA and the DEEP-SEA projects.

In order to create the VEF traces, the vmpirun script is used in the first place. This script starts the mpirun command, which performs the execution of the MPI program and also loads the instrumentation libraries. The next step is to use the *VEF Mixer* application, which mixes the temporary files generated by the instrumentation library to obtain the VEF traces.

However, the *VEF Mixer* application is not parallelized, and therefore, its completion requires a large amount of time, which becomes larger and larger the more tasks that make up the MPI program. This work provides a solution to the existing problem, using OpenMP to paralelize the application, and achieving a significant improvement in the *VEF Mixer* application performance.

Índice general

Índice de Figuras

Índice de Tablas

Índice de Listados

Capítulo 1

Introducción

1.1. Contexto

VEF Traces [1] es un framework que contiene todas las herramientas necesarias para la generación de carga de trabajo en simuladores de redes de interconexión. Las trazas VEF contienen todas las comunicaciones que una aplicación MPI realiza durante su ejecución. Estas trazas permiten modelar el tráfico de estas aplicaciones, tanto las comunicaciones punto a punto como las comunicaciones colectivas, y pueden utilizarse en cualquier simulador de red.

El framework VEF [2] está constituido por *VEF-TraceLib* y por *VEF-Prospector*. *VEF-Prospector* [3] es una aplicación de código abierto que captura las llamadas MPI de la aplicación paralela y las reúne en archivos de trazas utilizando el formato VEF. *VEF-TraceLib* [4] es una librería en C de código abierto que se encarga de generar el tráfico en la red a partir de las trazas generadas por *VEF-Prospector*. Las herramientas del framework [2, 3, 4] en su conjunto permiten capturar y reproducir el tráfico generado por aplicaciones paralelas que se basan en el modelo de programación MPI (Message Passing Interface) [11].

VEF Traces está siendo utilizado actualmente en varios proyectos de investigación, tanto a nivel nacional como europeo, tales como RED-SEA [7] y DEEP-SEA [6]. Estos proyectos tienen un gran interés en este framework, ya que pretenden contribuir a un ecosistema europeo de computación de alto rendimiento (HPC) completo y sostenible para la arquitectura a exaescala.

El principal objetivo de este proyecto consiste en paralelizar mediante la API OpenMP [30] la aplicación *VEF Mixer*. Esta aplicación se emplea para obtener las trazas VEF. Esto permitiría alcanzar una mejora de rendimiento, ya que existen trazas que tienen miles de tareas y el proceso de generación es muy costoso, sobre todo en los grandes proyectos.

1.2. Motivación

El proyecto VEF Traces fue desarrollado y publicado en 2015, y desde entonces, se ha estado actualizando y mejorando [1]. El tutor académico de este TFG, Francisco José Andújar, es uno de los principales desarrolladores del proyecto VEF Traces, y ha introducido nuevos cambios y mejoras en el framework desde que se publicó.

Los proyectos de la Unión Europea que utilizan actualmente VEF Traces en sus investigaciones están en contacto con el tutor. Como resultado de esta comunicación y del estudio del flujo de trabajo del framework y el mantenimiento del mismo que realizan los desarrolladores, se ha detectado que la ejecución de aplicación *VEF Mixer*, encargada de generar las trazas VEF, puede llegar a ser muy lenta si tiene que procesar programas MPI muy grandes, ya que no está paralelizada. La motivación de este proyecto nace por intentar solucionar este problema y mejorar el rendimiento de esta aplicación para que los proyectos que utilizan este software puedan realizar sus investigaciones de manera más efectiva.

1.3. Objetivos

El principal objetivo de este trabajo es desarrollar una versión paralela de la aplicación *VEF Mixer* para conseguir una mejora del rendimiento en la obtención de las trazas VEF. La paralelización se llevará a cabo con OpenMP. Otros objetivos secundarios son los siguientes:

- Realizar un estudio sobre el funcionamiento de la aplicación *VEF Mixer* y el framework VEF Traces.
- Llevar a cabo una evaluación de todas las pruebas realizadas para sacar conclusiones coherentes.

1.4. Estructura de la memoria

La estructura de la memoria se detalla a continuación:

Capítulo 1: Introducción. El primer capítulo describe de manera breve los conceptos básicos del proyecto, objetivos y estructura de la memoria.

Capítulo 2: Metodología y planificación. En este capítulo se describe la metodología aplicada y se establece la planificación inicial, riesgos y presupuesto, y la evolución de esta planificación.

Capítulo 3: Herramientas utilizadas. En este capítulo se comentarán las tecnologías que se han empleado para el desarrollo y consecución del proyecto.

Capítulo 4: Fundamentos / Estado del Arte. En este capítulo se habla de los conceptos básicos de VEF Traces que se necesitan entender para abordar el problema.

Capítulo 5: Diseño de algoritmos En este capítulo se detallan los algoritmos originales del programa y la solución que se propone con los cambios que deben hacerse en la aplicación en relación con su diseño.

Capítulo 6: Evaluación de rendimiento En este capítulo se detallan los métodos para aplicar el desarrollo y las pruebas que se llevan a cabo para la operatividad del sistema.

Capítulo 7: Conclusiones y trabajo futuro. Se establecen conclusiones finales y se comentan los resultados obtenidos.

Anexo A Resumen de enlaces adicionales. Incluye enlaces de interés sobre el proyecto, como el repositorio de código.

Capítulo 2

Metodología y Planificación

2.1. Metodología aplicada

Existen numerosas metodologías que se pueden aplicar a la gestión de un proyecto. En este proyecto se ha adoptado un enfoque tradicional de gestión de proyectos, la metodología en cascada, que va a permitir dividir el desarrollo en etapas bien definidas.

La metodología en cascada [8] fue el primer enfoque moderno establecido para la creación de un sistema. Fue establecido por Winston W. Royce en 1970 bajo el título "Managing the Development of Large Software Systems"[9].

Aunque originalmente el enfoque se establecía de un modo iterativo con 8 fases, como se puede ver en la Figura 2.1, esta metodología se puede adaptar a cada proyecto de manera que encaje óptimamente. Para este proyecto se decide escoger un enfoque en que se diferencian 5 etapas bien definidas, que son Documentación, Análisis, Implementación, Evaluación y Despliegue. El enfoque es fiel al diseño original de esta metodología, por lo que será iterativo, permitiendo volver a una etapa anterior siempre y cuando sea necesario.

Se deben tener en cuenta las ventajas y desventajas de esta metodología. Las ventajas son las siguientes:

- El enfoque es muy estructurado y resulta más fácil medir los progresos en función de hitos bien definidos.
- El coste total del proyecto puede calcularse con precisión.
- La fase de pruebas es más sencilla, ya que pueden hacerse en función de los escenarios definidos.
- Permite realizar fases en paralelo según el tipo de proyecto.

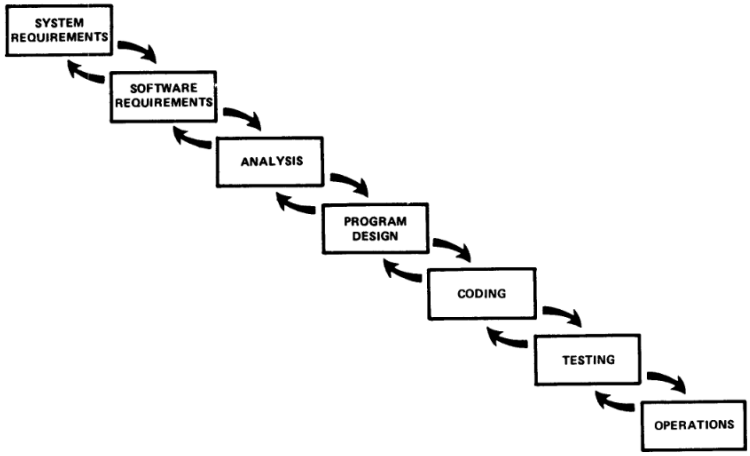

Figura 2.1: Enfoque original en cascada iterativo

En cuanto a las desventajas de esta metodología son:

- Puede resultar difícil definir requisitos a nivel abstracto.
- Al ser un modelo con estructura más definida, no va a ser tan flexible.
- Puede tardarse más en entregar el proyecto.

La razón de escoger esta metodología es porque se tienen claros los objetivos del proyecto desde el principio, y la realización en etapas o fases es la más beneficiosa, ya que su estructura bien definida permite avanzar eficaz y ordenadamente. Además, el enfoque iterativo es útil, sobretodo en las últimas fases, puesto que debido a la naturaleza del problema, es posible reanalizar el problema para resolver posibles fallos que se encuentren a la hora de hacer las pruebas.

Por otro lado, no se contempla que el desarrollo del proyecto vaya a cambiar demasiado, por lo que la falta de flexibilidad no afectaría gravemente a este proyecto.

2.2. Fases del Proyecto

Con la metodología que debemos aplicar decidida, se diferencian las siguientes fases del proyecto:

1. Documentación. La primera fase consiste en el entendimiento del enfoque general del framework VEF Traces, la documentación inicial y el funcionamiento del mismo. La

comprensión a nivel funcional va a ser de gran ayuda para afrontar el desarrollo eficientemente.

2. Análisis. En esta fase se realiza un análisis completo de la naturaleza del problema y de las necesidades mínimas para el correcto desarrollo. Además se estudian diferentes enfoques para encontrar la mejor solución. Se rediseña la aplicación para alcanzar los objetivos del proyecto.

3. Implementación. En esta fase se llevan a cabo todos los cambios en la aplicación y se asegura el buen funcionamiento de éstos.

4. Evaluación. En esta fase se realizan todas las pruebas de ejecución del desarrollo y se sacan conclusiones acerca del rendimiento y la mejora obtenida.

5. Finalización de la documentación. En esta fase se detallan los últimos aspectos del proyecto, y la redacción de la memoria del mismo.

2.3. Planificación inicial

Al principio del proyecto, se estableció una planificación con fechas límite ideales para el inicio y final de las distintas etapas.

La carga de trabajo que tiene asignada este trabajo son 12 ECTS. Un ECTS equivale a 25 horas, entonces el trabajo deberá tener una duración mínima de 300 horas. Se decide dedicar 21 horas semanales, repartidas de lunes a domingo en 3 horas diarias. El inicio del proyecto tiene lugar la primera semana de Marzo de 2023. La planificación se divide en 19 semanas desde el inicio hasta la fecha prevista de finalización, lo que hacen un total de 400 horas estimadas de trabajo.

La Tabla 2.1 recoge la planificación de fechas, mientras que la Figura 2.2 muestra el diagrama de Gantt de esta planificación, que permite visualizar fácilmente las tareas del proyecto y la distribución del tiempo para completarlas.

Tarea	Duración	Comienzo	Fin
DOCUMENTACIÓN			
Documentación framework VEF Traces	8 horas	01/03/23	04/03/23
Introducción a las herramientas del framework VEF	4 horas	01/3/23	04/03/23
Instalación de Software necesario	6 horas	05/03/23	07/03/23
Preparación del Software para su uso en ordenador personal	21 horas	06/03/23	12/03/23
ANÁLISIS			
Análisis de las necesidades del proyecto	51 horas	13/03/23	29/03/23
IMPLEMENTACIÓN			
Implementación parte GRP	60 horas	30/03/23	18/04/23
Implementación parte P2P	60 horas	18/04/2023	09/05/23
EVALUACIÓN			
Pruebas en local y depuración	50 horas	10/05/2023	27/05/23
Pruebas en el clúster	15 horas	28/05/2023	02/06/23
FINALIZACIÓN DEL PROYECTO			
Finalización de la documentación y entrega del TFG	125 horas	03/06/23	05/07/23

Tabla 2.1: Planificación inicial del proyecto

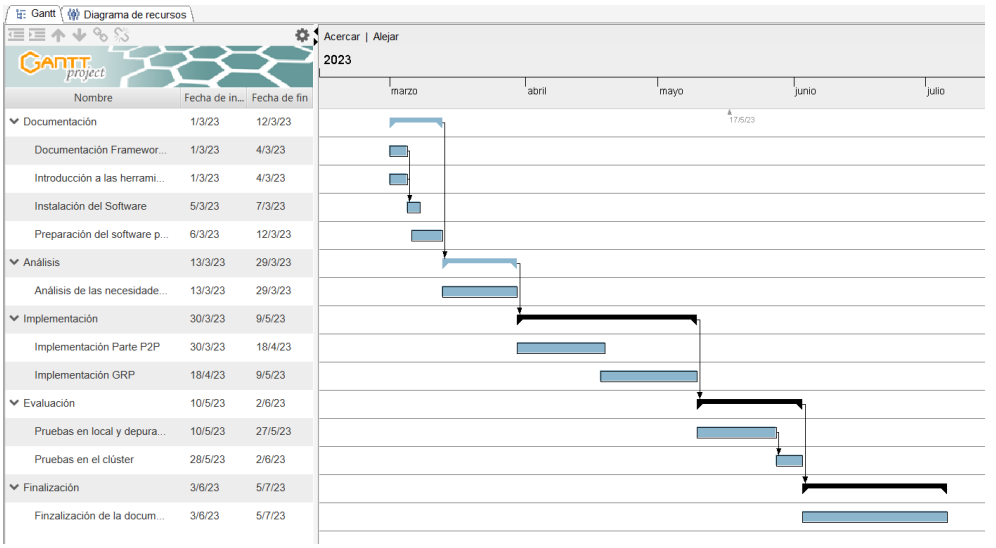

Figura 2.2: Diagrama inicial de Gantt

2.4. Plan de Riesgos

Elaborar un plan de riesgos es un aspecto fundamental en la gestión de cualquier proyecto. Se debe realizar una estimación sobre los posibles riesgos junto con la probabilidad de que estos ocurran y el impacto negativo que supondría si se materializan. Es decir, debe incluir lo siguiente:

- Posible riesgo y probabilidad de que se materialice. Se valorará en baja, media y alta.
- Impacto que tendrá el riesgo. Se valorará en bajo, medio y alto.
- Plan de mitigación. Medidas a llevar a cabo para minimizar la probabilidad de que se materialice.
- Plan de contingencia. Medidas a llevar a cabo si se materializa el riesgo, con el objetivo de reducir su impacto.

La Tabla 2.2 muestra el riesgo de que la estimación del tiempo para el desarrollo del proyecto sea incorrecta, R1.

R1	**Estimación incorrecta del tiempo**
Descripción	El desarrollo del proyecto no está siguiendo la planificación de tiempo inicial .
Probabilidad	Baja
Impacto	Medio
Plan de mitigación	▪ Estimar nuevamente el alcance real del proyecto. ▪ Realizar primero las tareas más iportantes.
Plan de contingencia	▪ Volver a definir el alcance del proyecto, según el tiempo disponible.

Tabla 2.2: Riesgo 1

La siguiente Tabla 2.3 muestra el riesgo de que se produzca un cambio de enfoque. Esto puede deberse por surgimiento de nuevas ideas o por encontrar obstáculos imprevistos.

Las Tablas 2.4 y 2.5 muestran los riesgos de que se presenten problemas técnicos, como fallas en el software, incompatibilidad de sistemas o dificultades con el hardware. Estos problemas pueden causar retrasos y afectar a la realización del trabajo.

R2	**Cambio de enfoque**
Descripción	Obstáculos imprevistos o nuevas ideas provocan la necesidad de un nuevo enfoque.
Probabilidad	Media
Impacto	Medio
Plan de mitigación	▪ Realizar un estudio completo sobre la naturaleza del problema.
Plan de contingencia	▪ Realizar un análisis para saber cómo abordar el nuevo enfoque.

Tabla 2.3: Riesgo 2

R3	**Problemas técnicos: Hardware**
Descripción	Surgen algunos problemas técnicos con el hardware del proyecto.
Probabilidad	Baja
Impacto	Alto
Plan de mitigación	▪ Mantener un buen cuidado de los dispositivos hardware para no dañarlos.
Plan de mitigacion	▪ Si se incapacita el ordenador de trabajo, reemplazarlo por otro ordenador.

Tabla 2.4: Riesgo 3

La Tabla 2.6 muestra el riesgo de que existan problemas de investigación. Los resultados obtenidos pueden no ser los esperados. También puede haber dificultades en la obtención de datos relevantes o en la interpretación de los resultados.

El riesgo 2.7 muestra el riesgo de que el alumno sufra una enfermedad.

El riesgo 2.8 muestra el riesgo de que el tutor académico sufra una enfermedad.

R4	**Problemas técnicos: Software**
Descripción	Surgen algunos problemas técnicos con el software o incompatibilidades de sistemas.
Probabilidad	Media
Impacto	Medio
Plan de mitigación	▪ Realizar copias de seguridad de los cambios realizados en local para no perder todo el progreso en caso de incapacidad de ordenador. ▪ Subir frecuentemente los cambios realizados al repositorio de Gitlab, en la rama del proyecto.
Plan de contingencia	▪ En caso de problemas con el software que usa el proyecto, comprobar si la versión utilizada y la configuración coincide con la que utiliza el proyecto. ▪ En caso de problemas con cualquier otro software, verificar su correcto funcionamiento antes de su uso. ▪ Si se trata de incompatibilidad de sistemas, comunicarse con el tutor para discutir la solución más adecuada.

Tabla 2.5: Riesgo 4

R5	**Problemas de investigación**
Descripción	Surgen dificultades con los datos obtenidos para la investigación o la interpretación de los resultados.
Probabilidad	Baja
Impacto	Alto
Plan de mitigación	▪ Estudiar la naturaleza del problema en profundidad para conocer los métodos de obtención de datos de manera efectiva. ▪ Comunicar al tutor todos los avances del proyecto para llevar un seguimiento.
Plan de mitigacion	▪ Reestablecer requisitos y remodelar el problema, incluyendo el nuevo alcance. ▪ Comunicarse con el tutor para buscar un nuevo enfoque.

Tabla 2.6: Riesgo 5

R6	**Problemas de salud del alumno**
Descripción	El alumno sufre una enfermedad o su estado de salud le incapacita continuar con el desarrollo.
Probabilidad	Baja
Impacto	Alto
Plan de mitigación	▪ Cuidar los aspectos relacionados con la salud.
Plan de contingencia	▪ Ir al médico y seguir los consejos indicados para recuperarse lo antes posible. ▪ Redistribuir las horas y el tiempo dedicado a cada fase

Tabla 2.7: Riesgo 6

R7	**Problemas de salud del tutor**
Descripción	El tutor sufre una enfermedad o su estado de salud le incapacita comunicarse con el alumno.
Probabilidad	Baja
Impacto	Medio
Plan de mitigación	▪ Cuidar los aspectos relacionados con la salud.
Plan de contingencia	▪ Ir al médico y seguir los consejos indicados para recuperarse lo antes posible. ▪ Comunicar al alumno la noticia y dejarle indicaciones en caso de que no pueda comunicarse durante el tiempo que dure en ese estado.

Tabla 2.8: Riesgo 7

2.5. Planificación final del Proyecto

Una vez terminadas las etapas del proyecto, se realiza una planificación final con los riesgos que se han materializado. Esto nos permite comparar fechas con la planificación inicial y ver los retrasos que se han producido en el proyecto.

