Universidad Politécnica
de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

OpenMP para Memoria Distribuida: OpenMPD. Definición de tipos de datos.

Autor: Álvaro Plaza Carro

Tutor(a): José Luis Pedraza Domínguez

Madrid, junio de 2024

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: OpenMP para Memoria Distribuida: OpenMPD. Definición de tipos de datos.
Junio 2024
Autor: Álvaro Plaza Carro

Tutor:

José Luis Pedraza Domínguez
Departamento de Arquitectura y Tecnología de Sistemas Informáticos (DATSI)
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

El presente Trabajo Fin de Grado forma parte de un conjunto de cuatro TFGs propuestos por profesores del DATSI con objeto de desarrollar una herramienta que permita integrar programación paralela y programación distribuida en un único modelo basado en el lenguaje C. El modelo utiliza directivas de OpenMP ampliadas con otras relacionadas con MPI, formando lo que se ha denominado OMPD (OpenMP Distribuido).

Dada la complejidad de esta propuesta, se planteó una distribución del desarrollo del proyecto global en cuatro subproyectos, cada uno de los cuales está encaminado a desarrollar e incorporar a OMPD un subconjunto de directivas de MPI asociadas a diferentes conceptos de este modelo de distribución de datos: 1-Definición de datos, 2-Reparto de iteraciones, 3-Envío de datos y 4-Recuperación de resultados.

El presente TFG corresponde al primero de los indicados, que se encarga de desarrollar y documentar las declaraciones correspondientes a la definición de tipos de datos.

Se utilizarán las herramientas Bison y Flex para el construir el *parser* y el scanner respectivamente, partiendo de un analizador de la arquitectura C99 de C y un analizador de OpenMP con gramática ampliada. Una primera etapa será la de crear y diseñar la tabla de símbolos, necesaria para registrar los tipos, tamaños y parámetros de las distintas variables y funciones. Posteriormente tendremos que incluir el manejo de la tabla de símbolos a la gramática.

La siguiente etapa consistiría en traducir las directivas OMPD de definición de tipo de datos hacia MPI, asegurando la transparencia para el resto del código.

Por último, se validará el funcionamiento correcto mediante varios ejemplos de pequeña entidad ajustados a las nuevas directivas.

La meta no es la de mejorar la eficiencia de las aplicaciones paralelas, esto es algo que ya se logra con la combinación de OpenMP y MPI. OpenMPD es una extensión del lenguaje OpenMP diseñada para facilitar la programación en sistemas de memoria distribuida. Aunque MPI es el estándar para estos sistemas, su complejidad dificulta su uso. OpenMPD simplifica la paralelización mediante directivas simples y claras, similar a OpenMP, pero para entornos de memoria distribuida.

Abstract

This Final Degree Project is part of a set of four TFGs proposed by professors from DATSI, aimed at developing a tool that integrates parallel and distributed programming into a single model based on the C language. The model uses OpenMP directives extended with those related to MPI, forming what has been termed OMPD (Distributed OpenMP).

Given the complexity of this proposal, the overall project development was divided into four subprojects, each focused on developing and incorporating into OMPD a subset of MPI directives associated with different concepts of this data distribution model: 1-Data Definition, 2-Iteration Distribution, 3-Data Sending, and 4-Result Retrieval.

This TFG corresponds to the first of the aforementioned subprojects, which is responsible for developing and documenting the declarations related to data type definitions. Bison and Flex tools will be used for building the parser and scanner, respectively, starting from a C99 architecture analyzer and an OpenMP analyzer with extended grammar. The initial stage involves creating and designing the symbol table, necessary for recording the types, sizes, and parameters of various variables and functions. Subsequently, the symbol table handling will be incorporated into the grammar.

The next stage involves translating OMPD data type definition directives to MPI, ensuring transparency for the rest of the code. Finally, the correct functionality will be validated through several small-scale examples adjusted to the new directives.

The goal is not to improve the efficiency of parallel applications, which is already achieved with the combination of OpenMP and MPI. OpenMPD is an extension of the OpenMP language designed to facilitate programming in distributed memory systems. Although MPI is the standard for these systems, its complexity makes it difficult to use. OpenMPD simplifies parallelization through simple and clear directives, similar to OpenMP, but for distributed memory environments.

Tabla de contenidos

1 Introducción

1.1 Contexto y Relevancia de la Computación de Alto Rendimiento

En la era moderna, la computación de alto rendimiento (HPC) ha evolucionado para abordar algunos de los desafíos más exigentes en diversas áreas como la ciencia, la ingeniería, y la investigación médica. A medida que las aplicaciones se vuelven más complejas y los volúmenes de datos crecen exponencialmente, la necesidad de procesamiento paralelo y distribuido se ha vuelto más crítica que nunca. En este contexto, tecnologías como OpenMP (Open Multi-Processing) y MPI (Message Passing Interface) han emergido como pilares fundamentales en el desarrollo de aplicaciones que pueden aprovechar eficazmente múltiples procesadores y máquinas distribuidas.