Los riesgos que se han materializado son los siguientes:

- La instalación del framework y software subyacente en el dispositivo hardware del alumno, su ordenador personal, no se desarrolló como se había planeado, surgiendo más problemas de lo normal. Esta situación activa el riesgo R3 (Tabla 2.4).
- Durante la fase de análisis, se encontraron obstáculos imprevistos debido a la naturaleza del problema y se tuvo que adoptar un nuevo enfoque. Esta situación activa el riesgo R2 (Tabla 2.3).
- El estudiante Daniel López González, a mediados de abril, sufrió una enfermedad, lo que le impidió continuar el desarrollo del proyecto con normalidad. No le incapacitaba por completo, pero debido a las consecuencias y al estrés mental y sufrimiento por dicha enfermedad el avance se ralentizó algunas semanas. Esta situación activa el riesgo R6 (Tabla 2.7).
- Las partes de implementación y depurado resultaron llevar más tiempo del previsto, retrasando la finalización de esta etapa. Esta situación activa el riesgo R1 (Tabla 2.2).

La materialización de estos riesgos supuso la ejecución de los planes de contingencia correspondientes. Esto llevó al retraso en las fases del proyecto. La planificación final se refleja en la Tabla 2.9. El diagrama de Gantt final se refleja en la Figura 2.3

Tarea	Duración	Comienzo	Fin
DOCUMENTACIÓN			
Documentación framework VEF Traces	8 horas	01/03/23	04/03/23
Introducción a las herramientas del framework VEF	4 horas	01/3/23	04/03/23
Instalación de Software necesario	8 horas	05/03/23	08/03/23
Preparación del Software para su uso en ordenador personal	22 horas	06/03/23	13/03/23
ANÁLISIS			
Análisis de las necesidades del proyecto	55 horas	13/03/23	30/03/23
IMPLEMENTACIÓN			
Implementación parte GRP	70 horas	31/03/23	24/04/23
Implementación parte P2P	70 horas	25/04/2023	12/05/23
EVALUACIÓN			
Pruebas en local y depuración	53 horas	12/05/2023	29/05/23
Pruebas en el clúster	15 horas	30/05/2023	04/06/23
FINALIZACIÓN DEL PROYECTO			
Finalización de la documentación y entrega del TFG	125 horas	05/06/23	07/07/23

Tabla 2.9: Planificación final del proyecto

2.6. Estimación de Costes

Para la correcta planificación del proyecto, se debe llevar a cabo una estimación de todos los costes del proyecto, calculando el coste de las herramientas utilizadas y el personal del proyecto.

2.6.1. Software

Las herramientas software para llevar a cabo el desarrollo del proyecto son gratuitas, por lo que el coste de esta parte es de 0€.

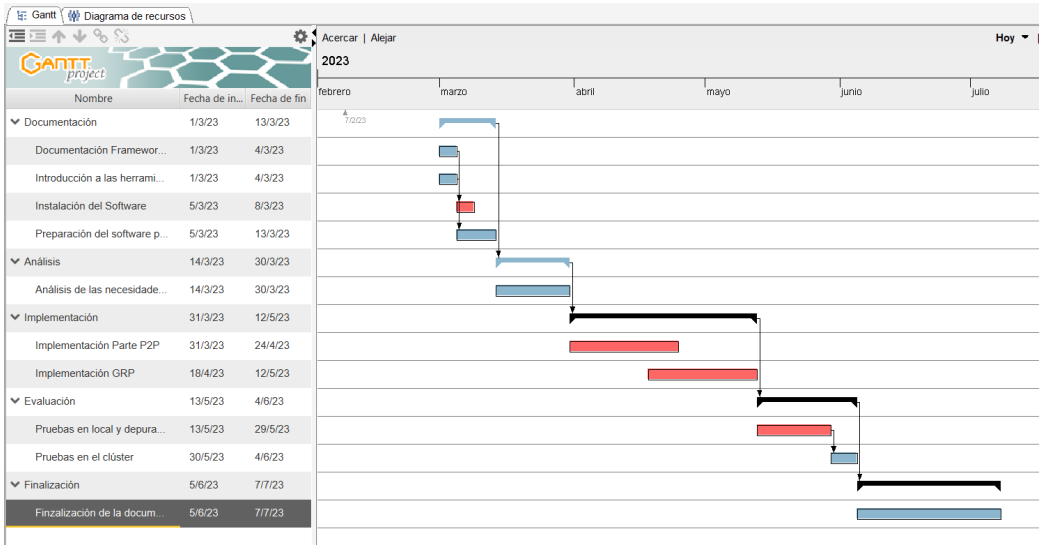

Figura 2.3: Diagrama final de Gantt

2.6.2. Hardware

Los dispositivos que se han utilizado para la realización del proyecto son los siguientes:

- Ordenador portátil del alumno: Asus VivoBook con procesador intel Core i5-1035G1 y 8GB de memoria RAM, con valor de aproximadamente 500€.
- Clúster del grupo Trasgo: se utilizan las máquinas del clúster del grupo Trasgo para realizar varias pruebas de rendimiento. No obstante, no se va a tener en cuenta para la estimación de costes, puesto que es difícil realizar un cálculo sobre el coste de amortización de las máquinas.

2.6.3. Personal

Considerando que el alumno tiene un puesto de desarrollador Junior, se debe calcular el coste de este desarrollador por las horas trabajadas. El salario de un desarrollador Junior ronda los 23.000 € anuales, o 1916,6 € al mes. Sacando el coste por hora serían 12 € por hora trabajada. El total de horas trabajadas por el alumno son 430, lo que produce un coste de 5160 €.

Se va a tener en cuenta el tutor académico como un desarrollador Senior que da soporte y supervisa al Junior. Podríamos estimar este coste como el total de las horas que han supuesto las reuniones además de ciertas horas a la semana de soporte vía mensaje.

Por tanto, considerando 1 reunión semanal de 45 minutos de duración desde el inicio del proyecto hasta la fecha de finalización, lo que supone un total de 19 semanas, sale un total

de 855 minutos por las reuniones, que hacen 14,25 horas. Teniendo en cuenta 1 hora semanal de soporte vía mensaje en la que se comunica con el alumno, eso formarían 19 horas más, lo que hace un total de 33,25 horas ≈ 33 horas. El salario de un desarrollador Senior en España ronda los 50.000 € anuales, 4166,6 € al mes. Sacando el coste por hora, serían 26,04 euros por hora. El coste que supondría este desarrollador Senior para el proyecto sería 859,32 €.

2.6.4. Presupuesto final

El total de los costes que supondría este proyecto se reflejan en la Tabla 2.10.

Concepto	**Coste**
Costes software	0 €
Costes hardware	500 €
Costes personal	6.019,32 €
Coste total	**6.519,32 €**

Tabla 2.10: Coste total del proyecto

Capítulo 3

Herramientas utilizadas

En este capítulo se presenta un resumen de las tecnologías utilizadas tanto para la gestión del proyecto, como para el desarrollo.

3.1. Visual Studio Code

Visual Studio Code [21] es un entorno de desarrollo integrado que permite la edición, depurado, compilado y ejecución de código en diferentes lenguajes. Viene con soporte integrado para JavaScript, TypeScript y Node.js y tiene un rico ecosistema de extensiones para otros lenguajes y tiempos de ejecución (como `C++`, `C#`, Java, Python, PHP, Go, .NET).

En este proyecto se ha utilizado como entorno para el desarrollo, con las extensiones de C/C++, C/C++ Extension Pack y CMake. La Figura 3.1 ilustra el logo de VS Code.

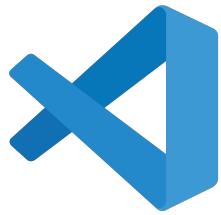

Figura 3.1: Logo de Visual Studio Code

3.2. Discord

Discord [20] es una aplicación de mensajería instantánea y chat de voz y vídeo que permite la comunicación con otras personas por medio de estas vías. Además tiene la función de

crear servidores y canales dedicados para interactuar con otros usuarios. Está disponible en diferentes sistemas operativos, como Windows, Linux, macOS, Android e iOS. Cuenta también con una versión web para interactuar desde el navegador.

Esta aplicación se ha utilizado durante el transcurso del trabajo para la comunicación con el tutor académico del proyecto. Se han realizado reuniones de videollamada o chat de voz para el seguimiento del proyecto cuando no eran presenciales, además de comunicar las dudas y otros temas relacionados con el desarrollo por medio de mensajes. La Figura 3.2 muestra el logo de esta aplicación.

Figura 3.2: Logo de GitLab

3.3. Vim

Vim [22] es un editor de texto altamente configurable construido para hacer muy eficiente la creación y modificación de cualquier tipo de texto. Se incluye como “vi” en la mayoría de los sistemas UNIX y en Apple OS X. Se debe aprender a utilizar la herramienta para poder utilizarla.

Se ha utilizado esta herramienta en este proyecto para visualizar de manera rápida en una terminal los ficheros, además de para la creación de los scripts de ejecución en las pruebas. La Figura 3.3 ilustra el logo de Vim.

Figura 3.3: Logo de Vim

3.4. OverLeaf

OverLeaf [23] [24] es un editor de texto en línea que permite la edición de documentos LaTeX de manera colaborativa y fácil de usar a través de la web, sin necesidad de instalación y con otras funcionalidades como control de versiones y documentación de aprendizaje de LaTex.

LaTex [25] es un sistema de composición tipográfica de alta calidad; incluye funciones diseñadas para la producción de documentación técnica y científica. LaTeX es el estándar de facto para la comunicación y publicación de documentos científicos. LaTeX está disponible como software libre.

Esta herramienta ha permitido la redacción de esta memoria y la revisión por parte del tutor académico. La Figura 3.4 ilustra el logo de OverLeaf.

Figura 3.4: Logo de OverLeaf

3.5. C

C [26] es un lenguaje de programación de propósito general con características de economía de expresión, moderno control de flujo y estructuras de datos, y un rico conjunto de operadores. C no es un lenguaje de "muy alto nivel", ni "grande", y no está especializado en ningún área de aplicación concreta. Pero su ausencia de restricciones y su generalidad lo hacen más conveniente y eficaz para muchas tareas que lenguajes supuestamente más potentes.

C [26] fue diseñado e implementado originalmente para el sistema operativo UNIX en el DEC PDP-11, por Dennis Ritchie. El sistema operativo, el compilador de C y prácticamente todos los programas de aplicación UNIX están escritos en C. C no está ligado a ningún hardware o sistema en particular, y es fácil escribir programas que se ejecuten sin cambios en cualquier máquina que soporte C. La Figura 3.5 ilustra el logo del lenguaje C.

Figura 3.5: Logo de C

3.6. Script Bash

Bash [28] [29] es el intérprete de comandos del Proyecto GNU: el Bourne Again SHell. Es un shell compatible con sh que incorpora características útiles del shell Korn (ksh) y del shell C (csh). Su objetivo es cumplir el estándar IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools. Ofrece mejoras funcionales sobre sh tanto para la programación como para el uso interactivo. Además, la mayoría de los scripts sh pueden ser ejecutados por Bash sin modificaciones.

Bash se ha utilizado en este proyecto para la ejecución de las pruebas tanto en local como en el clúster. En este último caso la ejecución se realiza a través de script. La Figura 3.6 ilustra el logo de Bash.

Figura 3.6: Logo de Bash

3.7. Open MP

OpenMP (Open Multi-Processing) [30] es una interfaz de programación de aplicaciones (API) para la programación multiproceso de memoria compartida en múltiples plataformas: C/C++ y Fortran. La API OpenMP define un modelo portátil y escalable con una interfaz sencilla y flexible para desarrollar aplicaciones paralelas en plataformas que van desde el escritorio hasta el superordenador.

OpenMP se ha utilizado en este proyecto para desarrollar una versión paralela de la herramienta *VEF Mixer*. La Figura 3.7 ilustra el logo de OpenMP.

Figura 3.7: Logo de OpenMP

3.8. CMake

CMake [31] es una familia de herramientas multiplataforma de código abierto diseñadas para compilar, probar y empaquetar software. CMake se utiliza para controlar el proceso de compilación de software mediante sencillos archivos de configuración independientes de la plataforma y el compilador, y generar makefiles y workspaces nativos que se pueden utilizar en el entorno de compilación elegido. El conjunto de herramientas CMake fue creado por Kitware en respuesta a la necesidad de un potente entorno de compilación multiplataforma para proyectos de código abierto como ITK y VTK.

CMake ha sido clave en el proyecto para la compilación y configuración automática de los ficheros necesarios para la ejecución. La Figura 3.8 ilustra el logo de CMake.

Figura 3.8: Logo de CMake

3.9. GitLab

GitLab [19] es una plataforma de desarrollo de software integral de código abierto con control de versiones integrado, seguimiento de incidencias, revisión de código, CI/CD y mucho más. Se puede hospedar GitLab en servidores propios, en un contenedor o en un proveedor en la nube.

El repositorio que contiene las herramientas de VEF Traces se encuentra en esta plataforma. Se ha utilizado GitLab para la gestión de código del proyecto y para el control de versiones. La siguiente Figura 3.9 ilustra el logo de GitLab.

Figura 3.9: Logo de GitLab

3.10. GanttProject

GanttProject [32] es una aplicación que permite realizar diagramas de Gantt de forma sencilla e intuitiva. Se ha utilizado en este proyecto para realizar los diagramas de Gantt de la planificación inicial y la final. La Figura 3.10 ilustra el logo de esta aplicación.

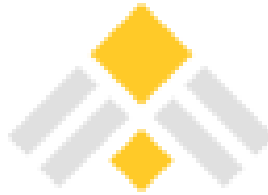

Figura 3.10: Logo de GanttProject

3.11. Draw.io

Draw.io [40] es una aplicación que permite hacer todo tipo de diagramas online. Dispone de muchas figuras y características para diseñar y personalizar el diagrama a gusto del creador. Además, su interfaz es muy intuitiva y fácil de manejar. Se ha utilizado esta herramienta en este proyecto para crear diagramas de estados con todas las casuísticas posibles para la parte de Matching de las comunicaciones colectivas, con el fin de ayudar a entender mejor toda esa funcionalidad. La Figura 3.11 ilustra el logo de esta aplicación.

Figura 3.11: Logo de Draw.io

Capítulo 4

Fundamentos/Estado del arte

En este capítulo se detallará en profundidad el funcionamiento del framework VEF Traces y otros conceptos fundamentales que son necesarios para la comprensión de la naturaleza del problema.

4.1. Framework VEF Traces

Como se ha mencionado en la sección 1, el framework VEF Traces [7] es un conjunto de herramientas de código abierto desarrolladas para facilitar el modelado y la caracterización de las comunicaciones generadas por aplicaciones basadas en MPI, así como reproducir estas comunicaciones en simuladores de redes de interconexión de altas prestaciones. Esencialmente, el framework de trazas VEF consta de dos herramientas principales. La primera, *VEF-Prospector* (Sección 4.1.1), que captura el tráfico de red y genera trazas en formato VEF (Sección 4.1.2). Estas trazas de tráfico almacenan las operaciones de comunicación, tanto punto a punto como colectivas, y pueden utilizarse para alimentar cualquier simulador de red de terceros, siempre que este simulador utilice la biblioteca *VEF-TraceLib* (Sección 4.1.3), que se encarga de reproducir el tráfico de la aplicación.

4.1.1. Generación de las trazas: *VEF-Prospector*

El primer paso para obtener las trazas de una aplicación MPI es instrumentalizar la aplicación. Para ello, se utiliza una herramienta de instrumentación intermediaria para obtener la traza y convertirla al formato de traza VEF. *VEF-Prospector* [1, 3, 7] es un conjunto de bibliotecas y aplicaciones dedicada a esta tarea. En primer lugar, se interceptan las llamadas de la aplicación a las bibliotecas MPI utilizando `vmpirun`. Vmpirun es un script envoltorio que precarga la biblioteca *VEF-Prospector* y posteriormente lanza la aplicación MPI con sus argumentos utilizando mpirun. Al precargar *VEF-Prospector*, la biblioteca captura las llamadas a las funciones MPI, interactuando con el driver MPI a través de la interfaz de profiling

de MPI, llamada PMPI (Profiling MPI) para que ejecute realmente las comunicaciones. El resultado de este proceso es la creación de archivos temporales que contienen las llamadas para cada tarea MPI. Estos archivos temporales se componen de dos tipos de ficheros:

- Los archivos x.veft contienen cada llamada MPI y una marca temporal absoluta que marca el inicio y el fin de esa comunicación.
- Los archivos x.comm contienen los comunicadores que ha utilizado la tarea x durante la ejecución de la aplicación.

Además, el paquete puede ser usado para hacer profiling de las aplicaciones MPI, detectar llamadas MPI que no están soportadas en el formato de traza, y capturarlas, a pesar de que esas llamadas no se pueden modelar dentro de las trazas VEF. En la Figura 4.1 se muestra una imagen del funcionamiento general de la generación de trazas [7].

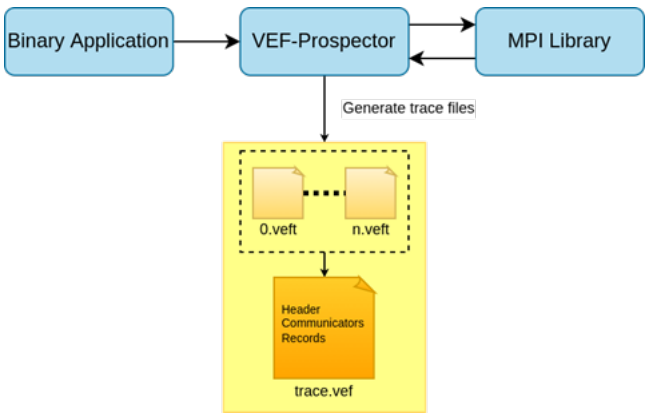

Figura 4.1: Esquema general generación trazas VEF

Por ejemplo, supongamos que la aplicación binaria realiza una tarea que utiliza la función MPI_Send. Entonces *VEF-Prospector* intercepta la llamada, recoge la información y llama a la función de profiling análoga PMPI_Send, que ejecuta la biblioteca de MPI. MPI devuelve el resultado a *VEF-Prospector*, y éste a su vez le devuelve el resultado a la aplicación. Así, cada tarea MPI generará un fichero con la información de todas sus comunicaciones.

El segundo paso consiste en la mezcla y procesado de estos ficheros temporales y su conversión a un fichero en formato .vef mediante la herramienta *VEF Mixer*. La traza final contiene toda la información necesaria para reproducir las comunicaciones MPI, pero los valores de las marcas de tiempo de cada evento ya no son absolutos, si no que son tiempos relativos entre las diferentes comunicaciones de cada tarea. De esta forma, al modificar la red simulada, cualquier variación en el tiempo de ejecución de una comunicación se reflejará en el tiempo de ejecución del sistema simulado. Con tiempos absolutos, el tiempo final lo marcaría siempre la ejecución del último registro. Este sistema de tiempos relativos y dependencias entre comunicaciones hace que las trazas VEF se denominen trazas *autorelacionadas*.

Una vez generadas las trazas VEF, se pueden analizar antes de ser utilizadas para la generación de tráfico en un simulador. Las herramientas que nos permiten analizar las trazas VEF son `tracetor` y `offline-vef-analysis.sh`. Por una parte, la herramienta **tracetor** (incluida en VEF-TraceLib) permite comprobar si la traza generada es funcional y no se ha generado con errores, generando una serie de estadísticas como el número de mensajes enviados, la productividad en bytes o el tiempo de ejecución al reproducirla en una red muy simple que modela las latencias de los mensajes mediante fórmulas matemáticas (no simula una red de verdad). Por otra parte, la herramienta **offline-vef-analysis.sh** es un script que analiza la traza VEF y proporciona un gran número de gráficos, archivos de texto e informes PDF con información sobre la generación de mensajes, la distribución de la generación de destinos, el tamaño de las cargas de trabajo, número y tipo de operaciones colectivas, etc.

4.1.2. Formato de las trazas VEF

La traza se compone de varios registros, siendo cada línea de la traza un registro independiente. Podemos distinguir tres tipos de registros en la traza VEF: la cabecera de la traza y los comunicadores, que contienen información global de la aplicación, y los registros de comunicación, que contienen información de las comunicaciones MPI llamadas por cada tarea MPI: [1][7]

- Cabecera de la traza: Es la primera línea del archivo. Contiene información básica de la traza, como el número de tareas MPI, el número de mensajes punto a punto o el número de comunicadores. Su formato es el siguiente:

  ```
  VEF3 nTareas nMsgs nCOMM nCollComm nLocalCollComm nRecvDep clock
  ```

  donde:

  - VEF3 hace referencia a la versión utilizada y debe ser la primera palabra escrita en cualquier fichero traza VEF. Actualmente se utiliza la tercera versión del formato, aunque aún existe soporte para la versión 2.
  - nTareas es el número de tareas MPI.
  - nMsgs es el número de registros referentes a comunicaciones punto a punto.
  - nCOMM es el número de comunicadores MPI.
  - nCollComm es el número de comunicaciones de colectivas globales (es decir, el número de comunicaciones colectivas realizadas por la aplicación).
  - nLocalCollComm es el número de registros de comunicaciones colectivas que contiene la traza. Cada comunicación colectiva "global" (es decir, referente a un comunicador) tiene múltiples comunicaciones colectivas "locales" (referentes a una tarea).
  - nRecvDep es un flag de optimización utilizado en anteriores versiones del formato, actualmente es un parámetro obsoleto y se mantiene por compatibilidad.
  - clock es el periodo de reloj de la traza medido en picosegundos.

- Comunicadores o COMMS. Una COMM especifica las tareas MPI implicadas en el intercambio de mensajes generados por una comunicación colectiva. Una tarea MPI puede utilizar varias COMMs para comunicarse con diferentes grupos de tareas MPI diferentes. Su formato es el siguiente:

  `id task0 [task1 ... taskN-1]`

  donde:

  - id es el identificador del comunicador que se compone de la letra "C" seguido de un número natural único.
  - task0 [task1 ... taskN-1] son identificadores de las tareas involucradas por el comunicador

- Registros de comunicación. Estos registros contienen la información básica sobre las comunicaciones. Modelan las llamadas a funciones MPI (ya sean punto a punto o colectivas) realizadas por una tarea MPI específica. Los campos típicos de un registro de comunicación son la tarea de origen, la tarea de destino (o la COMM en comunicaciones colectivas), el tamaño del mensaje, el tiempo de procesamiento necesario para realizar determinadas tareas (por ejemplo, procesamiento de la CPU, lectura/escritura desde/a la memoria secundaria, etc.) antes de lanzar la siguiente operación de comunicación en el nodo final, y un campo específico que indica si ese registro tiene una dependencia con otras tareas de la traza.

  Se distinguen dos tipos de registros diferentes. Por un lado, están los registros que se corresponden con las comunicaciones punto a punto. Por otro lado, están los registros que se corresponden con las comunicaciones colectivas. Se puede obtener información más detallada de estos registros aquí [5].

  En la Figura 4.2 se muestra un ejemplo de un archivo de traza VEF con todos los tipos de registros.

```
 1 VEF3 240 4793846 147 20518 214340 0 1000                    //Cabecera
 2 C0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
 3 C4 2 10 18 26                                               // Comunicadores
 4 C5 3 11 19 27
 5 C6 4 12 20 28
 6 C7 5 13 21 29
 7 0 17 16 4 4 681062 -1                                       /* Comunicaciones
 8 1 9 8 4 4 692353 -1                                         punto a punto
 9 3 17 17 0 1 4135 0                                          (send, receive
10 4 25 24 4 4 701573 -1                                       y dependencias de colectivas) */
11 G0 C0 0 151 0 4 0 6660046 -1                                /* Comunicaciones colectivas
12 G0 C0 0 24 0 4 0 6757082 -1                                 (registros independientes) */
13 G7 C0 0 0 4 0 3 48 G6                                       /* Comunicaciones colectivas
14 G8 C0 0 0 9 0 3 23 G7                                       (send,receive y dependencias de colectivas) */
```

Figura 4.2: Ejemplo de un archivo de traza VEF

4.1.3. Reproducción de las trazas: la biblioteca *VEF-TraceLib*

VEF-TraceLib [1] es una librería de código abierto, escrita en ANSI C, que ofrece la funcionalidad necesaria para reproducir el comportamiento de las trazas VEF en un simulador

de redes de interconexión. Proporciona un conjunto de funciones para la lectura de trazas, la asignación de tareas a nodos finales, la gestión de la ejecución de trazas y la comunicación entre la biblioteca y el simulador. *VEF-TraceLib* realiza todas estas funciones de forma transparente, de modo que el simulador de red sólo tiene que pedir a la biblioteca que le proporcione los mensajes de la aplicación almacenados en una traza VEF. Más concretamente, el simulador de red obtiene los mensajes de *VEF-TraceLib*, los inyecta en la red y devuelve de nuevo el control a la biblioteca cuando se reciben los mensajes en los nodos finales. Sin embargo, el simulador debe dividir estos mensajes en paquetes, inyectarlos en la red y reunirlos en los nodos finales de destino antes de informar a *VEF-TraceLib* de que un mensaje se ha recibido completamente.

Para cada mensaje generado, *VEF-TraceLib* [4] sólo proporciona al simulador los nodos finales de origen y destino, la longitud del mensaje y un id utilizado para identificar ese mensaje dentro de *VEF-TraceLib*. Este id es generado mediante una función hash, es único para cada mensaje y encapsula toda la información necesaria para su gestión. Gracias a esto, la gestión del mensaje para el simulador es totalmente transparente: no necesita saber si un mensaje específico fue generado por una comunicación punto a punto o por una comunicación colectiva, si una tarea está detenida esperando la recepción de un mensaje que el simulador aún no ha proporcionado a *VEF-TraceLib*, etc. Todo esto es gestionando por la biblioteca de forma transparente gracias a este id [7].

4.2. Fundamentos de MPI

En esta sección se abordan los conceptos básicos de MPI y las funciones más relevantes, con la intención de facilitar la comprensión del siguiente capítulo, en el cual se explican los algoritmos del programa original y los cambios realizados en la lógica para llevar a cabo la paralelización.

El estándar Message Passing Interface [11] (o MPI, abreviado) es una interfaz de programación de aplicaciones (API) que define adecuadamente la sintaxis y la semántica completa de una biblioteca de software que proporciona rutinas básicas estandarizadas para construir programas paralelos. Así, la interfaz MPI [12] permite codificar programas paralelos que intercambian datos enviando y recibiendo mensajes que encapsulan esos datos. Utilizar una API tiene la ventaja de dejar al programador libre de los muchos detalles relativos a la implementación de los procedimientos de red, y permite al ecosistema (academia, industria, programadores) beneficiarse de la interoperabilidad y la portabilidad de los códigos fuente. Es importante destacar que la API MPI no depende del lenguaje de programación subyacente que utilice. Así, podemos utilizar comandos MPI con los lenguajes de programación (secuenciales) más comunes como C, C++, Java, Fortran Python, etc. Es decir, existen varios lenguajes de programación de la API MPI.

MPI es la interfaz de programación dominante para algoritmos paralelos con memoria distribuida en la comunidad HPC. Esto ocurre principalmente gracias a la estandarización de muchas rutinas globales de comunicación y primitivas para realizar cálculos globales.

Siguiendo el modelo SPMD [13] (Simple Program Multiple Data), el usuario escribirá su

aplicación como un proceso secuencial del que se lanzarán varias instancias que cooperarán entre sí. Existen 2 modos de comunicación en MPI:

- Punto a punto. Los mensajes punto a punto son transferencias de datos llevadas a cabo por un emisor concreto y especificando un receptor determinado. Dentro de este tipo de comunicaciones, se distinguen 2 modos: síncrono o asíncrono, y bloqueante o no bloqueante.
- Colectivas. Son comunicaciones coordinadas que pasan datos a través de un grupo de procesos. Todos los procesos dentro de un grupo de procesos tiene que llamar a la función con los mismos parámetros.

Los procesos invocan diferentes funciones MPI que permiten:

- Iniciar, gestionar y finalizar procesos MPI.
- Comunicar datos entre dos procesos.
- Realizar operaciones de comunicación entre grupos de procesos.
- Crear tipos arbitrarios de datos.

Todas las funciones MPI se escriben como MPI_X, siendo X el tipo de función. Por ejemplo, si se habla de una función de Broadcast en colectivas se debe poner MPI_Broadcast. En lo que resta del documento, se nombrarán sin el prefijo "MPI_" para facilitar la lectura del mismo. Algunas de las funciones de comunicaciones colectivas más conocidas son:

- Barrier: Sincroniza todos los procesos.
- Broadcast: Envía datos de un proceso a todos los procesos.
- Gather: Reúne datos de todos los procesos a un proceso.
- Scatter: Dispersa datos de un proceso a todos los procesos.
- Reduce: Añade, multiplica, etc., datos distribuidos, dejando el resultado en un única tarea.
- AllReduce: Igual que la comunicación Reduce, pero todas las tareas envueltas en la operación obtienen el resultado final. Semánticamente, sería equivalente a ejecutar un Reduce seguido de un Broadcast.

Para comprender bien el siguiente capítulo (Capítulo 5), se procede a explicar las funciones P2P [16] en profundidad.

MPI tiene un amplio espectro de funciones punto a punto. Las más comunes son Send y Recv, incluyendo sus versiones no bloqueantes ISend e Irecv, para enviar y recibir datos explícitamente por el emisor y el receptor, respectivamente.

En la **comunicación asíncrona**, el remitente envía un mensaje sin saber cuándo será entregado o recibido por el destinatario. El destinatario no sabe cuándo se envió el mensaje. En la **comunicación síncrona**, el remitente sabe cuándo el destinatario ha recibido el mensaje. El destinatario sabe cuándo envió el mensaje el remitente.

En la **comunicación bloqueante**, el proceso se bloquea hasta que la operación se completa. En la **comunicación no bloqueante**, un proceso puede continuar con sus siguientes operaciones sin tener que esperar a que se complete. El proceso no bloqueado siempre puede averiguar el estado de la operación no bloqueante (mediante las funciones Test y Wait). Todas las operaciones no bloqueantes deben tener operaciones de espera (Wait) correspondientes, ya que algunos sistemas no pueden liberar recursos hasta que se haya llamado a la operación de espera. Las fases en una comunicación de este tipo son:

- Iniciar la comunicación no bloqueante
- Realizar otras funciones
- Esperar a que la comunicación no bloqueante se complete

En la Tabla 4.1, se recogen las distintas funciones de envío y recepción de las comunicaciones punto a punto en MPI.

Función	**Información**
Send	Envío estándar bloqueante
Isend	Envío estándar no bloqueante
Recv	Recepción Bloqueante
IRecv	Recepción no bloqueante
SSend	Synchronous Send. Bloqueante y síncrono
BSend	Buffered Send. No bloqueante y asíncrono
RSend	Ready Send, No bloqueante
ISSend	No bloqueante y síncrono

Tabla 4.1: Funciones P2P

4.3. Fundamentos de OpenMP

En esta sección se realiza una breve introducción a OpenMP, que será la herramienta que se utilice para desarrollar la versión paralela de la aplicación *VEF Mixer*.

OpenMP (Open Multi-Processing) [30], o abreviado OMP, es una interfaz de programación de aplicaciones (API) para la programación multiproceso de memoria compartida en múltiples plataformas: C/C++ y Fortran. La API OpenMP define un modelo portátil y escalable con una interfaz sencilla y flexible para desarrollar aplicaciones paralelas en plataformas que van desde el escritorio hasta el superordenador.

Se compone de un conjunto de directivas de compilador y rutinas de biblioteca para programadores de aplicaciones paralelas. La mayoría de construcciones de OMP son directivas de compilador, que reciben el nombre de cláusulas.

Se utiliza un modelo de paralelismo de hilos basado en fork-join [17, 18], en el que el hilo maestro despliega un conjunto de hilos según se precise. Todos los hilos desplegados comparten el mismo espacio de memoria. Por esta razón, es necesaria la sincronización de los hilos, establecer orden y proteger el acceso a los datos compartidos. Si no se realiza una sincronización adecuada, se producen condiciones de carrera, que son cambios indeseados en el funcionamiento del programa debido a conflictos entre los múltiples hilos intentando acceder al mismo espacio de memoria.

OpenMP funciona muy bien cuando se utiliza con bucles `for`. Permite dividir los hilos desplegados en las iteraciones del bucle con facilidad.

El formato general de una directiva en C es:

```
#pragma omp construct [opciones]
```

donde construct es el tipo de cláusula a establecer, y opciones son los atributos que se vinculan con esa cláusula, que puede ser opcional.

Para conseguir un nivel de sincronización mayor se utilizan principalmente 2 cláusulas:

- critical
- atomic

La cláusula `critical` establece una zona como sección crítica y aplica exclusión mutua, de manera que sólo un hilo puede acceder a los datos de la sección crítica, el resto de hilos se queda esperando. La cláusula `atomic` establece una operación como atómica, proporciona exclusión mutua pero sólo sobre la actualización de un espacio de memoria.

En la paralelización de *VEF Mixer* se usan sobre todo estas cláusulas junto con la combinación de sus opciones.

Capítulo 5

Diseño de algoritmos

En este capítulo se realiza un análisis del funcionamiento y de los algoritmos de la aplicación *VEF Mixer*. Seguidamente, se describen las modificaciones realizadas en los algoritmos existentes para su funcionamiento en paralelo.

La aplicación *VEF Mixer* está formada por un conjunto de programas escritos en C. El programa principal es `vef_mixer.c`, que contiene el punto de entrada del ejecutable, es decir, la función main, la cual es la primera que se invoca al iniciarse la aplicación.

En el funcionamiento general de la aplicación *VEF Mixer* se distinguen tres partes bien diferenciadas: la lectura de los eventos de cada tarea, su asociación (Matching) con su evento pareja o comunicador y el volcado en el fichero de salida en formato VEF. Estas partes se llevan a cabo por 2 funciones: `read_event`, encargada de la lectura y el matching, y `dump_vef_trace`, encargada del volcado.

El fichero `event_queue.c` contiene todas las funciones relacionadas con la lectura y gestión de los eventos y sus colas, y la escritura en el fichero de volcado. Los dos ficheros `P2P_Matching.c` y `GRP_Matching.c` se encargan del Matching de los eventos de su tipo, es decir, comunicaciones punto a punto, o comunicaciones colectivas, respectivamente. El fichero `vef_comm.c` contiene funciones de utilidad para las comunicaciones colectivas, como generar y buscar un identificador global de la comunicación, obtener el tamaño de la comunicación, o comprobar si una tarea pertenece a un comunicador en concreto, entre otras.

En primer lugar se validan todos los argumentos del programa, que deben ser 4 generalmente, siguiendo el formato de comando:

```
VEF-Prospector/vef_mixer -i rutaTemporales/VEFT.main -o traza.vef
```

donde rutaTemporales es la ruta que contiene todos los ficheros temporales .veft y .comm generados anteriormente con vmpirun. Después, se procede a leer el fichero que se le ha pasado, que contiene todas las tareas MPI, sus llamadas y la información de los comunicadores de cada tarea.

Antes de proceder a explicar los algoritmos se necesita conocer varios términos y definiciones. El Matching al que se va a hacer referencia en este documento reiteradas veces consiste en la asociación de eventos MPI que proceden de las tareas MPI del fichero que recibe el programa como argumento. Como se ha introducido en la Sección 4.2, los eventos pueden ser punto a punto o de colectivas.

La comunicación de los eventos punto a punto siempre tiene un emisor y un receptor. Una parte envía datos y la otra los recibe. Pueden distinguirse operaciones bloqueantes o no bloqueantes. Todas las operaciones no bloqueantes deben tener **operaciones de espera (Wait)** correspondientes, ya que algunos sistemas no pueden liberar recursos hasta que se haya llamado a la operación de espera. El proceso no bloqueado siempre puede averiguar el estado de la operación no bloqueante (mediante las funciones Test y Wait).

En la comunicación de los eventos con colectivas se agrupan operaciones globales que realiza un conjunto de procesos utilizando un comunicador para cada grupo, que permite comunicarse entre ellos, separando los contextos en los cuales se llevan a cabo las comunicaciones.

En este programa se guarda toda la información de los eventos en un tipo de dato denominado `VEF_event`, que guarda la información de la llamada MPI, el estado de asociación, estado de volcado, el comunicador de las colectivas y el evento Wait para las comunicaciones P2P no bloqueantes. Para este último tipo de comunicaciones, P2P, además se guardan otros datos como el objetivo o "target"del evento, o identificadores punto a punto.

La llamada a MPI es un tipo de dato `MPI_event`, que guarda datos de la comunicación de la tarea MPI. Se trata de información como la request, tipo de evento, tag, comunicador, longitud, objetivo o "target"de la tarea y los tiempos de duración del evento MPI. Conviene saber de cara a las futuras explicaciones que el "target" es el objetivo del evento, es decir, a quién envía los datos o de quién los recibe, y la "request" es el identificador de los eventos Wait, y es único para cada evento no bloqueante y su Wait asociado.

5.1. Diseño original

En el diseño original, la parte de asociación o Matching estaba incorporada en la función de lectura de los eventos, de tal manera que una vez leídos y confirmados los eventos, se llamaba a la función de Matching correspondiente para proceder a su asociación.

En el Listado 1 se puede ver el bucle principal del programa, en el fichero `vef_mixer.c`, en el que la condición de entrada llama a la función `are_events_to_dump`, que comprueba si existen eventos para volcar o "dumpear". Después hay un bucle `for` que va recorriendo las tareas MPI. Para cada tarea, se comprueba si existen eventos para leer, y en ese caso, se llama a la función `read_event`, para leer nuevos eventos de la tarea desde su fichero *.veft*. Las variables `reads` y `chunk` se utilizan como administradoras de la cantidad máxima de eventos que se pueden leer, que son 1000. Se van leyendo eventos de cada tarea de millar en millar. Tras la lectura y asociación de dichos eventos, se procede a su volcado en la función `dump_vef_trace`.

```
while (are_events_to_dump(ANY_SOURCE) != END_OF_VEFT_FILE) {
        //first read
        TIME init_match = getRealTime();
        for (int task = 0; task < vef_info.numTask; task++) {
            int32_t chunk = 1000;
            int32_t reads = 0;
            int8_t flag;
            while (are_events_to_read(task) == OPERATION_COMPLETE && reads < chunk) {
                if ((out = read_event(task, &flag)) != OPERATION_COMPLETE) {
                    fnError("Error reading an Event. %s", getProspectorOut(out));
                }
                reads += flag;
            }
        }
        TIME end_match = getRealTime();
        matching_time += (end_match - init_match);
        //dump the trace
        dump_vef_trace();
        dumping_time += (getRealTime() - end_match);
        print_progress();
    }

```

Listado 1: Bucle del programa principal vef_mixer.c

5.1.1. Lectura y asociación de eventos

A continuación, se muestra en los Listado 2 y 3 el algoritmo de lectura de eventos, `read_event`. En primer lugar (Listado 2), se asigna espacio si hay hueco para ese evento en un nuevo chunk de la lista de eventos, `VEF_events`, llamando a la función `getEventSlot`. Una vez asignado espacio, se llama a la función `readNextEvent`, que se encarga de leer el evento MPI del fichero en formato veft y de inicializar el resto de variables del evento. A continuación, se actualiza la información de la tarea y se obtiene el alcance o "scope"del evento, que determina si es comunicación colectiva (GRP_COMM), una comunicación punto a punto (P2P), o se trata de un evento no soportado por el framework (NOT_SUPPORTED).