OpenMP es ampliamente utilizado para la programación paralela en sistemas de memoria compartida, ofreciendo una forma sencilla y flexible de escribir programas que pueden ejecutarse en múltiples núcleos de un procesador. Por otro lado, MPI ha sido el estándar de facto para la programación en sistemas de memoria distribuida, permitiendo la comunicación eficiente entre nodos en un clúster de computadoras. Aunque cada uno de estos enfoques ofrece soluciones robustas para escenarios específicos, la integración de estos dos modelos en aplicaciones que se benefician simultáneamente de la memoria compartida y distribuida aún representa un desafío significativo debido a su complejidad.

En este contexto surge OpenMPD (OpenMP Distribuido), una extensión del lenguaje OpenMP diseñada específicamente para facilitar la programación en sistemas de memoria distribuida. OpenMPD busca simplificar la paralelización mediante directivas simples y claras, similares a las de OpenMP, pero adaptadas para entornos de memoria distribuida. Esta extensión tiene como objetivo combinar la accesibilidad de OpenMP con la potencia y flexibilidad de MPI.

1.2 Objetivos y Alcance del Proyecto

El proyecto tiene como objetivo la creación de un traductor de código fuente a código fuente que permita la adaptación de código utilizando directivas OpenMPD hacia MPI, buscando un equilibrio entre la accesibilidad de OpenMP y la potencia de MPI en entornos distribuidos. Para lograr este objetivo, se plantean los siguientes objetivos específicos:

Adquirir conocimientos de OpenMPD y MPI:

Aprender las características básicas y las funcionalidades de OpenMP Distribuido (OMPD) y Message Passing Interface (MPI) para poder diseñar un

traductor eficiente que integre ambas tecnologías, facilitando su uso en entornos de computación distribuida.

Dominar las herramientas de desarrollo del traductor de código fuente, especialmente Bison y Flex:

Familiarizarse con Bison y Flex para realizar un análisis sintáctico y léxico adecuado. El objetivo es construir un traductor que maneje las estructuras del código fuente en C y traduzca correctamente las directivas de OpenMPD a llamadas de MPI.

Implementar la traducción de directivas de OpenMPD orientadas a memoria distribuida a llamadas de MPI:

Desarrollar un método para traducir eficientemente las directivas de OpenMPD relacionadas con la gestión de memoria distribuida a su equivalente en MPI, asegurando la coherencia y funcionalidad del código traducido.

Crear ejemplos y validar el funcionamiento del traductor:

Generar ejemplos de código que utilicen las directivas traducidas para verificar la funcionalidad del traductor y realizar ajustes necesarios para mejorar la precisión y fiabilidad del mismo.

Documentar el proceso de desarrollo y los resultados obtenidos en informes intermedios y en la memoria final del trabajo:

Documentar de forma organizada las etapas del desarrollo del traductor, incluyendo los problemas encontrados y las soluciones aplicadas, para proporcionar un registro completo del proyecto.

Preparar una presentación efectiva del proyecto para su defensa ante un panel académico:

Elaborar una presentación que detalle los aspectos técnicos y los resultados del proyecto, comunicando claramente su valor y sus aportaciones al campo de la computación distribuida.

El proyecto se centrará exclusivamente en la traducción de directivas de memoria distribuida de OpenMPD a MPI, sin abordar la totalidad de las posibles directivas y funcionalidades de MPI ni otros aspectos de la programación en OpenMP que no estén directamente relacionados con la memoria distribuida. Además, el desarrollo del traductor se limitará a entornos de prueba controlados, y no se extenderá a la optimización para hardware específico o configuraciones de red particulares.

1.3 Justificación y Relevancia del Estudio

Este proyecto es esencial por varias razones estratégicas y técnicas. La capacidad de integrar directivas de memoria distribuida en OpenMP con la robustez del modelo de comunicación de MPI ayudará a reducir la curva de aprendizaje de la programación paralela y distribuida.

Rellenar el vacío en herramientas existentes: Actualmente, las herramientas que facilitan la traducción o la interoperabilidad entre OpenMP y MPI son limitadas y no satisfacen completamente las necesidades de los desarrolladores que trabajan con sistemas de memoria distribuida. La necesidad de tal integración ha sido discutida en trabajos como el de Rabenseifner (2009) [1], que exploran las sinergias entre MPI y OpenMP en contextos de computación de alto rendimiento.