Según el alcance del evento se gestionará de una forma u otra. Si es colectiva, se obtiene el identificador del comunicador global y el número de tareas de dicho comunicador. Si el comunicador solo tiene 1 tarea, no es una comunicación colectiva, y por tanto, se establece el alcance a no soportado. Si la operación no es soportada, se actualizan las estadísticas y se libera el recurso. Antes de gestionar el evento con alcance P2P, se confirma la inserción del evento en la lista con la función `confirmNewEvent`, que actualiza el chunk y el número de elementos de la lista de eventos `VEF_events` de esa tarea.

Seguidamente (Listado 3), se gestiona el evento en caso que sea P2P. Se comprueba si es una comunicación no bloqueante tanto para los envíos (ISend, IBSend, ISSend, IRSend) como para las recepciones (IRecv). En caso afirmativo, se procede a la búsqueda y eliminación de las request de los Wait asociados a las llamadas no bloqueantes. Si el Wait es no soportado, se pone la request del evento origen a 0. Si encuentra la request del Wait y es un evento

```
prospector_out read_event(int32_t task, int8_t * new_event_stored) {
    VEF_event* source_ev = getEventSlot(task);
    if (source_ev == NULL)
        return ALLOCATION_FAILED;
    prospector_out out;
    if ((out = readNextEvent(task, source_ev)) != OPERATION_COMPLETE)
        return out;

    //update the task info
    info[task].read_events++;
    if (info[task].total_events == info[task].read_events)
        read_tasks++;
    t_mpi_scope scope = GET_EVENT_SCOPE(source_ev->mpi_ev.type);
    if (scope == GRP_COMM) {
        uint16_t gid;
        if (global_comm_id(source_ev->mpi_ev.comm, task, &gid) == OPERATION_COMPLETE) {
            uint32_t size;
            getCommSize(gid, &size);
            if (size == 1) {
                //this avoid collective operations of one task
                scope = NOT_SUPPORTED;
            }
        }
    }
    if (scope == NOT_SUPPORTED) {
        info[task].match_events++;
        info[task].dump_events++;
        dumped_events++;
        *new_event_stored = 0;
        return OPERATION_COMPLETE;
    }
    if ((out = confirmNewEvent(task)) != OPERATION_COMPLETE)
        return out;
    *new_event_stored = 1;

```

Listado 2: Método read_event del fichero event_queue.c

IRecv, se comprueba si los datos son correctos en el remitente (sender).

La segunda parte consiste en el Matching de los eventos, haciendo las llamadas a las funciones `matching_P2P_events` (punto a punto) y `matching_GRP_events` (colectivas) de los ficheros `P2P_Matching.c` y `GRP_Matching.c`, respectivamente. Por último, en caso de que el alcance del evento no sea de los tipos punto a punto o colectiva, significa que son los eventos de inicio o finalización (`MPI_INIT` o `MPI_FINALIZE`) del cómputo MPI, y se sincroniza esa tarea y se encola para su volcado, actualizando las estadísticas. Por último, se sincronizan los tiempos del evento con la función `syncEvent`. Esta función simplemente recalcula los tiempos del evento en base al inicio de la ejecución de la tarea, marcado por el tiempo de la operación MPI_INIT.

```
    if (scope == P2P) {
        //Is the event a non blocking call?
        if (IS_NON_BLOCKING_SEND(source_ev->mpi_ev.type)
        || source_ev->mpi_ev.type == IRecv) {
            uint32_t req = source_ev->mpi_ev.request;
            //If deleting the request (the hash key) fails,
            //this non blocking call has no wait associated
            infoWait info;
            int8_t supported = 0, found = 0;
            if ((out = hash_search_and_delete(twait + task,
            int2voidp req, &(info.pointer))) == OPERATION_COMPLETE) {
                supported = found = 1;
            } else if (source_ev->mpi_ev.type == IRecv) {
                //If the event is a IRecv, maybe is a Ignored Wait
                if (hash_search_and_delete(twait_ign + task, int2voidp req,
                &(info.pointer)) == OPERATION_COMPLETE) {
                    found = 1;
                }
            }
            if (!supported)
                source_ev->mpi_ev.request = 0;
            if (source_ev->mpi_ev.type == IRecv && found) {
                //check the correctness of the data;
                if (source_ev->mpi_ev.target_task < 0) {//first, check the sender
                    source_ev->mpi_ev.target_task = info.sender;
                } else if (source_ev->mpi_ev.target_task != info.sender) {
                    fnError("The sender in a MPI_Irecv
                    and its Wait are not the same\n");
                }
                if (source_ev->mpi_ev.lenght < info.message_size)
                    fnError("The message lenght in a MPI_Irecv
                    is lower than the received message size\n");
                source_ev->mpi_ev.lenght = info.message_size;
            }
        }
    if ((out = matching_P2P_events(task, source_ev)) != OPERATION_COMPLETE)
            return out;
    } else if (scope == GRP_COMM) {
        if ((out = matching_GRP_events(task, source_ev)) != OPERATION_COMPLETE)
            return out;
    } else {
        info[task].match_events++;
        if (IS_MPI_INIT(source_ev->mpi_ev.type)) {
            set_zero_time(task, source_ev->mpi_ev.init_time);
            queue_task(task, 0);
        }
        if (IS_MPI_FINALIZE(source_ev->mpi_ev.type)) {
            if (info[task].read_events != info[task].total_events)
                fnError("There are more events after MPI_Finalize\n");
        }
    }
    syncEvent(task, source_ev);
    return OPERATION_COMPLETE;
}
```

Listado 3: Método read_event del fichero event_queue.c. Parte 2

A continuación se detalla el algoritmo del Matching de los eventos P2P y GRP. Al principio del capítulo se mencionó el estado de asociación, un atributo presente en los eventos VEF. El estado de asociación permite conocer cuál es el nivel de asociación del evento con su pareja o con el comunicador en caso de colectivas, y será fundamental para entender el algoritmo de Matching. Un evento puede tener los siguientes estados de asociación, que conforman el modelo de estados:

- INIT: indica que aún no se ha hecho nada.
- P2P_TARGET_FOUND: Se ha encontrado el objetivo o target pero no el Wait.
- P2P_WAIT_FOUND: Se ha encontrado el Wait pero no el target.
- P2P_COMPLETE: Se ha encontrado toda la información necesaria y se ha completado el matching P2P.
- GRP_COMPLETE: Se ha completado el matching GRP.

Para ayudar a comprender mejor el algoritmo de Matching, se han creado una serie diagramas de estados con las posibles casuísticas. Estos diagramas se van a utilizar para acompañar la explicación del Matching de las comunicaciones punto a punto, que son las que más complejidad tienen.

En el Listado 4, se encuentra el inicio de la función `matching_P2P_events`. Se determina el objetivo del evento. Para los eventos Wait, la tarea destino debe ser la misma que la tarea origen (línea 4), ya que el destino será siempre una comunicación no bloqueante realizada previamente, y el origen el Wait, y ambos pertenecen a la misma tarea.

```
1     //determine the target task
2     int32_t destination;
3     if (IS_SUPPORTED_WAIT(src_ev->mpi_ev.type)) //In the wait case, we search the
  ↪   associatted Irecv|ISend in the source task event list
4         destination = src;
5     else //in other case, we searh the associated mpi call in the target task event
  ↪   list
6         destination = src_ev->mpi_ev.target_task;
7     //uses a list iterator to search the target event in the list
8     VEF_event_iterator iter;
9     init_event_queue_iterator(&iter, VEF_events + destination);
10    while (event_queue_has_next(&iter) != EMPTY_DATA_STRUCT && src_ev->target_ev ==
  ↪   NULL) {
11        VEF_event* dst_ev = event_queue_next(&iter);
12        VEF_matching_state dst_state = dst_ev->match_state;
13        t_mpi_event ttype = (t_mpi_event) dst_ev->mpi_ev.type;
14        if (GET_EVENT_SCOPE(ttype) == P2P
15                && dst_state < P2P_COMPLETE) {
16            switch (src_ev->mpi_ev.type) {
17
```

Listado 4: Matching P2P

Se utiliza un iterador[1], llamado `iter`, que va a recorrer la lista de eventos con la finalidad de encontrar el objetivo del evento, es decir, la parte a quién se envían datos o de quién se reciben los datos. Para ello, se emplea un bucle while cuya condición de parada es que el siguiente evento no tenga objetivo y no haya más eventos en el iterador. Eso significaría que no hay más eventos con los que se pueda asociar. El matching solo se hará si el alcance del evento es P2P y el estado indica que no se ha completado aún (P2P_COMPLETE), ya que si estuviera completado no tendría sentido seguir intentando asociarlo, pues ya se ha asociado anteriormente. Se utiliza un switch para abarcar todas las posibles casuísticas.

```
case Send:
case BSend:
case RSend:
case SSend:
case SendRecv_S:
    if (dst_state != P2P_TARGET_FOUND && dst_state != P2P_COMPLETE) {
        if (matching_Send(src, &(src_ev->mpi_ev), destination, &(dst_ev->mpi_ev)) ==
↪  OPERATION_COMPLETE) {
            VEF_event* sender = src_ev;
            VEF_event* receiver = dst_ev;
            receiver->target_ev = sender;
            if (IS_BLOCKING_RECEIVE(ttype)) {
                sender->is_trigger = 1;
                sender->target_ev = receiver;
                sender->match_state = receiver->match_state = P2P_COMPLETE;
                info[destination].match_events++;
                info[src].match_events++;
            } else { //The events is a IRecv
                if (dst_state == P2P_WAIT_FOUND) {
                    VEF_event* wait_ev = receiver->wait_ev;
                    sender->match_state = receiver->match_state = wait_ev->match_state
↪  = P2P_COMPLETE;
                    sender->is_trigger = 1;
                    info[destination].match_events += 2;
                    info[src].match_events++;
                    sender->target_ev = wait_ev;
                    wait_ev->target_ev = sender;
                } else {
                    if (receiver->mpi_ev.request == 0) {
                        sender->is_trigger = 0;
                        sender->match_state = receiver->match_state = P2P_COMPLETE;
                        info[destination].match_events++;
                        info[src].match_events++;
                    } else {
                        sender->match_state = receiver->match_state = P2P_TARGET_FOUND;
                        sender->is_trigger = 1;
                    } sender->target_ev = receiver;
                }
            }
        return OPERATION_COMPLETE;
```

Listado 5: Tratamiento de asociación para envíos bloqueantes

[1]La estructura y funciones del iterador son proporcionados por la propia biblioteca a través del fichero *list.c*.

En el caso de envíos bloqueantes (Listado 5), que son los tipos Send, BSend, RSend, SSend y SendRecv_S (información de profiling de la parte del envío de una función MPI_Sendrecv()), se procede de la siguiente manera. Si el remitente es el evento origen y el destinatario es el evento destino, se realiza el matching si el estado del evento es `INIT` o `P2P_WAIT_FOUND`. Es decir, no se realiza matching para este caso si el evento ya está completado o si ya se ha encontrado el objetivo o "target". La función `matching_Send` asocia el evento origen con su posible target si se cumplen las siguientes condiciones:

- El evento destino es una recepción.
- El objetivo del evento destino coincide con la tarea origen y el objetivo del evento origen coincide con la tarea de destino.
- La longitud del evento de origen y destino es la misma.
- El tag del evento de origen y destino son el mismo.

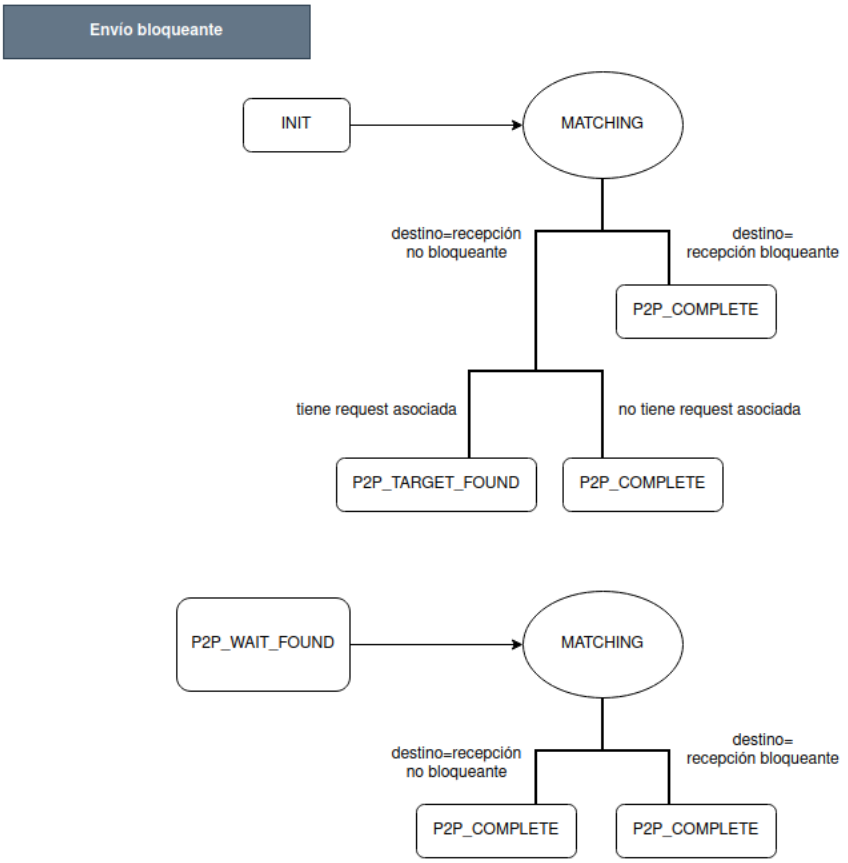

Figura 5.1: Diagrama de asociación de eventos para los envíos bloqueantes

```
case ISend:
case IBSend:
case IRSend:
case ISSend:
    if (dst_state != P2P_TARGET_FOUND && dst_state != P2P_COMPLETE) {
        if (matching_Send(src, &(src_ev->mpi_ev), destination, &(dst_ev->mpi_ev))
        == OPERATION_COMPLETE) {
            VEF_event* sender = src_ev; VEF_event* receiver = dst_ev;
            receiver->target_ev = sender;
            if (IS_BLOCKING_RECEIVE(ttype)) {
                sender->is_trigger = 1;
                sender->target_ev = receiver;
                receiver->match_state = P2P_COMPLETE;
                sender->match_state = (sender->mpi_ev.request == 0) ?
                P2P_COMPLETE : P2P_WAIT_FOUND;
                info[destination].match_events++;
            } else { //The events is a IRecv
                if (dst_state == P2P_WAIT_FOUND) {
                    sender->is_trigger = 1;
                    VEF_event* wait_ev = receiver->wait_ev;
                    receiver->match_state = wait_ev->match_state = P2P_COMPLETE;
                    sender->match_state = (sender->mpi_ev.request == 0) ?
                    P2P_COMPLETE : P2P_WAIT_FOUND;
                    info[destination].match_events += 2;
                    sender->target_ev = wait_ev;
                    wait_ev->target_ev = sender;
                } else {
                    if (receiver->mpi_ev.request == 0) {
                        sender->is_trigger = 0;
                        receiver->match_state = P2P_COMPLETE;
                        info[destination].match_events++;
                        sender->match_state = (sender->mpi_ev.request == 0) ?
                        P2P_COMPLETE : P2P_WAIT_FOUND;
                    } else {
                            sender->match_state =
                            receiver->match_state = P2P_TARGET_FOUND;
                            sender->is_trigger = 1;
                        }sender->target_ev = receiver;
                    }
            }
            if (sender->match_state == P2P_COMPLETE)
                info[src].match_events++;
            return OPERATION_COMPLETE;

```

Listado 6: Tratamiento de asociación para envíos no bloqueantes

Si el evento destino es una recepción también bloqueante, es decir, Recv o SendRecv_R, entonces se completan ambos eventos. En caso contrario, se comprueba si el evento destino ha encontrado su Wait asociado. Si es así, ambos eventos se completan y se asocian entre sí el evento Wait y el evento origen. Si no es así, y el destinatario no tiene Wait asociado, ambos eventos se completan. Pero en el caso de que el evento destino tenga Wait asociado y no lo haya encontrado aún, se establece los estados de los eventos origen y destino a `P2P_TARGET_FOUND`, quedándose pendientes de encontrar el evento Wait. Una vez que se

completan los eventos se actualizan las estadísticas correspondientes. En la Figura 5.1 se muestra la casuística para este tipo de funciones, es decir, los envíos bloqueantes.

En el caso de envíos no bloqueantes (Listado 6), la lógica es similar que en el caso de los envíos bloqueantes pero con una diferencia. Cuando se completan los eventos, en los mismos escenarios que se han explicado más arriba, se comprueba si el evento origen tiene request asociada. Si no la tiene se marca como completado. Si la tiene, se queda a la espera de que llegue su evento Wait asociado, marcando su estado como `P2P_WAIT_FOUND`. Esto se puede observar en las líneas 15, 23 y 33. Esto es debido a que el evento origen es no bloqueante, y debe asociarse con un evento Wait como se ha visto en la sección 4.2. En las Figuras 5.2 y 5.3 se muestra la casuística para este tipo de funciones, es decir, los envíos no bloqueantes. La primera muestra el tratamiento para estos eventos cuando el evento se encuentra en un estado inicial, `INIT`. La segunda muestra el tratamiento para estos eventos cuando se ha encontrado el evento Wait asociado, el estado del evento destino es `P2P_WAIT_FOUND`.

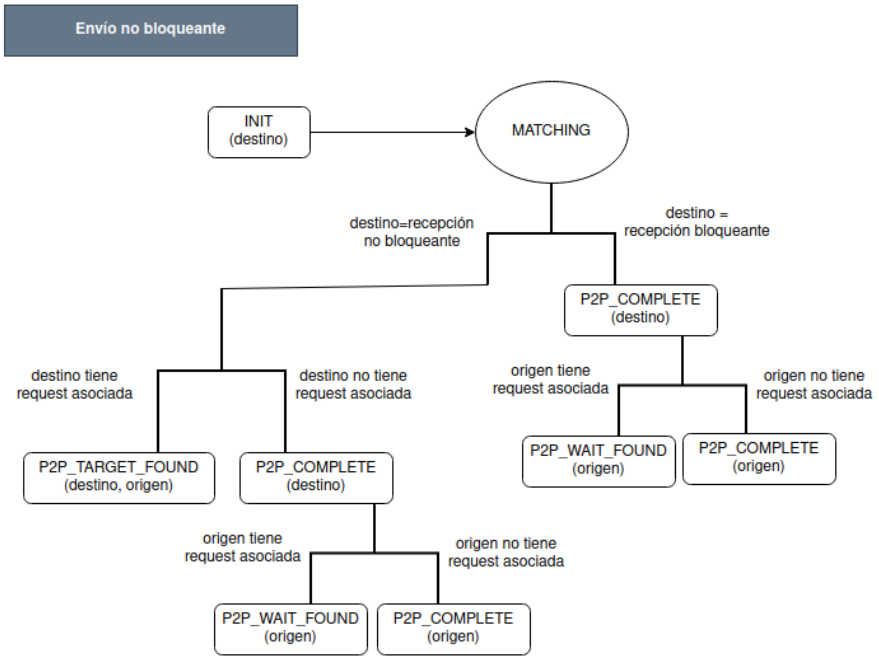

Figura 5.2: Diagrama de Matching para los envíos no bloqueantes con estado `INIT`

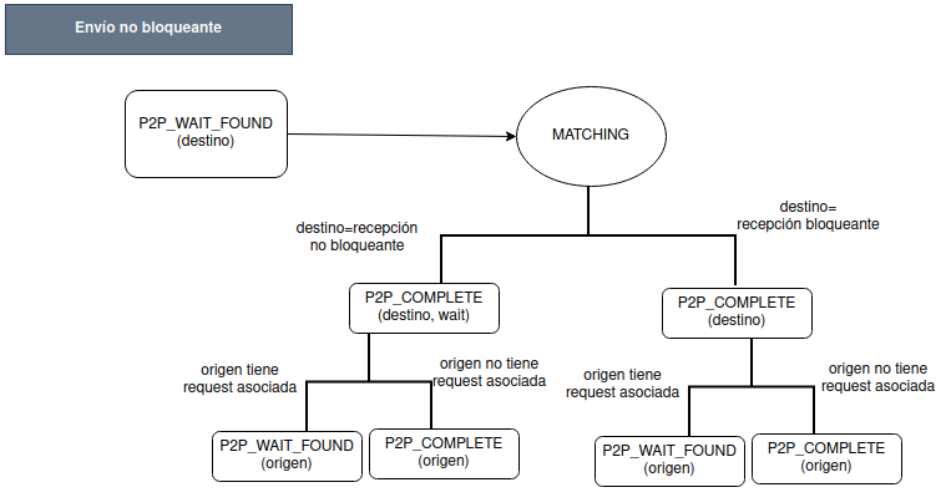

Figura 5.3: Diagrama del Matching para los envíos no bloqueantes con estado `P2P_WAIT_FOUND`

En el caso de las recepciones, se muestra el tratamiento de asociación para las recepciones bloqueantes y no bloqueantes en el Listado 7. En ambos se realiza el matching solo si aún no se ha hecho nada con el evento, es decir, su estado es `INIT`. El remitente es el evento destino y el destinatario es el evento origen. Al igual que en los envíos, se llama a la función análoga `matching_Recv`, que tiene la misma funcionalidad salvo la comprobación de que el evento destino es un envío en lugar de una recepción. Es decir, asocia el evento en función de si:

- El evento destino es un envío.
- El objetivo del evento destino coincide con la tarea origen y el objetivo del evento origen coincide con la tarea de destino.
- La longitud del evento de origen y destino es la misma.
- El tag del evento de origen y destino son el mismo.

En el primer caso, se asocian el evento origen con el evento destino, y se marca el evento destino como completado en caso de ser de tipo bloqueante. En el caso opuesto se queda a la espera del evento objetivo marcándose con el estado `P2P_WAIT_FOUND`.

En el segundo caso, recepciones no bloqueantes (IRecv), se completa el matching si los eventos origen y destino no tienen Wait asociados. Si lo tienen, entonces se tiene que esperar a recibir el evento Wait asociado para terminar la comunicación entre ellos, marcando los estados de ambos eventos, origen y destino, como `P2P_TARGET_FOUND`. Se muestran los diagramas para estos casos en las Figuras 5.4 y 5.5, respectivamente.

```
case Recv:
case SendRecv_R:
    if (dst_state == INIT) {
        if (matching_Recv(src, &(src_ev->mpi_ev), destination, &(dst_ev->mpi_ev)) ==
↪   OPERATION_COMPLETE) {
        VEF_event* sender = dst_ev, receiver = src_ev;

        //asociate the destination event to the source event;
        receiver->target_ev = sender;
        receiver->match_state = P2P_COMPLETE;
        info[src].match_events++;
        //asociate the  source event to the destinantion event;
        sender->target_ev = receiver;
        sender->is_trigger = 1;
        // The target event is a blocking Send,  a Isend without request
        if (IS_BLOCKING_SEND(sender->mpi_ev.type)
            || sender->mpi_ev.request == 0) {
                sender->match_state = P2P_COMPLETE;
                nfo[destination].match_events++;
        } else
            sender->match_state = P2P_WAIT_FOUND;
        return OPERATION_COMPLETE;}
    }
    break;
case IRecv:
    if (dst_state == INIT) {
        if (src_ev->mpi_ev.target_task == MPI_ANY_SOURCE)
            fnError("Something was wrong: A Irecv has with MPI_ANY_SOURCE sender has
↪   not obtained the real sender during the preprocessing phase\n");
        if (matching_Recv(src, &(src_ev->mpi_ev), destination, &(dst_ev->mpi_ev)) ==
↪   OPERATION_COMPLETE) {
            VEF_event* sender = dst_ev;
            VEF_event* receiver = src_ev;
            receiver->target_ev = sender;
            sender->target_ev = receiver;
            if (receiver->mpi_ev.request == 0) {
                receiver->match_state = P2P_COMPLETE;
                sender->is_trigger = 0;
                if (IS_BLOCKING_SEND(sender->mpi_ev.type)
                || sender->mpi_ev.request == 0) {
                    sender->match_state = P2P_COMPLETE;
                    info[destination].match_events++;
                } else
                sender->match_state = P2P_TARGET_FOUND;
            } else {
                sender->is_trigger = 1;
                sender->match_state = receiver->match_state = P2P_TARGET_FOUND;
            }
            if (receiver->match_state == P2P_COMPLETE)
                    info[src].match_events++;
            return OPERATION_COMPLETE;
        }
    }
    break;

```

Listado 7: Tratamiento de asociación para recepciones bloqueantes y no bloqueantes

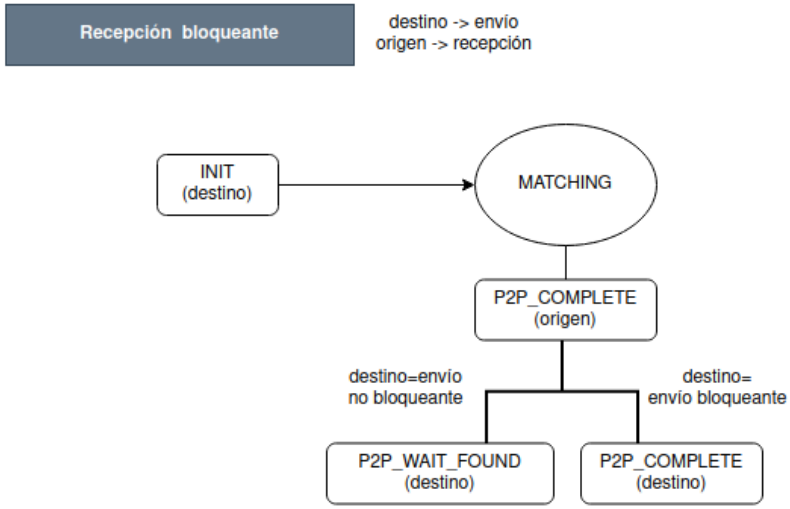

Figura 5.4: Diagrama del Matching para las recepciones bloqueantes

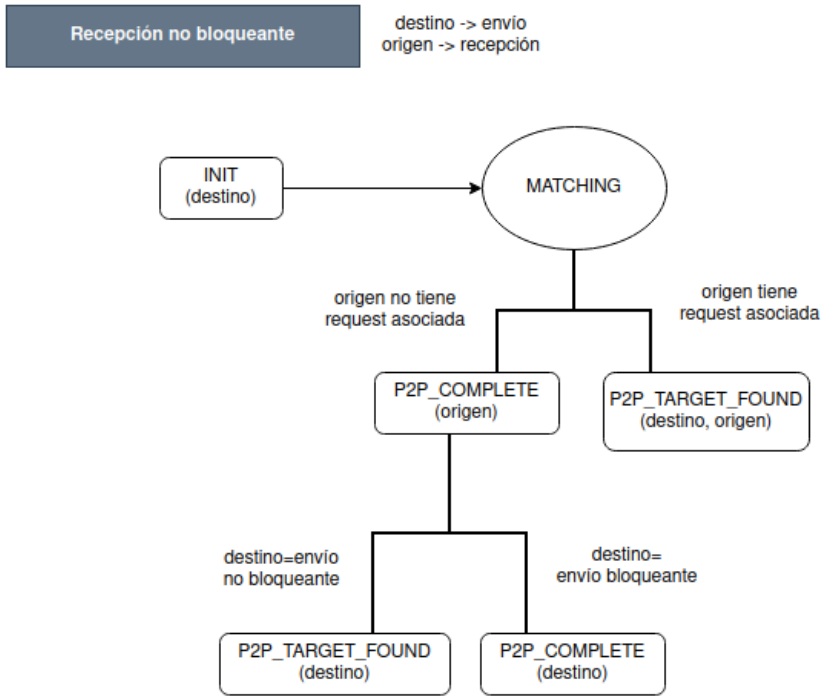

Figura 5.5: Diagrama del Matching para las recepciones no bloqueantes

Para el tratamiento de los eventos Wait, se distinguen dos casos: Wait asociado a envío no bloqueante (ISend) y Wait asociado a recepción no bloqueante (IRecv). En ambos casos, se realiza la llamada a la función `matching_Wait` que comprueba si el evento es una recepción no bloqueante IRecv o un envío no bloqueante, si la tarea origen es igual a la de destino y si el identificador de request de ambos eventos es el mismo.

En el caso del Wait asociado al IRecv (Listado 8), el evento destino será el destinatario y el evento origen será el evento Wait. Se completan los eventos destino y Wait si se ha encontrado el objetivo o target del evento destino, es decir, el estado es `P2P_TARGET_FOUND`. Se trata el objetivo del evento destino como el remitente (envío) de la comunicación, y se asocian entre sí el Wait y el envío. Si además el remitente no tiene request asociada o es bloqueante, se completa también. Si tiene Wait asociado entonces se queda a la espera del objetivo del evento, marcando su estado como `P2P_WAIT_FOUND`. En caso de que no se haya encontrado el objetivo del destinatario, se asigna su estado a `P2P_WAIT_FOUND` para que quede pendiente y se complete más adelante.

La lógica para el evento Wait asociado a ISend es algo distinta (Listado 9). El remitente (sender) es el evento destino, mientras que el evento origen será el Wait. Se hace el matching, se completa el evento Wait, y si el remitente ha encontrado el Wait pero no el objetivo, se marca como completado y se actualizan las estadísticas de los eventos.

```
case Wait_from_IRecv:
case Waitall_from_IRecv:
    if (matching_Wait(src, &(src_ev->mpi_ev), destination, &(dst_ev->mpi_ev)) ==
↪  OPERATION_COMPLETE) {
        VEF_event* receiver = dst_ev, *wait_ev = src_ev;

        if (receiver->match_state == P2P_TARGET_FOUND) {
            receiver->match_state = wait_ev->match_state = P2P_COMPLETE;
            //Note that src and destination are the same
            info[src].match_events += 2;
            VEF_event* sender = receiver->target_ev;
            sender->target_ev = wait_ev; //associate the wait to the send
            wait_ev->target_ev = sender; //associate the send to the wait

            if (IS_BLOCKING_SEND(sender->mpi_ev.type)
                || sender->mpi_ev.request == 0) {
                    sender->match_state = P2P_COMPLETE;
                    info[receiver->mpi_ev.target_task].match_events++;
                } else {
                    sender->match_state = P2P_WAIT_FOUND;
                }
            } else {
                //Target not found
                receiver->match_state = wait_ev->match_state = P2P_WAIT_FOUND;
                receiver->wait_ev = wait_ev;
                }
            return OPERATION_COMPLETE;
        }
    break;
```

Listado 8: Tratamiento Matching para los eventos Wait asociados a un evento IRecv

```
                case Wait_from_ISend:
                case Waitall_from_ISend:
                    if (matching_Wait(src, &(src_ev->mpi_ev), destination,
↪  &(dst_ev->mpi_ev)) == OPERATION_COMPLETE) {
                        //associate the destination event to the source event;
                        VEF_event* sender = dst_ev, *wait_ev = src_ev;
                        //The wait event will be discarder. Simply put its state to
↪  complete and analyze the sender
                        wait_ev->match_state = P2P_COMPLETE;
                        info[src].match_events++;
                        //also, put the Isend request to zero for not checking more
↪  times for the Wait

                        sender->mpi_ev.request = 0;
                        //check the sender state. if the receiver part is complete,
↪  then finisfh
                        if (sender->match_state == P2P_WAIT_FOUND) {
                            sender->match_state = P2P_COMPLETE;
                            info[src].match_events++;
                        }
                        return OPERATION_COMPLETE;
                    }
                    break;
                default:
                    return INVALID_MPI_EVENT;
            }
        }
    }
    return OPERATION_COMPLETE;
}
```

Listado 9: Tratamiento Matching para los eventos Wait asociados a un envío no bloqueante

La Figura 5.6 muestra el diagrama que refleja la casuística para los eventos Wait asociados a envíos no bloqueantes y la Figura 5.7 muestra el diagrama que refleja la casuística para los eventos Wait asociados a recepciones no bloqueantes

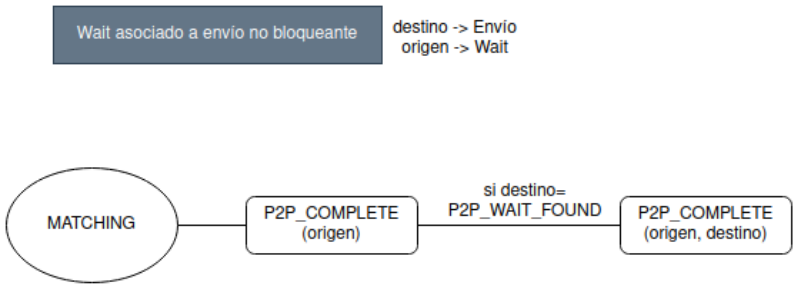

Figura 5.6: Diagrama del Matching para los Wait asociados a envíos no bloqueantes

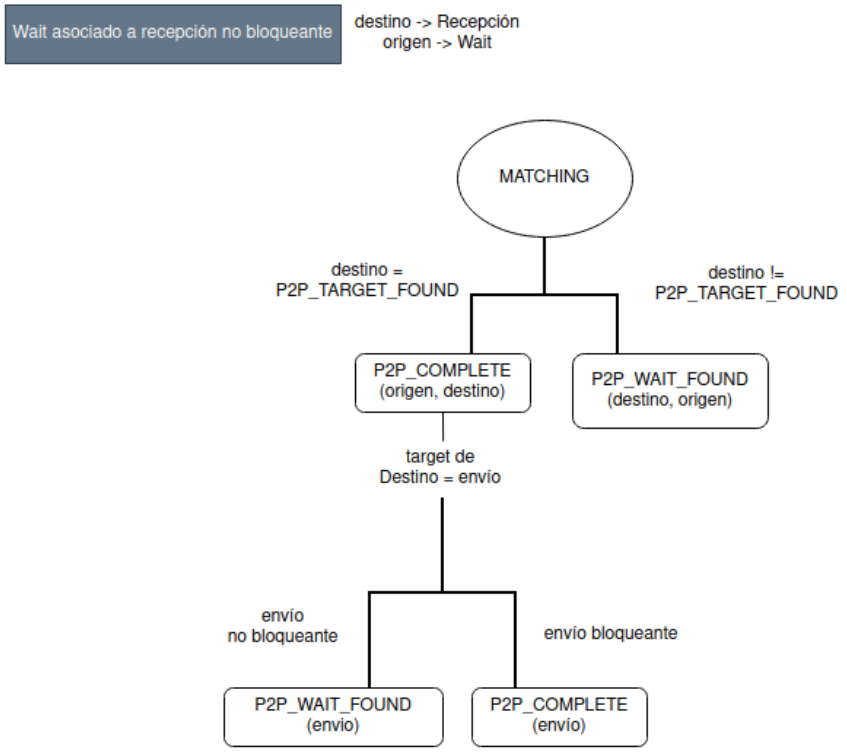

Figura 5.7: Diagrama del Matching para los Wait asociados a recepciones no bloqueantes

La asociación de los eventos de comunicaciones colectivas es más simple que en P2P. En el Listado 10 se muestra la gestión del Matching para este tipo de eventos, a través de la función `matching_GRP_events`.

En primer lugar, se inicia el iterador `iter` para recorrer la lista de comunicaciones colectivas globales, `grplist`. Se utiliza este iterador para iterar en el bucle while hasta que no haya más comunicaciones globales en la lista y por tanto, esta comunicación todavía no exista. La función `matching_grpComm` busca el identificador del comunicador global del evento a través del identificador del comunicador local del evento y comprueba que sea el mismo comunicador, la misma operación colectiva y que la tarea actual no está aún asociada.

Si encuentra la comunicación global `grp`, entonces se le asocia la comunicación local del evento y se actualizan las estadísticas. Se hace una comprobación extra mediante la función `GRPcomm_readFinished`, la cual, en función del tamaño del comunicador y del número de tareas asociadas, determina si la asociación de la comunicación global se ha completado, y en ese caso, la elimina de la lista de comunicaciones globales al no ser ya necesaria.

Al acabar el bucle while, si se ha encontrado el comunicador, significa que ha conseguido asociarse. Si no lo ha encontrado, debe crear uno nuevo. Esto se hace en la segunda parte de la función `matching_GRP_events`, creando las estructuras de datos necesarias para la

gestión de la comunicación colectiva global. También se genera un nuevo identificador para la comunicación global, que será utilizado posteriormente para identificar los eventos locales pertenecientes a la misma operación colectiva en la traza VEF. Por último, se inserta en la lista de eventos de comunicaciones colectivas globales y se actualizan las estadísticas.

```
1
2  prospector_out matching_GRP_events(int32_t source, VEF_event* source_ev) {
3      //uses a list iterator to search the target event in the list
4      list_iterator iter;
5      init_list_iterator(&iter, &grplist);
6      MPI_event* m_ev = &(source_ev->mpi_ev);
7      while (liter_HasNext(&iter) != EMPTY_DATA_STRUCT && source_ev->grp == NULL) {
8          VEF_grpComm* grp = next(&iter);
9          if (matching_grpComm(source, m_ev, grp) == OPERATION_COMPLETE) {
10             //grp found
11             source_ev->grp = grp;
12             set_bit(grp->match_tasks, source);
13             grp->num_match_tasks++;
14             grp->acumm_Bytes += m_ev->lenght;
15             //have been read all the events associated to gr ?
16             if (GRPcomm_readFinished(grp) == OPERATION_COMPLETE) {
17                 //grp is no longer need in the list
18                 removeData(&grplist, iter.current_index);
19             }
20         }
21     }
22     if (source_ev->grp == NULL) {
23         VEF_grpComm* grp = malloc(sizeof (VEF_grpComm));
24         if (grp == NULL)
25             return ALLOCATION_FAILED;
26         NEW_Ulimit_BitMask(grp->match_tasks, vef_info.numTask);
27         if (grp->match_tasks == NULL)
28             return ALLOCATION_FAILED;
29         //assign a new id
30         grp->global_id = grp_id_generator++;
31         vef_info.numGlobalgrp++;
32         prospector_out out;
33         if ((out = global_comm_id(m_ev->comm, source, &(grp->comm_id))) !=
   ↪  OPERATION_COMPLETE)
34             fnError("Error getting the global Comm id. %s", getProspectorOut(out));
35         grp->operation = m_ev->type;
36         clear_all_bits(grp->match_tasks, vef_info.numTask);
37         set_bit(grp->match_tasks, source);
38         grp->num_match_tasks = 1;
39         grp->num_dump_tasks = 0;
40         grp->acumm_Bytes = m_ev->lenght;
41         source_ev->grp = grp;
42         insert(&grplist, grp, 0);
43     }
44     source_ev->match_state = GRP_COMPLETE;
45     vef_info.numLocalgpr++;
46     return OPERATION_COMPLETE;
47 }
```

Listado 10: Matching GRP

5.2. Modificaciones realizadas

En esta sección se explican las modificaciones que se han realizado en los algoritmos mencionados para el desarrollo de una versión paralelizada de la aplicación.