Promover la adopción de estándares abiertos: Al centrarse en OpenMPD y MPI, ambos estándares abiertos ampliamente aceptados en la comunidad científica y de ingeniería, el proyecto facilita la interoperabilidad y la portabilidad entre diferentes plataformas y arquitecturas. Esto es crucial para acercar el uso de grandes infraestructuras computacionales a una base de programadores mucho más extensa.

Contribución al avance tecnológico: Integrar estas tecnologías puede conducir a avances significativos en el campo de la computación de alto rendimiento (HPC). Al facilitar el desarrollo de las aplicaciones que requieren alto grado de paralelismo y distribución, este proyecto contribuye directamente a la capacidad de realizar simulaciones complejas y procesamiento de grandes volúmenes de datos más rápidamente y con menor consumo de recursos. Dagum y Menon (1998) [2] describen cómo el diseño optimizado de bibliotecas como MPI ha revolucionado el procesamiento paralelo, apoyando la relevancia de su uso a través de este proyecto.

Impacto académico y formativo: Este proyecto no solo proporciona una herramienta útil para la comunidad científica, sino que también sirve como una plataforma educativa para profundizar en el campo del paralelismo y la computación distribuida. A través de este proyecto, se adquirirán conocimientos clave como análisis sintáctico, diseño de software y optimización de rendimiento, que son ampliamente demandadas en la industria tecnológica. El valor educativo de tales proyectos ha sido destacado por la ACM en sus recomendaciones curriculares.

Viabilidad y sostenibilidad: Al ser diseñado con componentes y tecnologías de código abierto, el proyecto asegura una base sostenible y adaptable para futuras expansiones y modificaciones. La posibilidad de que la comunidad contribuya al desarrollo y mejora del traductor asegura su relevancia y utilidad a largo plazo.

1.4 Estructura General del Proyecto

Este proyecto se enfoca en desarrollar una versión inicial del traductor capaz de manejar un subconjunto limitado de nuevas directivas de OMPD, generando las correspondientes traducciones en directivas y cláusulas de MPI. Dado el alcance y la complejidad del trabajo, el proyecto se dividió en cuatro Trabajos Fin de Grado (TFGs) que se desarrollaron de forma colaborativa. El propósito común de estos TFGs es integrar la programación paralela y distribuida en un único modelo basado en el lenguaje C, utilizando directivas de OpenMP y MPI para formar OMPD (OpenMP Distribuido).

En mi TFG, me he enfocado en desarrollar y documentar las declaraciones correspondientes a la definición de tipos de datos. Los otros tres TFGs se centraron en los siguientes aspectos: 1) Reparto de iteraciones [3], 2) Envío de datos [4], y 3) Recolección de resultados [5].

Durante las primeras fases del trabajo, enfrentamos varios desafíos al intentar modificar el repositorio existente de *Mercurium* [6], un compilador de código abierto desarrollado en la Universidad de Barcelona que prometía ser una base sólida para nuestro desarrollo. Sin embargo, nos encontramos con numerosas dificultades relacionadas con librerías desactualizadas y problemas de compilación que obstaculizaron significativamente el progreso. Esta situación nos llevó a tomar la decisión de empezar desde cero, evitando así las complejidades inherentes a la actualización de un sistema legado.

El desarrollo del proyecto comenzó entonces evaluando alternativas de desarrollo y optando por herramientas más elementales, pero bien probadas y documentadas, como Bison y Flex (ya escritas). Estas herramientas nos permitieron un control más preciso sobre el proceso de análisis léxico y sintáctico.

Para la implementación, utilizamos las gramáticas de C99 y OpenMP como base. La gramática de C99 proporciona una base sólida para el análisis sintáctico del lenguaje C, mientras que la gramática de OpenMP añade soporte para directivas de paralelismo. La combinación de ambas gramáticas junto a la ampliación de la gramática de OpenMP, fue un proceso complejo que requirió una integración cuidadosa.

El reparto inicial del trabajo fue el siguiente:

- Otro miembro del equipo y yo nos dedicamos a generar la tabla de símbolos, una estructura esencial para registrar los tipos, tamaños y parámetros de las distintas variables y funciones.
- Un tercer miembro del equipo se encargó de fusionar las gramáticas de C99 y OpenMP, asegurando que ambas pudieran trabajar juntas de manera coherente.
- El cuarto miembro del equipo se enfocó en incorporar a la gramática OpenMP las nuevas directivas y cláusulas de OMPD.

Este enfoque colaborativo ha permitido un desarrollo más organizado, asegurando que cada componente del proyecto se integre de manera coherente en la solución final, y simplificando parte de la complejidad del proyecto para cada uno de los integrantes.

2 Marco teórico

2.1 Conceptos Básicos de MPI (Message Passing Interface)

Introducción a MPI:

El Message Passing Interface (MPI) es un estándar de comunicación que permite a programas que se ejecutan en múltiples procesadores con memoria distribuida intercambiar mensajes para realizar cálculos paralelos. MPI es esencial en la computación de alto rendimiento (HPC) porque facilita la implementación de algoritmos en entornos de supercomputación, clusters y sistemas distribuidos.

Características Principales de MPI:

- **Estandarización y portabilidad**: MPI proporciona una interfaz estándar que garantiza la portabilidad de aplicaciones paralelas a través de diferentes plataformas de hardware y software.
- **Modelo de comunicación**: Permite el envío y recepción de mensajes entre procesos, soportando tanto la comunicación punto a punto como la colectiva.
- **Control de datos y sincronización**: Ofrece métodos para controlar el flujo de datos y sincronizar procesos para maximizar la eficiencia y evitar condiciones de carrera.

Funcionamiento de MPI:

MPI opera mediante el envío y recepción de mensajes entre procesos individuales, lo que permite un control detallado sobre la computación distribuida y la paralelización. Los programas que utilizan MPI inician múltiples instancias del mismo programa en diferentes nodos o procesadores, donde cada instancia realiza parte del cálculo y se comunica con las otras según sea necesario.

Tipos de Comunicación en MPI:

- **Comunicación Punto a Punto**: Implica el envío de mensajes entre dos procesos específicos. Es útil para tareas que requieren intercambio de datos entre un emisor y un receptor determinados.
- **Comunicación Colectiva**: Involucra a todos los procesos en un grupo y es crucial para operaciones que necesitan coordinación y datos de múltiples fuentes, como barreras de sincronización, difusión (broadcast), y reducción.

Aplicaciones y Ventajas de MPI:

El uso de MPI permite la escalabilidad y eficiencia en aplicaciones que requieren grandes volúmenes de cálculos y gestión de datos, tales como simulaciones

científicas, análisis financiero, y modelado climático. Además, MPI es altamente eficiente en la utilización de recursos de computación distribuida, permitiendo que las aplicaciones se escalen a un gran número de procesadores sin degradar el rendimiento.

Desafíos en la Implementación de MPI:

La programación con MPI requiere una comprensión profunda de la arquitectura de los sistemas distribuidos y de las dinámicas de comunicación entre procesos. Los errores de sincronización, la gestión de procesos que se eliminan o mueren, y la complejidad del diseño son desafíos comunes en la programación con MPI.

2.2 Fundamentos de OpenMP y OpenMP Distribuido (OMPD)

Introducción a OpenMP:

OpenMP (Open Multi-Processing) es un modelo de programación en paralelo que utiliza directivas del compilador, bibliotecas de rutinas y variables de entorno para crear aplicaciones eficientes en plataformas de memoria compartida. Desde su introducción a finales de los años 90, OpenMP ha sido ampliamente adoptado debido a su simplicidad para desarrollar programas paralelos, especialmente para sistemas con múltiples procesadores y núcleos.

Características Principales de OpenMP:

- **Directivas del compilador**: Permiten al programador especificar regiones de código para ejecutar en paralelo.
- **Constructores de sincronización**: Controlan el acceso a las regiones críticas y coordinan el trabajo entre varios hilos.
- **Modelo de memoria compartida**: Todos los hilos tienen acceso a una memoria común, facilitando la comunicación y la sincronización entre ellos.

OpenMP Distribuido (OMPD)

A medida que la computación en clusters y sistemas distribuidos ganó popularidad, surgieron varios paquetes para extender OpenMP para aprovechar entornos de memoria no compartida. OpenMP Distribuido (OMPD) es una extensión de OpenMP diseñada para funcionar en sistemas distribuidos, donde cada nodo de procesamiento tiene su propia memoria local.

Diferencias entre OpenMP y OpenMPD:

- **Modelo de memoria**: OpenMP usa memoria compartida, donde los *threads* acceden a una memoria común, simplificando la sincronización, pero limitando la escalabilidad. OpenMPD, en cambio, utiliza memoria distribuida, con cada nodo gestionando su propia memoria local y comunicándose mediante mensajes, lo que permite mayor escalabilidad y eficiencia en clústeres y supercomputadoras.
- **Directivas extendidas**: OpenMP utiliza directivas sencillas como `#pragma omp parallel` y `#pragma omp for` para paralelizar secciones del código en memoria compartida. OpenMPD introduce directivas adicionales como `#pragma ompd distvar` para declarar variables distribuidas y `#pragma ompd sync` para sincronizar datos entre nodos, adaptando así la facilidad de uso de OpenMP a la complejidad de la memoria distribuida, permitiendo a los desarrolladores gestionar explícitamente la distribución y sincronización de datos en entornos de memoria distribuida.