El bucle que se debe paralelizar directamente es el del programa principal `vef_mixer.c`. Este bucle es el que realiza las llamadas a la lectura y asociación de eventos. Se deben analizar bien todas las llamadas y variables globales compartidas del programa en su conjunto para saber con precisión qué secciones deben ser críticas al desplegar varios hilos que trabajen simultáneamente. Se van a desplegar hilos de tal manera que cada hilo procese la lectura y matching completo de una tarea, desde el inicio hasta el final, sin llegar a realizar el volcado.

Observando el comportamiento del programa, se llega a la conclusión de que se debe desacoplar la lectura de los eventos y la asociación o Matching de los mismos, ya que con el diseño original se ve afectado el funcionamiento correcto de la versión paralela. El Matching de los eventos utiliza iteradores cuyos punteros iteran sobre listas utilizadas globalmente. Si hay hilos modificando una lista global, y otros consultándola a través de iteradores, los iteradores no funcionan correctamente. Por esta razón, se ha realizado este desacoplamiento, ya que las listas globales solo se modifican en la parte de lectura, y solo se recorren con iteradores en la función de Matching.

```
1  prospector_out match_event(int32_t task) {
2      //uses a list iterator to search the target event in the list
3      VEF_event_iterator iteraMatch;
4      init_event_queue_iterator(&iteraMatch, VEF_events + task);
5      prospector_out out;
6      VEF_event* source_ev;
7
8      while (event_queue_has_next(&iteraMatch) != EMPTY_DATA_STRUCT) {
9          source_ev = event_queue_next(&iteraMatch);
10             t_mpi_scope scope = GET_EVENT_SCOPE(source_ev->mpi_ev.type);
11         if (scope == P2P && source_ev->match_state != P2P_COMPLETE) {
12             out = matching_P2P_events(task, source_ev);
13             if ( out != OPERATION_COMPLETE)
14                 return out;
15         } else if (scope == GRP_COMM && source_ev->match_state != GRP_COMPLETE) {
16             out=matching_GRP_events(task, source_ev);
17             if (out != OPERATION_COMPLETE)
18                 return out;
19         }
20     }
21     return OPERATION_COMPLETE;
22 }
```

Listado 11: Nueva función match_event

Para llevar a cabo este cambio, se ha creado una nueva función denominada `match_event` en el que se incluye el fragmento de código que solo afecta al Matching. La otra parte del código permanece en la función `read_event` ya existente. Básicamente las llamadas a las

funciones `matching_P2P_events` y `matching_GRP_events` ya no se realizan en la lectura. En la nueva función, se crea un nuevo iterador llamado `iteraMatch` que va a ir recorriendo la lista de eventos de cada tarea haciendo el matching de cada evento si éste no está completado. Ahora se realiza la lectura de todos los eventos de una tarea y después se procede a su Matching. Este cambio se refleja en el Listado 11.

Modificaciones en el Matching de comunicaciones punto a punto

Para el algoritmo de P2P, se han realizado las siguientes modificaciones. En primer lugar, se han añadido las cláusulas `#pragma` necesarias que establecen una zona crítica para la modificación de las variables que son globales, principalmente cuando se actualizan las estadísticas de la estructura `info` de la tarea afectada. Si no establecemos esta zona como crítica, podrían acceder dos o más hilos al mismo tiempo y alterar los datos reales, dando lugar a problemas de concurrencia o condiciones de carrera.

Al disponer de múltiples hilos no va a ser necesario que todos los eventos intenten asociarse. Para hacer el matching, se obtiene el objetivo del evento origen. Si el destino objetivo es menor que la tarea actual, entonces no se hace matching, ya que debe estar realizado por una tarea anterior. Es decir, solo se intentan asociar los eventos cuyo destino sea el mismo o superior al número de tarea al que pertenece dicho evento. Para ayudar a comprender este concepto, se pone el siguiente ejemplo. Si la tarea 3 envía 10 eventos a la tarea 8, entonces se intenta asociar cuando sea el turno de la tarea 3 de asociarse y deja ambos extremos de la comunicación completados. Cuando llega la tarea 8, que debe asociar 10 eventos de la tarea 3, esta tarea ya se debe haber asociado. Esto evita problemas de concurrencia si la tarea 3 y 8 intentan asociar un par de eventos simultáneamente, ya que el resultado final podría no ser consistente y generar una traza incorrecta. Estos cambios se reflejan en el Listado 12.

```
1  prospector_out matching_P2P_events(int32_t src, VEF_event* src_ev) {
2      int32_t destination;
3      if (IS_SUPPORTED_WAIT(src_ev->mpi_ev.type))
4          destination = src;
5      else
6          destination = src_ev->mpi_ev.target_task;
7      if (destination < src) {
8          return OPERATION_COMPLETE;
9      }
10     VEF_event_iterator iter;
11     init_event_queue_iterator(&iter, VEF_events + destination);
12     VEF_matching_state src_state = src_ev->match_state;
13     while (event_queue_has_next(&iter) != EMPTY_DATA_STRUCT && src_ev->target_ev ==
   ↪   NULL) {
14         VEF_event* dst_ev = event_queue_next(&iter);
15         VEF_matching_state dst_state = dst_ev->match_state;
16         t_mpi_event ttype = (t_mpi_event) dst_ev->mpi_ev.type;
17         if (GET_EVENT_SCOPE(ttype) == P2P && src_state == INIT
18         && dst_state < P2P_COMPLETE) {
```

Listado 12: Modificación Matching P2P

Se añade una nueva comprobación a la hora de realizar el matching. El estado que indica la asociación actual, `match_state`, debe ser `INIT` en el evento origen. Esto es debido a un cambio en el modelo de estados de asociación de los eventos. El nuevo algoritmo elimina el estado `P2P_WAIT_FOUND`. El anterior modelo de estados no funcionaba correctamente en la versión paralela. Esto es debido a que es posible que tanto el target como el Wait de una comunicación no bloqueante ocurran simultáneamente. Si la asociación con el target ocurre ligeramente después que la asociación con el Wait, el evento quedaba marcado incorrectamente como `P2P_TARGET_FOUND`. Al estar el Wait marcado como completado, el evento no bloqueante nunca se vuelve a asociar con su Wait, haciendo que la generación de la traza no se complete nunca. Este cambio evita los problemas de concurrencia que se generaban con el anterior modelo de estados.

Acompañando este cambio, se elimina también la funcionalidad vinculada a ese estado presente en la mayoría de funciones, que ya no resulta útil. El nuevo modelo de estados para las comunicaciones punto a punto sería el siguiente:

- INIT
- P2P_TARGET_FOUND
- P2P_COMPLETE.

Se hace el matching para los eventos de envío y recepción, ya sean bloqueantes o no bloqueantes, cuando el estado es el primero, `INIT`, y no se ha hecho nada aún. En los eventos de espera o Wait se hace el matching siempre y cuando haya encontrado primero el objetivo, y por tanto, su estado se encuentre a `P2P_TARGET_FOUND`. Esto significaría que para completar un evento, se necesita realizar primero el Matching con su target y posteriormente el matching con Wait asociado en las comunicaciones no bloqueantes, que son los que pueden ocasionar problemas de concurrencia.

Además, para los eventos Wait asociados a un envío no bloqueante se establece una condición extra para completar la asociación del evento destino. Se comprueba que el objetivo del evento destino no sea un evento IRecv (Listado 21). Esto permite su funcionamiento en paralelo evitando que se completen los eventos de recepciones no bloqueantes antes de tiempo. Con este nuevo algoritmo evitamos que sucedan problemas de concurrencia y se asocien eventos Wait con envíos o recepciones no bloqueantes que no deberían. El resto del algoritmo de Matching se mantiene como el original. A continuación, se muestran las modificaciones de la siguiente forma:

- Envío bloqueante. Listado 13.
- Envío no bloqueante. Listados 14 y 15.
- Recepción bloqueante: Listado 16 y 17.
- Recepción no bloqueante: Listado 18 y 19.
- Wait asociado a recepción no bloqueante: Listado 20.

- Wait asociado a envío no bloqueante: Listado 21.

```
case Send:
case BSend:
case RSend:
case SSend:
case SendRecv_S:
    if (dst_state ==INIT) {
        in=true;
        if (matching_Send(src, &(src_ev->mpi_ev), destination, &(dst_ev->mpi_ev)) ==
↪  OPERATION_COMPLETE) {
            //asociate the  source event to the destinantion event (regardless of is a
↪  Recv or a Irecv);
            P2P_Found=true;
            VEF_event* sender = src_ev, receiver = dst_ev;
            receiver->target_ev = sender;
            if (IS_BLOCKING_RECEIVE(ttype)) {
                sender->is_trigger = 1;
                sender->target_ev = receiver;

                #pragma omp atomic update
                info[destination].match_events++;
                #pragma omp atomic update
                info[src].match_events++;
                sender->match_state = receiver->match_state = P2P_COMPLETE;
            }
            else { //The events is a IRecv
                sender->target_ev = receiver;
                //has the IRecv  an associated wait?
                if (receiver->mpi_ev.request == 0) {
                    sender->is_trigger = 0;
                    sender->match_state = receiver->match_state = P2P_COMPLETE;

                    #pragma omp atomic update
                    info[destination].match_events++;

                    #pragma omp atomic update
                    info[src].match_events++;
                }else {//yes, and the Irecv hasn't found  its assocciated wait;
                    sender->match_state = receiver->match_state = P2P_TARGET_FOUND;
                    sender->is_trigger = 1;
                }
            }
            return OPERATION_COMPLETE;
    else
        in=false;
    break;

```

Listado 13: Modificaciones Envío bloqueante P2P

Además de estos cambios, se añaden 2 variables booleanas, `P2PFound` e `in` que se utilizan en el matching de la siguiente forma. La variable `P2PFound` indica si ha llegado encontrar evento con el que asociarse a través de las distintas funciones matching (`matching_Send, matching_Wait, matching_Recv`). Si al haber recorrido todas las iteraciones para intentar asociarse de un evento, no se ha asociado, se detiene el matching para esa tarea. Esto se complementa con la variable `in`, que se utiliza para distinguir las funciones que son envíos y recepciones de los Wait, con el objetivo de solo detener los envíos y recepciones, no los Wait. Esto permite dar tiempo a que se encuentren los eventos a través de otras tareas, y cuando vuelva a intentar asociarse con la misma tarea tardará menos. No se incluyen los eventos Wait ya que no es necesario detenerlos, pues no se van a asociar si no se han encontrado primero los eventos de envío o recepción. En el Listado 22 se muestra la condición que se usa para abortar la ejecución del Matching. Se añade además la condición de que el evento origen es `INIT`, es decir, no se ha hecho nada aún, y no ha podido asociarse. Los cambios en el algoritmo de asociación de eventos P2P puede suponer un aumento en el tiempo de ejecución al ejecutarse en secuencial. No obstante, con el despliegue de hilos trabajando de forma paralela, esto debería compensar este aumento y mejorar el rendimiento.

```
1                 case ISend:
2                 case IBSend:
3                 case IRSend:
4                 case ISSend:
5                     if (dst_state ==INIT) {
6                         in=true;
7                         if (matching_Send(src, &(src_ev->mpi_ev), destination,
  ↪  &(dst_ev->mpi_ev)) == OPERATION_COMPLETE) {
8                             P2P_Found=true;
9                             VEF_event* sender = src_ev;
10                            VEF_event* receiver = dst_ev;
11                            receiver->target_ev = sender;
12
13                            if (IS_BLOCKING_RECEIVE(ttype)) {
14                                sender->is_trigger = 1;
15                                //asociate the destination event to the source event;
16                                sender->target_ev = receiver;
17                                //the target event is complete
18                                receiver->match_state = P2P_COMPLETE;
19                                //the source event is complete it has no request
  ↪  associated
20                                sender->match_state = (sender->mpi_ev.request == 0) ?
  ↪  P2P_COMPLETE : P2P_TARGET_FOUND;
21
22                                #pragma omp atomic update
23                                info[destination].match_events++;
24                            }
25
```

Listado 14: Modificaciones Envío no bloqueante P2P. Parte 1

```
                            } else { //The events is a IRecv
                                    sender->target_ev = receiver;
                                    if (receiver->mpi_ev.request == 0) {
                                        sender->is_trigger = 0;
                                        receiver->match_state = P2P_COMPLETE;

                                        #pragma omp atomic update
                                        info[destination].match_events++;
                                        sender->match_state = (sender->mpi_ev.request
    ↪  == 0) ? P2P_COMPLETE : P2P_TARGET_FOUND;

                                    } else {//yes, and the Irecv hasn't found  its
    ↪  assocciated wait;
                                        sender->match_state = receiver->match_state =
    ↪  P2P_TARGET_FOUND;
                                        sender->is_trigger = 1;
                                    }
                            }
                            if (sender->match_state == P2P_COMPLETE)

                                #pragma omp atomic update
                                info[src].match_events++;
                            return OPERATION_COMPLETE;
                        }
                    }
                    else
                        in=false;
                    break;


```

Listado 15: Modificaciones Envío no bloqueante P2P. Parte 2

```
                case Recv:
                case SendRecv_R:
                    if (dst_state == INIT) {
                        in=true;
                        if (matching_Recv(src, &(src_ev->mpi_ev), destination,
    ↪  &(dst_ev->mpi_ev)) == OPERATION_COMPLETE) {
                            P2P_Found=true;
                            VEF_event* sender = dst_ev, receiver = src_ev;
                            receiver->target_ev = sender;
                            receiver->match_state = P2P_COMPLETE;
                            #pragma omp atomic update
                            info[src].match_events++;
                            //asociate the  source event to the destinantion event;
                            sender->target_ev = receiver;
                            sender->is_trigger = 1;
```

Listado 16: Modificaciones Recepción bloqueante P2P. Parte 1

```
                            if (IS_BLOCKING_SEND(sender->mpi_ev.type)
                                    || sender->mpi_ev.request == 0) {
                                sender->match_state = P2P_COMPLETE;
                                #pragma omp atomic update
                                info[destination].match_events++;
                                sender->match_state = P2P_COMPLETE;
                            } else {
                                sender->match_state = P2P_TARGET_FOUND;
                            }
                            return OPERATION_COMPLETE;
                        }
                    }
                    else
                        in=false;
                    break;
```

Listado 17: Modificaciones Recepción bloqueante P2P. Parte 2

```
case IRecv:
    if (dst_state == INIT) {
        in=true;
        if (src_ev->mpi_ev.target_task == MPI_ANY_SOURCE)
            fnError("Something was wrong: A Irecv has with MPI_ANY_SOURCE sender has
↪  not obtained the real sender during the preprocessing phase\n");
        if (matching_Recv(src, &(src_ev->mpi_ev), destination, &(dst_ev->mpi_ev)) ==
↪  OPERATION_COMPLETE) {
            P2P_Found=true;
            VEF_event* sender = dst_ev, receiver = src_ev;
            receiver->target_ev = sender;
            sender->target_ev = receiver;

            if (receiver->mpi_ev.request == 0) {
                receiver->match_state = P2P_COMPLETE;
                sender->is_trigger = 0;
                if (IS_BLOCKING_SEND(sender->mpi_ev.type)
                || sender->mpi_ev.request == 0) {
                        sender->match_state = P2P_COMPLETE;

                        #pragma omp atomic update
                        info[destination].match_events++;


```

Listado 18: Modificaciones Recepción no bloqueante P2P. Parte 1

```
        } else
            sender->match_state = P2P_TARGET_FOUND;
            } else {
                sender->is_trigger = 1;
                sender->match_state = receiver->match_state = P2P_TARGET_FOUND;
            }
            if (receiver->match_state == P2P_COMPLETE)
                #pragma omp atomic update
                info[src].match_events++;
                return OPERATION_COMPLETE;
        }
    }
    else
        in=false;
    break;
```

Listado 19: Modificaciones Recepción no bloqueante P2P. Parte 2

```
case Wait_from_IRecv:
case Waitall_from_IRecv:
in=false;
    if (dst_ev->match_state==P2P_TARGET_FOUND && matching_Wait(src, &(src_ev->mpi_ev),
    ↪  destination, &(dst_ev->mpi_ev)) == OPERATION_COMPLETE) {
        P2P_Found=true;
        VEF_event* receiver = dst_ev, *wait_ev = src_ev;
        receiver->match_state = wait_ev->match_state = P2P_COMPLETE;
        #pragma omp atomic update
        info[src].match_events += 2;
        VEF_event* sender = receiver->target_ev;
        sender->target_ev = wait_ev; //associate the wait to the send
        wait_ev->target_ev = sender; //associate the send to the wait
        if (IS_BLOCKING_SEND(sender->mpi_ev.type)
            || sender->mpi_ev.request == 0) {
            sender->match_state = P2P_COMPLETE;
            #pragma omp atomic update
            info[receiver->mpi_ev.target_task].match_events++;
        } else
            sender->match_state = P2P_TARGET_FOUND;
    return OPERATION_COMPLETE;
    }
    break;

```

Listado 20: Modificaciones Wait asociado a Recepción no bloqueante P2P.

```
case Wait_from_ISend:
case Waitall_from_ISend:
in=false;
    if (dst_ev->match_state==P2P_TARGET_FOUND && matching_Wait(src, &(src_ev->mpi_ev),
    destination, &(dst_ev->mpi_ev)) == OPERATION_COMPLETE) {
        P2P_Found=true;
        VEF_event* sender = dst_ev, *wait_ev = src_ev;
        wait_ev->match_state = P2P_COMPLETE;
        #pragma omp atomic update
        info[src].match_events++;
        //also, put the Isend request to zero for not checking more times for the Wait
        sender->mpi_ev.request = 0;

        if (sender->match_state == P2P_TARGET_FOUND && sender->target_ev->mpi_ev.type
    != IRecv) {
            sender->match_state = P2P_COMPLETE;
            #pragma omp atomic update
            info[src].match_events++;
        }
        return OPERATION_COMPLETE;
    }
    break