Aplicaciones y Ventajas de OpenMPD:

La extensión de OpenMP hacia sistemas distribuidos ofrece varias ventajas, incluyendo:

- **Escalabilidad**: Permite aprovechar mejor los recursos en clústeres y supercomputadoras, escalando eficientemente a un gran número de nodos.
- **Flexibilidad**: Facilita la combinación de directivas de OpenMPD con codificación explícita en MPI para manejar aplicaciones más complejas
- **Interoperabilidad con MPI**: OpenMPD puede utilizarse en combinación con toda la funcionalidad de MPI, añadiendo al código fuente OMPD directivas y elementos de MPI que no se han incorporado a OMP (MPI es muchísimo más extenso que las directivas que se han incorporado a OpenMPD) para optimizar aún más el paralelismo híbrido en aplicaciones que requieren un alto grado de escalabilidad.

Retos en la Implementación de OpenMPD:

A pesar de sus beneficios, la implementación de OpenMPD presenta desafíos específicos, como la complejidad en la gestión de la memoria distribuida y la necesidad de sincronización eficiente entre nodos. Estos desafíos requieren herramientas y estrategias específicas para asegurar que el rendimiento y la eficiencia no se vean comprometidos.

2.3 Trabajos Previos y Tecnologías Relacionadas

La integración de OpenMP y MPI ha sido explorada en múltiples contextos de alto rendimiento, donde la combinación de memoria compartida y comunicación entre procesos distribuidos ha demostrado ser especialmente beneficiosa:

Simulación de Dinámica de Fluidos Computacional (CFD):

- **Nasa Advanced Supercomputer Division (NAS):** Investigadores del NAS utilizaron MPI para gestionar la comunicación entre nodos de cálculo y OpenMP para paralelizar las operaciones dentro de cada nodo. Este enfoque híbrido permitió un escalado eficiente de simulaciones de CFD en supercomputadoras, mejorando significativamente los tiempos de ejecución y la eficiencia del uso de recursos [7].

Proyectos de Modelado Climático:

- "**Parallel Computations in Problems of Climate Modeling**": Este estudio aborda la implementación paralela de modelos climáticos en multiprocesadores de clúster, utilizando tanto MPI como OpenMP para manejar componentes atmosféricos y oceánicos. Las pruebas realizadas en el modelo atmosférico global INM AGCM y benchmarks atmosféricos han demostrado la eficacia de esta implementación [8].

Implementación Paralela en Análisis de Datos a Gran Escala:

- **Proyectos en Bioinformática:** El análisis de datos genómicos y proteómicos a gran escala se ha beneficiado enormemente de la implementación paralela utilizando MPI y OpenMP. Al dividir las grandes bases de datos en fragmentos más pequeños y procesarlos en paralelo, se ha logrado una reducción significativa en los tiempos de análisis, facilitando investigaciones más profundas y rápidas en áreas críticas como la medicina personalizada y la epidemiología [9].

La evolución de la computación de alto rendimiento ha sido acompañada por el desarrollo de nuevas tecnologías que complementan o potencian el uso de OpenMP y MPI:

Computación en la Nube:

- **Plataformas como AWS, Azure y Google Cloud**: Estas plataformas ofrecen servicios que permiten desplegar aplicaciones MPI y OpenMP en la nube, proporcionando escalabilidad, flexibilidad y acceso a hardware de alto rendimiento sin necesidad de infraestructura física propia.

Computación en GPUs:

- **CUDA de NVIDIA y OpenCL**: Estas tecnologías permiten la programación de GPUs para realizar cálculos paralelos intensivos, que pueden ser integrados con arquitecturas basadas en OpenMP y MPI para mejorar aún más el rendimiento en aplicaciones específicas como la inteligencia artificial y el procesamiento de imágenes.

Frameworks de Paralelismo de Datos:

- **Apache Spark y Dask**: Estos frameworks son utilizados para el procesamiento de grandes volúmenes de datos y análisis en tiempo real, proporcionando una capa de abstracción que puede integrarse con entornos de computación paralela tradicionales para facilitar el manejo de datos a gran escala.

2.4 Bison y Flex: Herramientas para el Análisis Sintáctico y Léxico

Introducción a Bison y Flex:

Bison y Flex son herramientas de automatización utilizadas para generar analizadores sintácticos y léxicos, respectivamente. Originadas como parte del conjunto de herramientas de desarrollo de software en sistemas Unix, estas herramientas son esenciales para la construcción de compiladores y traductores que transforman el código fuente en una forma que una máquina puede ejecutar. Mas adelante hablaremos del uso e implementación de las gramáticas de C y OMPD en nuestro proyecto.