```

Listado 21: Modificaciones Wait asociado a Envío no bloqueante P2P.

```
     if(src_ev->match_state == INIT && !P2P_Found && in)
         return INVALID_TASK;

    return OPERATION_COMPLETE;
}

```

Listado 22: Tratamiento para la detención de la asociación de eventos

Modificaciones en el Matching de comunicaciones colectivas

En la parte de asociación de eventos de colectivas (Listado 10), se observa una dependencia que impide su funcionamiento en paralelo, y es la lista global `grplist`. En el diseño original, se insertan y se eliminan eventos en la misma función. Esto no supondría ningún problema en la versión secuencial, pero al recorrer la lista usando un iterador, esto supone problemas de concurrencia cuando hay tareas modificando la lista y otras consultándola, tal y como explicamos al inicio de la sección. La solución que se ha adoptado para este problema ha sido doble. Por una parte, se ha desacoplado la parte de creación de nuevas comunicaciones globales e incluido en la parte de lectura de eventos. En el Listado 23 se puede ver la modificación realizada sobre la función `read_event`.

```
1  if (scope == GRP_COMM) {
2          uint16_t gid;
3          if (global_comm_id(source_ev->mpi_ev.comm, task, &gid) == OPERATION_COMPLETE) {
4              uint32_t size;
5              getCommSize(gid, &size);
6              if (size == 1) {
7                  //this avoid collective operations of one task
8                  scope = NOT_SUPPORTED;
9              }
10
11             prospector_out out;
12             MPI_event* m_ev = &(source_ev->mpi_ev);
13
14             if ((out = is_first_task(gid, task)) == OPERATION_COMPLETE){
15                 //NOT grp found. Create a new comm
16                 VEF_grpComm* grp = malloc(sizeof (VEF_grpComm));
17                 if (grp == NULL)
18                     return ALLOCATION_FAILED;
19                 //allocate the matching bitmask
20                 NEW_Ulimit_BitMask(grp->match_tasks, vef_info.numTask);
21                 if (grp->match_tasks == NULL)
22                     return ALLOCATION_FAILED;
23
24                 #pragma omp critical
25                 grp->global_id = grp_id_generator++;
26                 #pragma omp atomic update
27                 vef_info.numGlobalgrp++;
28                 if ((out = global_comm_id(m_ev->comm, task, &(grp->comm_id))) !=
   ↪  OPERATION_COMPLETE)
29                     fnError("Error getting the global Comm id. %s",
   ↪  getProspectorOut(out));
30                 grp->operation = m_ev->type;
31                 //clear the matching bitmask
32                 clear_all_bits(grp->match_tasks, vef_info.numTask);
33                 set_bit(grp->match_tasks, task);
34                 grp->num_match_tasks = 1;
35                 grp->num_dump_tasks = 0;
36                 //set the message size
37                 grp->acumm_Bytes = m_ev->lenght;
38
39                 #pragma omp critical
40                 insert(&grplist, grp, 0);
41                 source_ev->match_state = GRP_COMPLETE;
42                 source_ev->grp = grp;
43                 #pragma omp atomic update
44                 vef_info.numLocalgpr++;
45             }
46
47         }
48     }
49
```

Listado 23: Modificación del algoritmo de lectura de eventos

En la función modificada, se establecen cláusulas para establecer secciones críticas para la generación de identificador del comunicador, representada por la variable `grp_id_generator`, para la inserción del nuevo comunicador en la lista `grplist` y para la actualización del número total de colectivas globales y locales.

Para saber qué tarea es la primera del comunicador y poder realizar el desacoplamiento correctamente, se ha creado una función denominada `is_first_task` (Listado 24). Su objetivo es comprobar si una determinada tarea es la primera del comunicador utilizado por la comunicación colectiva o no. Esto se consigue mediante un bucle que itera sobre el número total de tareas mientras va comprobando si el bit de la tarea y la máscara de bits de tareas del comunicador coinciden. Esto implica que la tarea pertenece al comunicador. La tarea que pase esta comprobación será la primera tarea del comunicador y la encargada de crear la comunicación colectiva global.

Esta función se utiliza al principio de la función `matching_GRP_events`. Si la tarea no es la primera, entonces se procede a realizar el matching. En caso contrario, no se hace nada, puesto que ya se ha creado anteriormente en la lectura de eventos. También se utiliza en la creación del nuevo comunicador en la función `read_event`. Se creará un nuevo comunicador siempre y cuando la tarea actual sea la primera del comunicador, siguiendo el funcionamiento original.

```
prospector_out is_first_task(uint16_t global_comm, uint32_t task){
    void * pointer;
    if((hash_search(&ctable, int2voidp global_comm, &pointer) == OPERATION_COMPLETE)){
            vef_comm* comm = (vef_comm*) pointer;
            uint32_t first_task;
        for (int i = 0; i < vef_info.numTask; i++) {
            if (get_bit(comm->tasks, i)==1){
                first_task=i;
                break;
            }
        }
        if(first_task==task)
            return OPERATION_COMPLETE;
        else
            return INVALID_TASK;
    }
    return INVALID_GLOBAL_COMM_ID;
}

```

Listado 24: Función is_first_task

Por otra parte, se debe solucionar la parte de borrado de la lista para evitar que los iteradores en uso por otros hilos dejen de funcionar. Para esta situación, la solución que se ha adoptado ha sido desacoplar el borrado de la función `matching_GRP_events` y realizar la eliminación después del Matching. Para llevar a cabo esta solución, se han seguido los siguientes pasos.

En primer lugar, se necesita saber qué elementos de la lista se deben eliminar. Para esto, se va a introducir una funcionalidad de marcado para borrar posteriormente los elementos. Se establece un nuevo campo `statusNode` en la estructura de la lista de nodos. Este campo consiste en un enumerador (`enum`) con las variables necesarias para el borrado, que son `MARKED_TO_REMOVE` y `NOT_REMOVE`. Además, este nuevo campo permite añadir funcionalidad extra en el futuro, simplemente agregando nuevas variables al enumerador (Listado 25).

```
1    enum status {
2         NOT_REMOVE=0,
3         MARKED_TO_REMOVE=1,
4     };
5
6     struct listNode {
7         uint64_t m_key;
8         const void* data_ptr;
9         struct listNode *next;
10        enum status statusNode;
11    };
```

Listado 25: Nuevo atributo en lista de nodos y enum status

Se ha creado una nueva función denominada `markToRemove` en la que se marca el elemento para borrar si así se requiere cambiando su variable `statusNode`, recorriendo la lista dinámica hasta el punto en que se localiza el nodo. Se necesita entonces otra función para el borrado del evento. Esta función será `removeMarked`, la cual recorre la lista dinámica y elimina aquellos nodos que están marcados para eliminar. Esta función es llamada en el programa principal `vef_mixer.c`, tras realizar el matching de los eventos de una tarea en concreto. Dichas funciones se muestran en los Listado 26 y 27.

```
1  prospector_out markToRemove(list *L,uint32_t index) {
2
3      if (index < L->size) {
4          int cont = 0;
5          listNode *last = NULL;
6          listNode *lnode = L->head;
7          while (cont < index) {
8              last = lnode;
9              lnode = lnode->next;
10             cont++;
11         }
12         lnode->statusNode=1;
13         return OPERATION_COMPLETE;
14     }
15     return INVALID_TASK;
16 }
```

Listado 26: Función de marcado para eliminar un evento

```
prospector_out removeMarked(list *L){
    //remove from grplist
    //go through the grplist and delete the nodes marked to delete
    int cont = 0;
    listNode *last = NULL;
    listNode *lnode = L->head;
    while (cont <= L->size - 1 &&lnode!=NULL) {

        if(lnode->statusNode){
            if (last == NULL) /*Head node*/
                L->head = lnode->next;
            else
                last->next = lnode->next;

        /*Is the last node erased???*/
        if (lnode == L->lastNode)
            L->lastNode = last;
        listNode * temp=lnode;
        lnode = lnode->next;
        L->size--;
        free(temp);
        //no increment of cont and last not updated
        }else{
            last = lnode;
            lnode = lnode->next;
            cont++;
        }
    }
    return OPERATION_COMPLETE;
}
```

Listado 27: Función para la eliminación de eventos marcados de una lista

Para hacer el algoritmo más eficaz, se utiliza la variable booleana `grpFound`, que se va a activar cuando el evento se asocie con su comunicación colectiva global. Al terminar el bucle, si no se ha asociado el evento con ninguna comunicación global de la lista, entonces se devuelve `INVALID_GLOBAL_COMM_ID` para parar el matching de esa tarea y retomarlo más adelante, ya que donde se llama a esta función es en el bucle de la función `match_event` que iba recorriendo los eventos de la tarea. Esto permite dar tiempo a que la tarea correspondiente genere la comunicación global, y cuando vuelva a intentar hacer el Matching con la misma tarea, tendrá menos posibilidades de no encontrar la comunicación global con la que asociarse.

En el Listado 28 se muestran las modificaciones realizadas en la función `matching_GRP_events`. Además, el Listado 29 muestra el bucle del programa principal `vef_mixer.c`, en el que se llama a la función de eliminación de los eventos que han sido marcados para ello. También se han establecido las cláusulas de OpenMP para desplegar hilos en los bucles internos `for`.

```
prospector_out matching_GRP_events(int32_t source, VEF_event* source_ev) {
    prospector_out out;
    uint16_t gcomm_id;
    MPI_event* m_ev = &(source_ev->mpi_ev);
    if ((out = global_comm_id(m_ev->comm, source, &gcomm_id)) != OPERATION_COMPLETE)
        fnError("Error getting the global Comm id. %s", getProspectorOut(out));
    if ((out = is_first_task(gcomm_id, source)) != OPERATION_COMPLETE){
        list_iterator iter;
        init_list_iterator(&iter, &grplist);
        bool grpFound=false;

        while (liter_HasNext(&iter) != EMPTY_DATA_STRUCT && source_ev->grp == NULL) {
            VEF_grpComm* grp = next(&iter);

            if (matching_grpComm(source, m_ev, grp) == OPERATION_COMPLETE) {
                grpFound=true;
                source_ev->grp = grp;
                #pragma omp critical
                {
                    set_bit(grp->match_tasks, source);
                    grp->num_match_tasks++;
                    grp->acumm_Bytes += m_ev->lenght;
                }
                if (GRPcomm_readFinished(grp) == OPERATION_COMPLETE) {
                    //grp is no longer need in the list
                    markToRemove(&grplist, iter.current_index);
                }
                source_ev->match_state = GRP_COMPLETE;
                #pragma omp critical
                vef_info.numLocalgpr++;
            }
        }
        if(!grpFound){
            return INVALID_GLOBAL_COMM_ID;
        }
    }
}
```

Listado 28: Modificación Matching GRP

```
    while (are_events_to_dump(ANY_SOURCE) != END_OF_VEFT_FILE) {
        //first read
        TIME init_match = getRealTime();
        #pragma omp parallel for
        for (int task = 0; task < vef_info.numTask; task++) {
            int32_t chunk = 1000;
            int32_t reads = 0;
            int8_t flag;
            while (are_events_to_read(task) == OPERATION_COMPLETE && reads < chunk) {
                if ((out = read_event(task, &flag)) != OPERATION_COMPLETE) {
                    fnError("Error reading an Event. %s", getProspectorOut(out));
                }

                reads += flag;
            }

        }
        #pragma omp parallel for
        for (int task = 0; task < vef_info.numTask; task++) {
            match_event(task);

        }

        removeMarked(&grplist);
        TIME end_match = getRealTime();
        matching_time += (end_match - init_match);
        //dump the trace
        dump_vef_trace();
        dumping_time += (getRealTime() - end_match);
        print_progress();
    }

```

Listado 29: Bucle del programa principal `vef_mixer.c`

Capítulo 6

Evaluación de rendimiento

6.1. Pruebas de rendimiento

En este capítulo se realizan las pruebas de rendimiento de la aplicación *VEF Mixer* paralelizada. El objetivo de estas pruebas es evaluar el rendimiento de la versión paralela y realizar una comparación con el modelo secuencial mostrando los resultados obtenidos.

Para la realización de este tipo de pruebas, un ordenador personal no resulta ser demasiado útil, puesto que tiene la limitación de no tener suficientes cores o núcleos de procesador. Este inconveniente no permite probar la escalabilidad del programa con el paralelismo. Se necesita una cantidad de procesos concurrentes elevada, por lo que cuantos más núcleos se tengan más resultados potenciales se pueden llegar a obtener.

Por ello, se ha utilizado el clúster del grupo de investigación Trasgo [33] de la Universidad de Valladolid. El clúster dispone de varias máquinas, pero en este proyecto se ha utilizado principalmente la máquina *gorgon*, compuestas por 2 CPUs AMD EPYC 7713 (Ryzen 3) CPU @ 2.0GHz, con 64 cores y 128 hilos cada uno, utilizando Simultaneous Multi-Threading. La tecnología Simultaneous Multi-Threading (SMT) [34] es una técnica que combina el multihilo por hardware con la tecnología de procesador superescalar para permitir que varios hilos emitan instrucciones en cada ciclo. A diferencia de otras arquitecturas multihilo por hardware (como el Tera MTA), en las que sólo un contexto de hardware (es decir, un hilo) está activo en un ciclo determinado, SMT permite que todos los contextos de hilo compitan simultáneamente por los recursos del procesador y los compartan.

Cuando la tecnología SMT está activa, la CPU expone dos contextos de ejecución por núcleo físico. Esto significa que un núcleo físico ahora funciona como dos "núcleos lógicos" que pueden manejar diferentes subprocesos de software. Dos núcleos lógicos pueden realizar las tareas de forma más eficiente que un núcleo tradicional de un solo hilo, ya que un hilo puede utilizar las unidades funcionales físicas de la CPU que no son utilizadas por el otro hilo. El SMT mejora el rendimiento de la CPU al aprovechar el tiempo de inactividad en el que el núcleo estaría esperando a que se completaran otras tareas.

Cabe destacar que el total de procesos ejecutándose concurrentemente no puede ser mayor de los que soporta el procesador, ya que de lo contrario se pierde eficiencia y puede llegar a generar resultados erráticos.

Para las pruebas se utiliza MPICH [35] en su versión 3.4.2 para compilar la biblioteca *VEF-Prospector*. Además, se utiliza el compilador gcc 11.1 [41]. Se han utilizado 3 aplicaciones distintas para evaluar la aplicación paralela. Se trata de las aplicaciones Gromacs [36], Alexnet [37] y HPL [38].

- Gromacs es una aplicación de dinámica de moléculas de alto rendimiento. La traza de esta aplicación ha sido generada con 240 hilos MPI. Contiene múltiples operaciones colectivas y punto a punto.
- Alexnet es una aplicación de Machine Learning para clasificar imágenes. Esta traza ha sido generada con 98 hilos MPI. Solo contiene operaciones colectivas.
- HPL es una aplicación de alto rendimiento computacional (High Performance Computing) del benchmark LINPACK que resuelve un sistema lineal denso (aleatorio) en aritmética de doble precisión (64 bits) en ordenadores de memoria distribuida. Esta traza ha sido generado con 32 hilos MPI y solo contiene operaciones punto a punto.

Cada una de las aplicaciones se ha ejecutado un total de 30 veces para cada número de hilos establecido, para así poder obtener una media representativa del tiempo de ejecución de cada prueba. Cada prueba se realiza de forma aislada de las demás, y se espera a que termine una para lanzar la siguiente. Se recoge el tiempo de ejecución total, y el tiempo de ejecución de las partes:

- Pre-procesado: se trata del preprocesado de los ficheros temporales para almacenar la información de las operaciones tipo Wait. Abarca el inicio, antes de llegar a la lectura de eventos.
- Matching. Abarca lectura y asociación de eventos.
- Dumping. Abarca el volcado de los eventos.

El tiempo de ejecución nos va a permitir obtener la mejora de rendimiento de la paralelización con respecto al programa original.

La cantidad de mejora global de rendimiento de un sistema, formalmente se le llama aceleración o *speed-up* [39] del sistema. Permite saber cuánto más rápido se ejecutará la nueva versión mejorada. Se calcula comparando los tiempos de ejecución antes y después de aplicar una mejora:

$$SpeedUp = \frac{T_1}{T_n}$$

donde T_1 es el tiempo de ejecución del programa original y T_n es el tiempo de ejecución de la paralelización.

Como se recogen un gran número de pruebas, conviene automatizar el lanzamiento a través de un script. Se crea un script para aplicación probada, es decir un total de 3. Estos se encargan de lanzar la prueba con los parámetros correspondientes, guardar los resultados en un fichero con la salida del programa, y borrar los ficheros traza del disco, ya que no se podrían almacenar todos, puesto que son muy pesados. Se crea un fichero para cada grupo de hilos de cada aplicación probada.

A continuación se describen las pruebas realizadas para este proyecto.

6.2. Evaluación

6.2.1. Alexnet

Para la prueba de Alexnet se han tenido en cuenta los siguientes grupos de hilos: 0, 1, 2, 4, 8, 16 y 32. El grupo de hilos 0 indica que la prueba es secuencial, utilizando el programa original. Cabe recordar que esta prueba solo contenía comunicaciones colectivas.

La Tabla 6.1 muestra la tabla con los tiempos de ejecución para esta prueba. La Figura 6.1 muestra los resultados de la prueba. La aplicación original, es decir, la prueba con 0 hilos, ha obtenido una media del tiempo de ejecución total de 72.1782 segundos. La prueba con 1 hilo ha disminuido mínimamente el tiempo total, lo que es coherente, debido a los cambios de diseño que experimenta la parte de procesado de las comunicaciones colectivas. Al desacoplar la creación de nuevos comunicadores de la función de Matching e incorporarla en la parte de lectura de eventos, se reduce la posibilidad de que un evento no encuentre el comunicador en su intento de asociación, permitiendo mejorar ligeramente el tiempo de ejecución.

A partir de ese punto se puede observar que se empieza a disminuir el tiempo de ejecución cuando se despliegan hilos, llegando hasta el punto máximo de rendimiento, que sucede con 8 hilos. El tiempo de ejecución mínimo alcanzado es de 36,7090. El porcentaje de mejora del tiempo de ejecución es del 50,86 %, produciendo un *speed-up* de 1,966. El programa paralelizado es aproximadamente 2 veces más rápido que el programa original.

Se observa también que con 16 y 32 hilos, aunque se siguen obteniendo tiempos más reducidos que con la aplicación original, no sigue escalando la mejora de rendimiento.

Los resultados en esta prueba no han sido los esperados, ya que se preveía que el programa paralelo escalara más antes de dejar de escalar. No obstante, se reduce el tiempo de ejecución un 50 %, obteniendo un speed-up de 2X.

Pre Procesado	Matching	Dumping	Total	Hilos
0,3551	63,1759	8,6472	72,1782	0
0,3533	57,1495	8,5760	66,0788	1
0,3524	41,7278	9,5000	51,5805	2
0,3372	28,8414	10,2352	39,4138	4
0,3434	25,5000	10,8655	36,7090	8
0,3486	39,2883	12,4497	52,0865	16
0,3457	43,1993	13,2863	56,7312	32

Tabla 6.1: Tabla de evaluación de la prueba Alexnet

Figura 6.1: Tiempo de ejecución de la traza Alexnet

6.2.2. Gromacs

Para las pruebas de Gromacs se ha utilizado el nombre Gmx como abreviatura, que es el nombre surgido de la descarga de la muestra. Se han tenido en cuenta los grupos de hilos de 0, 1, 2, 4, 8 y 16 hilos para esta prueba.