Funcionalidades y Operación de Flex:

- **Generador de Analizadores Léxicos**: Flex es un generador de analizadores léxicos que lee un archivo de especificaciones definido por el usuario y produce código fuente en C para un escáner léxico. Este escáner descompone la entrada de texto en tokens, que son las unidades básicas de sintaxis que se utilizarán en el análisis sintáctico.

- **Uso en Proyectos de Software**: Flex se utiliza comúnmente para identificar palabras clave, operadores, identificadores y otros elementos sintácticos en el código fuente, facilitando la tarea de análisis y comprensión del código.

Funcionalidades y Operación de Bison:

- **Generador de Analizadores Sintácticos**: Bison es un generador de analizadores sintácticos que toma como entrada una descripción gramatical de un lenguaje de programación y produce un código en C que es capaz de analizar ese lenguaje. Es compatible con gramáticas de tipo LALR (1), que son suficientemente potentes para la mayoría de los lenguajes de programación.
- **Integración con Flex**: A menudo, Bison se utiliza en conjunto con Flex, donde Flex maneja la descomposición léxica y Bison interpreta la estructura gramatical de los tokens generados, produciendo un árbol de análisis sintáctico que refleja la estructura del programa.

Aplicaciones en Proyectos de Traducción de Código:

Bison y Flex son críticos para desarrollar el traductor de OpenMPD a MPI. Estas herramientas permiten:

- **Análisis Eficiente**: Analizar y transformar las directivas OpenMPD en código MPI de manera eficiente, apoyándose en las llamadas a la tabla de símbolos y trabajando en conjunto con el analizador OMPD de gramática ampliada.
- **Manejo de Complejidad Sintáctica**: Manejar la complejidad sintáctica de los programas en C que utilizan OpenMPD, asegurando que todas las transformaciones mantengan la coherencia y la funcionalidad del código original.

Beneficios de Utilizar Bison y Flex:

- **Automatización y Precisión**: Estas herramientas automatizan la creación de componentes críticos de software, reduciendo el riesgo de errores humanos y aumentando la precisión del análisis.
- **Flexibilidad y Personalización**: Permiten la personalización del proceso de análisis para adaptarse a las necesidades específicas del proyecto, como es necesario en la traducción de directivas específicas de memoria distribuida.

3 Diseño y Desarrollo del traductor

3.1 Arquitectura General del Traductor

La arquitectura general del traductor se basa en una serie de componentes y módulos que interactúan entre sí para analizar, procesar y transformar el código fuente de entrada. A continuación, se describen los principales componentes y una ilustración conceptual de la arquitectura:

Diagrama de la Arquitectura

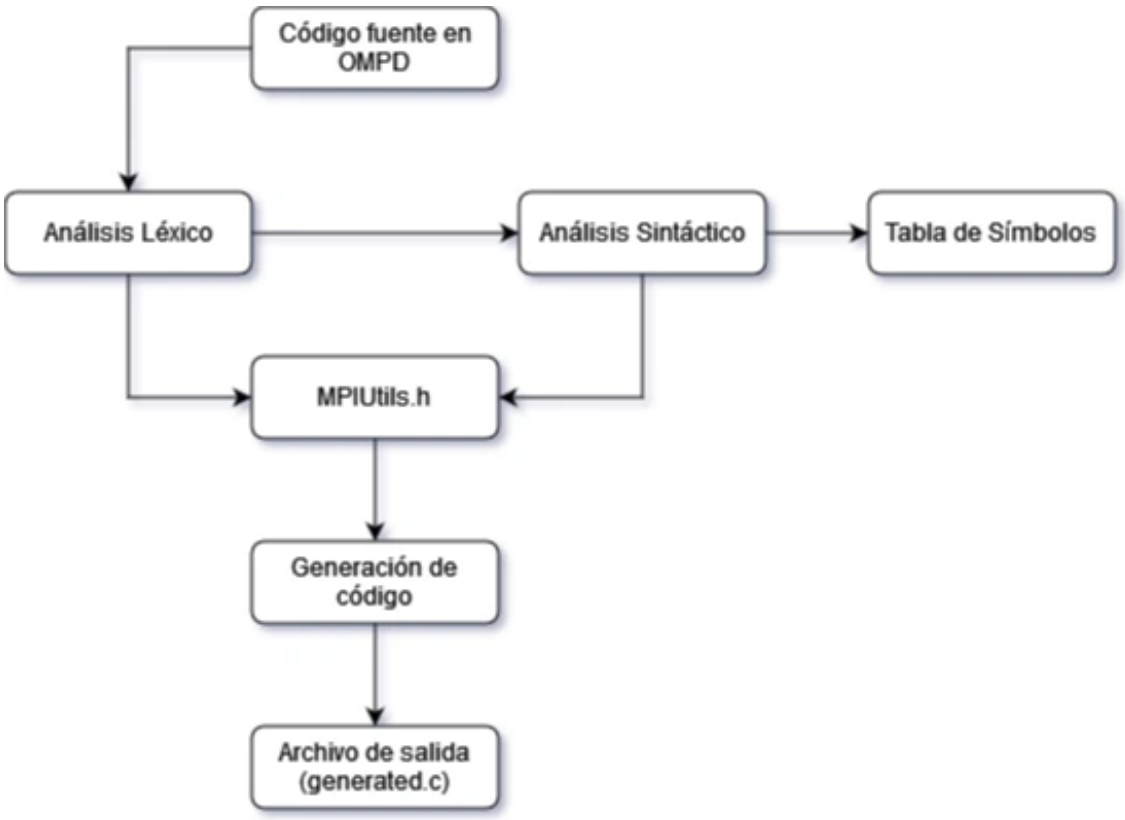

Ilustración 1: Diagrama de arquitectura

Esta arquitectura modular permite que cada componente desempeñe un papel específico en el proceso de traducción, contribuyendo a mantener un funcionamiento eficiente y coordinado del sistema.

Análisis Léxico y Sintáctico

Como ya hemos comentado, el análisis léxico y sintáctico del código fuente se realiza utilizando Flex y Bison respectivamente.

- **Flex (Scanner Léxico)**: Flex se encarga de leer el código fuente y convertirlo en una secuencia de tokens. Cada token representa una unidad léxica significativa como palabras clave, identificadores, operadores y literales. Estos tokens son luego procesados por el analizador sintáctico.
- **Bison (Parser Sintáctico)**: Bison toma los tokens generados por Flex y los organiza en una estructura de árbol sintáctico. Este árbol representa la estructura gramatical del código fuente y es utilizado para realizar las transformaciones necesarias.

Tabla de Símbolos

La tabla de símbolos es una estructura de datos en formato *hashmap* que almacena información sobre las variables, funciones y otros identificadores presentes en el código fuente. Esta información incluye el tipo de dato, el alcance (scope), y cualquier otra característica relevante.

- **Definición y Gestión de Símbolos**: La clase *SymbolTable* y sus componentes (*ScopeTable* y *SymbolInfo*) se encargan de gestionar la inserción, búsqueda y eliminación de símbolos. Esto permite mantener un seguimiento de las variables y funciones definidas en diferentes ámbitos del programa.

MPIUtils

El módulo *MPIUtils.h* es responsable de las funciones necesarias para la traducción de directivas OpenMPD. Este módulo gestiona la creación de tipos MPI, la inicialización y finalización del entorno MPI, y la inserción de las secuencias de código correspondientes.

- **Gestión de Funciones MPI**: Clases como *Funcion* y *MainFuncion* encapsulan la lógica para insertar código específico de MPI antes y después de las funciones originales. Esto incluye la inicialización de variables, la gestión de secuencias de código previo y posterior a la ejecución de directivas OpenMPD, y la definición de tipos MPI personalizados.
- **Gestión de Variables Globales y Cabecera**: La clase principal *MPIUtils* se encarga de añadir las librerías correspondientes a MPI, gestionar y añadir variables globales al archivo de traducción.

- **Traducción de Tipos de Datos**: Se traducirán los tipos definidos en OpenMPD a MPI y se añadirá la inicialización de declaración de tipos a la función *main* correspondiente.

Generación de Código

Una vez que se ha realizado el análisis y se han identificado las transformaciones necesarias, el siguiente paso es la generación del código de salida. Este componente toma el árbol sintáctico y la información de la tabla de símbolos para producir el archivo de traducción.

- **Archivo de Salida**: El código transformado se escribe en un archivo de salida, *generated.c*, que contiene el programa equivalente con las directivas OpenMPD reemplazadas por llamadas a MPI. Este archivo es generado por el módulo *MPIUtils.h* y se asegura de que el código sea funcional y coherente.

Manejo de Errores y Depuración

El sistema también incluye mecanismos para la gestión de errores y la depuración. Estos componentes aseguran que algunos errores en el código fuente original o durante el proceso de traducción sea reportado. Para esto hemos incluido un *flag* DEBUG en el programa.