En la Tabla 6.2 se muestra la tabla con los tiempos de ejecución de esta prueba y en la Figura 6.2 se detalla gráficamente la comparativa de esta prueba. Se puede observar que el tiempo del Preprocesado no se ha visto afectado por las modificaciones de los algoritmos, lo que resulta lógico porque esa parte no se ha modificado.

El tiempo del Dumping se ve afectado notablemente con el aumento del despliegue de hilos hasta duplicar el valor inicial. Debido a las modificaciones realizadas en la parte de Matching en las operaciones punto a punto para evitar condiciones de carrera que hacían que la traza se generará incorrectamente, es menos probable que en cada iteración del bucle

while de la función main haya eventos para volcar al fichero. Esto aumenta el tiempo de dumping, ya que se ejecuta más veces que en la versión original. Aunque no es mucho tiempo en relación con el total, es un resultado que se debe tener en cuenta. El tiempo del Matching, lógicamente se ha reducido significativamente debido a las modificaciones de los algoritmos y al paralelismo.

La aplicación original produce un tiempo medio de ejecución de 129,302 segundos. El tiempo de ejecución se ha incrementado hasta 191,45 segundos en la siguiente prueba con 1 hilo.

El cambio de algoritmo en la parte de comunicaciones punto a punto es el desencadenante de este resultado, ya que en las pruebas hechas en Alexnet, que son exclusivamente de colectivas, esto no ha ocurrido. Esto se debe principalmente, como se ha comentado en el Capítulo 5, a la detención del Matching en múltiples ocasiones para no seguir con la asociación del evento cuando no encuentra a su target o su Wait asociado en el caso de las comunicaciones no bloqueantes. Al ejecutar la nueva aplicación de forma secuencial, estos cambios hacen que aumente el tiempo de ejecución del Matching.

A partir de 2 hilos, se puede observar cómo va disminuyendo progresivamente el tiempo medio de ejecución hasta llegar a su mínimo, 35,618 segundos con 16 hilos. Esto produce una reducción del tiempo del 72,45 %, obteniendo un speed-up de 3,63X.

Pre Procesado	**Matching**	**Dumping**	**Total**	**Hilos**
2,052	124,419	2,831	129,302	0
2,065	186,531	2,857	191,454	1
2,075	110,624	3,535	116,235	2
2,059	66,770	4,032	72,862	4
2,052	40,173	4,817	47,042	8
2,046	27,660	5,912	35,618	16

Tabla 6.2: Tabla de evaluación de la prueba Gmx

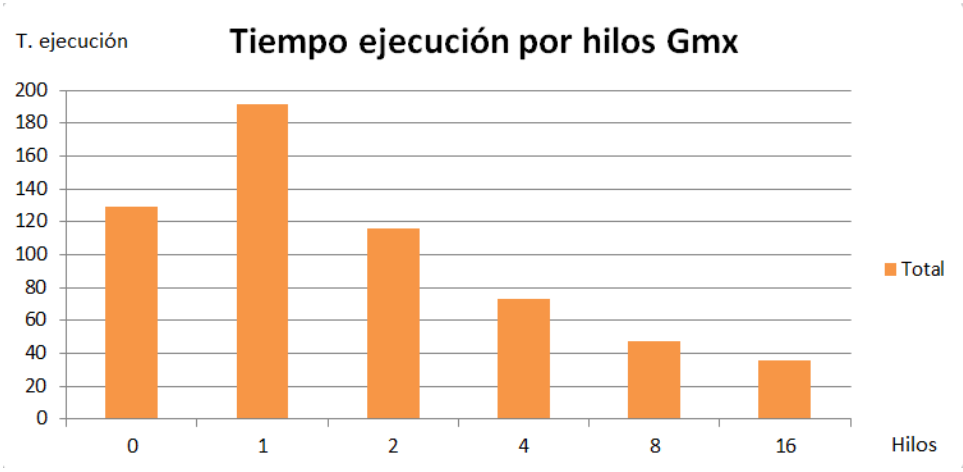

Figura 6.2: Tiempo de ejecución de la traza Gmx

6.3. Prueba HPL

Para las pruebas de la aplicación HPL, se han tenido en cuenta los grupos de hilos: 0, 1, 2, 4, 8, 16, 32 y 64 hilos.

Los resultados de las pruebas se pueden observar en la Tabla 6.3 y en la Figura 6.3, que muestran la tabla de evaluación y la gráfica comparativa, respectivamente. Al ser una prueba tan pequeña, solo 32 tareas, dura muy poco tiempo. Aún así, se puede observar mejora de rendimiento con la paralelización. El tiempo de Dumping, al igual que en las otras pruebas, se ve afectado con el despliegue de hilos, aumentando casi hasta el doble su valor inicial. Al ser tan poco tiempo comparado con el total, es un aumento de tiempo asumible. El tiempo de Matching sí que mejora, obteniendo un speed-up de 4,96X.

El tiempo de ejecución medio del programa original es de 2,5280 segundos. La ejecución de la prueba con 2 hilos no mejora el tiempo de ejecución, sino que aumenta ligeramente hasta los 2,68 segundos. Este resultado se debe a las modificaciones introducidas en en la función de Matching, tal y como ocurría con la aplicación Gromacs en la sección anterior. A pesar de que todas las comunicaciones son punto a punto como en la anterior aplicación, utiliza otras funciones distintas que provocan más detenciones en la función de Matching, que al ejecutase únicamente con 2 hilos, no llega a compensarse con la mejora de la paralelización. Es por eso que en la anterior prueba no se mostraba este efecto. Tampoco pasa con 1 hilo porque es secuencial, y no se realiza matching de varias tareas a la vez, lo que reduce la posibilidad de tener que detener el matching de una tarea por no encontrar un evento asociado.

Pre Procesado	Matching	Dumping	Total	Hilos
0,1720	2,1500	0,2120	2,5280	0
0,1350	1,8840	0,2070	2,2260	1
0,1290	2,4740	0,2730	2,6850	2
0,1310	1,4730	0,2990	1,9010	4
0,1340	0,9510	0,3160	1,4000	8
0,1390	0,6070	0,3410	1,0870	16
0,1320	0,4330	0,3490	0,9150	32
0,1130	0,4460	0,3650	0,9280	64

Tabla 6.3: Tiempo de ejecución de la traza HPL

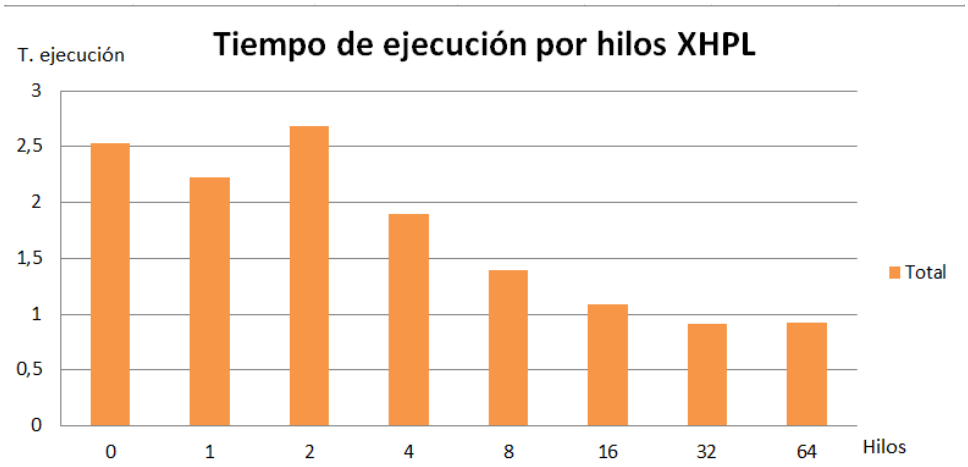

Figura 6.3: Gráfica de evaluación de la prueba XHPL

A partir de 4 hilos, sí que se observa que se mejoran los tiempos de ejecución. El punto máximo de rendimiento son los 32 hilos, que alcanza un tiempo medio de ejecución de 0,9150 segundos. Esto produce una reducción del tiempo de ejecución del 63,8 %, consiguiendo un *speed-up* de 2,76X.

La traza generada tiene un total de 32 tareas MPI, por lo que el despliegue de más hilos de las tareas existentes, no tiene mucho sentido, ya que 32 hilos ya están asignados a las 32 tareas, un hilo por tarea y el trabajo que haga el resto de hilos es inútil. Esto se ve reflejado en la ejecución de 64 hilos, que se aprecia cómo el tiempo de ejecución de 64 hilos se queda a un nivel muy cercano del tiempo medio obtenido con 32 hilos, incluso incrementa el tiempo de ejecución ligeramente.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

El proyecto ha conseguido lograr sus objetivos iniciales con éxito. Se ha desarrollado una versión paralela de la aplicación *VEF Mixer* y se ha conseguido aumentar el rendimiento de dicha aplicación en al menos 2X, es decir, se ha reducido el tiempo de ejecución en un 50 %.

Para la realización de este proyecto, se han podido aplicar los conocimientos aprendidos en las asignaturas de la carrera, en especial Computación Paralela (CPAR), ya que la paralelización es el foco de este trabajo, y Rendimiento y Evaluación de Computadores (REC), para analizar los resultados de las pruebas ejecutadas y obtener las mejoras de rendimiento en base al programa original. Se ha trabajado en un sistema operativo Linux, utilizando la terminal frecuentemente, por lo que las asignaturas de Fundamentos de Sistemas Operativos (FSO) y Estructura de Sistemas Operativos (ESO) también han resultado útiles. También se han aplicado los conocimientos de Planificación y Gestión de Proyectos para este proyecto.

Ha sido un trabajo interesante a la par que complicado, ya que la comprensión del programa a nivel funcional requería un análisis exhaustivo para no perder los pequeños detalles. Este análisis ha permitido llevar a cabo el desarrollo correctamente y conseguir la consecución del proyecto.

7.2. Líneas de trabajo futuras

A pesar de que el proyecto ha cumplido con sus objetivos, siempre cabe espacio para la mejora. Algunas posibles líneas de trabajo futuras pueden ser:

- Optimización de la paralelización. Se puede continuar este proyecto, intentando conseguir mejor rendimiento cambiando las estructuras de datos de la aplicación, e inves-

tigando cómo lograr más escalado de paralelización en la parte de las comunicaciones colectivas.

- Aplicar paralelización con MPI. Otro proyecto interesante podría ser aplicar una paralelización con MPI en lugar de OpenMP, o incluso combinando las 2 tecnologías al mismo tiempo, y realizar nuevas evaluaciones sobre resultados obtenidos.
- Estudiar el impacto del tamaño del chunk en el proceso de generación de la traza. Es posible que un cambio en la cantidad de eventos que se procesan de una sola vez afecte al rendimiento, y sería interesante una comparativa analizando distintos tamaños.

Bibliografía

[1] F. J. Andujar, J. A. Villar, J. L. Sanchez, F. J. Alfaro, y J. Escudero-Sahuquillo, "VEF traces: A framework for modelling MPI traffic in interconnection network simulators", en 2015 IEEE International Conference on Cluster Computing, 2015, pp. 841–848.

[2] F. J. Andújar, J. A. Villar, F. J. Alfaro, J. L. Sánchez, y J. Escudero-Sahuquillo, "An open-source family of tools to reproduce MPI-based workloads in interconnection network simulators", J. Supercomput., vol. 72, núm. 12, pp. 4601–4628, 2016.

[3] "VEF-Prospector", GitLab. [En línea]. Disponible en: `https://gitraap.i3a.info/fandujar/VEF-Prospector`. [Consultado: 01-mar-2023].

[4] "VEF-TraceLIB", GitLab. [En línea]. Disponible en: `https://gitraap.i3a.info/fandujar/VEF-TraceLIB`. [Consultado: 01-mar-2023].

[5] J. Cano-Cano, F. J. Andujar, F. J. Alfaro, y J. L. Sanchez, "VEF3 traces: Towards a complete framework for modelling network workloads for exascale systems", en 2018 IEEE 4th International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), 2018, pp. 32–39.

[6] "Proyectos SEA", Sea-projects.eu. [En línea]. Disponible en: `https://sea-projects.eu/`. [Consultado: 03-mar-2023]

[7] "The VEF traces framework", Redsea-project.eu. [En línea]. Disponible en: `https://redsea-project.eu/the-vef-traces-framework/`. [Consultado: 01-mar-2023].

[8] "The traditional waterfall approach", Umsl.edu. [En línea]. Disponible en: `https://www.umsl.edu/~hugheyd/is6840/waterfall.html`. [Consultado: 15-mar-2023].

[9] W. W. Royce. 1987. Managing the development of large software systems: concepts and techniques. In Proceedings of the 9th international conference on Software Engineering (ICSE '87). IEEE Computer Society Press, Washington, DC, USA, 328–338. Disponible en: `https://www.praxisframework.org/files/royce1970.pdf` [Consultado: 15-mar-2023].

[10] A. C. Team, «Waterfall Methodology: Project Management — Adobe Workfront». [En línea]. Disponible en: `https://business.adobe.com/blog/basics/waterfall` [Consultado: 16-mar-2023].

[11] "MPI: A Message-Passing Interface Standard", Mpi-forum.org. [En línea]. Disponible en: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf. [Consultado: 17-mar-2023].

[12] Researchgate.net. [En línea]. Disponible en: https://www.researchgate.net/profile/Frank-Nielsen-3/publication/327797677_Introduction_to_HPC_with_MPI_for_Data_Science/links/5d994e73458515c1d398993a/Introduction-to-HPC-with-MPI-for-Data-Science.pdf. [Consultado: 17-mar-2023].

[13] "Introducción a MPI", Informatica.uv.es. [En línea]. Disponible en: http://informatica.uv.es/iiguia/ALP/materiales2005/2_2_introMPI.htm. [Consultado: 20-mar-2023].

[14] "Introduction to MPI". Kadin Tseng, Boston University, Research Computing Services. [En línea]. Disponible en: https://www.bu.edu/tech/files/2014/06/intro2MPI.pdf. [Consultado: 21-mar-2023].

[15] I. Foster, "8 Message Passing Interface", Anl.gov. [En línea]. Disponible en: https://www.mcs.anl.gov/~itf/dbpp/text/node94.html. [Consultado: 22-mar-2023].

[16] "MPI Communication: Sending and Receiving Message", Utm.my. [En línea]. Disponible en: https://people.utm.my/nur-haliza/files/2020/09/06-MPI-communications.pdf. [Consultado: 22-mar-2023].

[17] T. Mattson y L. M. P. Engineer, "A 'Hands-on' Introduction to OpenMP", Openmp.org. [En línea]. Disponible en: https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf. [Consultado: 25-mar-2023].

[18] "OpenMP Shared-Memory Parallel Programming", Orian Louant, November 2020. [En línea]. Disponible en: https://indico.cism.ucl.ac.be/event/72/attachments/78/175/omp-slides.pdf. [Consultado: 25-mar-2023].

[19] "Web Oficial de GitLab", GitLab . [En línea]. Disponible en: https://gitlab.com/gitlab-org/gitlab. [Consultado: 30-mar-2023].

[20] "Discord", Discord. [En línea]. Disponible en: https://discord.com/. [Consultado: 02-abr-2023].

[21] L. M. A. Extensions, "Visual Studio Code - code editing. Redefined", Visualstudio.com. [En línea]. Disponible en: https://code.visualstudio.com/. [Consultado: 02-abr-2023].

[22] "Welcome home: Vim online", Vim.org. [En línea]. Disponible en: https://www.vim.org/. [Consultado: 05-abr-2023].

[23] "Overleaf, Editor de LaTeX online", Overleaf.com. [En línea]. Disponible en: https://es.overleaf.com/. [Consultado: 07-abr-2023].

[24] "Overleaf, Home", GitHub. [En línea]. Disponible en: https://github.com/overleaf/overleaf/wiki. [Consultado: 07-abr-2023].

[25] "LaTeX - A document preparation system", Latex-project.org. [En línea]. Disponible en: https://www.latex-project.org/. [Consultado: 07-abr-2023].

[26] B. W. Kernighan y D. Ritchie, C Programming Language. Prentice Hall, 1988. Disponible en: https://colorcomputerarchive.com/repo/Documents/Books/The%20C%20Programming%20Language%20(Kernighan%20Ritchie).pdf. [Consultado: 02-jul-2023].

[27] "C language", Cppreference.com. [En línea]. Disponible en: https://en.cppreference.com/w/c/language. [Consultado: 15-abr-2023].

[28] "Advanced Bash-Scripting Guide",Mendel Cooper Tldp.org. [En línea]. Disponible en: https://tldp.org/LDP/abs/html/index.html. [Consultado: 17-abre-2023].

[29] G. Who?, "Bash - GNU project - free software foundation", Gnu.org. [En línea]. Disponible en: https://www.gnu.org/software/bash/. [Consultado: 17-abr-2023].

[30] "Home", OpenMP, 12-ene-2015. [En línea]. Disponible en: https://www.openmp.org/. [Consultado: 18-abr-2023].

[31] "CMake", Cmake.org. [En línea]. Disponible en: https://cmake.org/. [Consultado: 23-abr-2023].

[32] BarD Software s. r. o, "GanttProject: free project management tool for Windows, macOS and Linux", GanttProject. [En línea]. Disponible en: https://www.ganttproject.biz/. [Consultado: 29-abr-2023].

[33] "Grupo trasgo", Uva.es. [En línea]. Disponible en: https://trasgo.infor.uva.es/. [Consultado: 10-may-2023].

[34] "Simultaneous Multithreading home page", Washington.edu. [En línea]. Disponible en: https://dada.cs.washington.edu/smt/. [Consultado: 15-may-2023].

[35] "MPICH", Mpich.org. [En línea]. Disponible en: https://www.mpich.org/. [Consultado: 20-may-2023].

[36] "Welcome to GROMACS — GROMACS webpage", Gromacs.org. [En línea]. Disponible en: https://www.gromacs.org/. [Consultado: 25-may-2023].

[37] "Alexnet", Mathworks.com. [En línea]. Disponible en: https://es.mathworks.com/help/deeplearning/ref/alexnet.html. [Consultado: 26-may-2023].

[38] "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers", ICL - UTK Computer Science Department, A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary. Netlib.org. [En línea]. Disponible en: https://netlib.org/benchmark/hpl/. [Consultado: 06-jun-2023].

[39] "Parallel Programming: Speedups and Amdahl's law", Oregon State University, Oregonstate.edu. [En línea]. Disponible en: https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/speedups.and.amdahls.law.1pp.pdf. [Consultado: 20-jun-2023].

[40] "Flowchart maker & online diagram software", Diagrams.net. [En línea]. Disponible en: https://app.diagrams.net/. [Consultado: 26-jun-2023].

[41] "GCC, the GNU compiler collection", Gnu.org. [En línea]. Disponible en: https://gcc.gnu.org/. [Consultado: 29-jun-2023].

Apéndice A

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/-/tree/mixer_OMP`.
- Repositorio de VEF-Prospector: `https://gitraap.i3a.info/fandujar/VEF-Prospector`
- Repositorio de VEF-TraceLIB: `https://gitraap.i3a.info/fandujar/VEF-TraceLIB`
- Tutorial sobre el funcionamiento de VEF Traces:
  `https://redsea-project.eu/the-vef-traces-framework/`