- **Errores Léxicos y Sintácticos**: Los errores encontrados durante el análisis léxico y sintáctico son manejados por el compilador de C. Sin embargo, decidimos definir funciones específicas que registran errores de inserción por caso de repetición en la tabla de símbolos, o errores sintácticos en caso de fallo en la gramática de C o OMPD. Los mensajes de error se llevarán a los archivo de *log* y de *errores* que proporcionan información detallada para la depuración.
- **Depuración con DEBUG**: El modo de depuración de Bison (yydebug) puede activarse para generar información adicional sobre el proceso de análisis sintáctico, lo cual es útil para detectar y corregir problemas en la gramática del lenguaje. Al igual que durante la depuración también enseñaremos en el fichero de *log* la inserción de tokens en la tabla de símbolos, la entrada y salida de un *scope,* y errores de varios tipos. Al finalizar el programa se generará un archivo “sym_tables.txt” mostrando la información de la tabla de símbolos y tablas scope recogidas durante la depuración del programa.

3.2 Implementación de la Tabla de Simbolos

El trabajo de implementar la tabla de símbolos se nos encargó a otro de mis compañeros de equipo y a mí. En vez de empezar desde cero, buscamos referencias y ejemplos en los que pudiéramos basarnos. Pudimos encontrar varios proyectos que ofrecían distintas estrategias para almacenar la información en la tabla de símbolos, sin embargo, al final nos decidimos por el desarrollo de una tabla de símbolos en formato *hashmap* con resistencia a repeticiones. Pese a la complejidad que puede suponer frente a otras opciones, optamos por este desarrollo debido a su eficiencia en la búsqueda y almacenamiento de símbolos. A continuación, se describe detalladamente la implementación de la tabla de símbolos, utilizando la clase *SymbolTable* y sus componentes.

Estructura de la Tabla de Símbolos

La tabla de símbolos está compuesta por dos clases principales: *ScopeTable* y *SymbolInfo*. Estas clases se encargan de organizar y almacenar la información de los identificadores en el código fuente.

Clase SymbolInfo

La clase *SymbolInfo* representa un símbolo individual en la tabla de símbolos. Cada instancia de *SymbolInfo* contiene la información relevante sobre un identificador, como su nombre, tipo y características adicionales. Los principales atributos y métodos de esta clase son:

Atributos:

- `symbolName`: Nombre del símbolo.
- `symbolType`: Tipo del símbolo.
- `nextSymbol:` Para encadenamiento en caso de repetición de índices en la tabla de símbolos.
- `hashIdx:` Para recoger la información del índice en la tabla de símbolo.
- `varType`: Tipo de variable del símbolo (int, double, char…).
- `paramList`: Lista de parámetros (para funciones, structs o enums).
- `isFunc`, `isType`, `isDef`, `isArr`, `isPoint`, `isStrct`, `hasPragma`: *Flags* que indican características específicas del símbolo.
- `arrSize`: Vector de tamaños de las dimensiones de los arrays (si aplica).

Métodos:

- Métodos getter y setter para los atributos.

- `getParamListString()`: Devuelve una cadena con la lista de parámetros(Útil para el DEBUG de las entradas con parametros).
- `getSizeList()`: Devuelve una cadena con los tamaños de las dimensiones de los arrays (Util para el DEBUG de arrays y matrices).

Clase ScopeTable

Las scope tables son estructuras de datos utilizadas en compiladores para gestionar la visibilidad y el alcance de variables y funciones en diferentes bloques de código. Las scope tables se organizan jerárquicamente, permitiendo que un *scope* hijo acceda a los símbolos del *scope* padre. Esto refleja la estructura anidada de los bloques de código en un programa. Tiene una función parecida a la tabla de símbolos principal.

Funcionalidad

- **Inserción y Búsqueda**: Los símbolos se insertan y buscan utilizando una función hash que maneja colisiones mediante encadenamiento.
- **Scope Padres e Hijos**: Cada *scope* table tiene un puntero a su *scope* padre y puede crear *scope* hijos, formando una jerarquía que facilita la gestión del alcance de los símbolos.
- **Impresión y Depuración**: Proporciona métodos para imprimir la tabla de símbolos del ámbito actual, lo que facilita la depuración y verificación del estado de la tabla.

Atributos:

- `scopeID`: Identificador del *scope*.
- `childCount`: Contador de *scope* hijos.
- `total_buckets`: Número total de espacios en la tabla hash.
- `chainHashTable`: Array de punteros a objetos *SymbolInfo*, representando la tabla hash.
- `parentScope`: Puntero al scope padre.
- `IsScopeReturn`: Flag que podemos activar en el scope para ayudarnos a identificar que nuevos *scope* son funciones y cuáles no.

Métodos:

- `insert(SymbolInfo *symbol)`: Inserta un símbolo en la tabla hash.
- `deleteSymbolFromCurrScope(string key)`: Elimina un símbolo del scope actual.
- `lookup(string key)`: Busca un símbolo en la tabla.

- `printCurr()`: Imprime la tabla de símbolos del ámbito actual (Útil para DEBUG).
- Métodos auxiliares como `sdbmhash(string key)` para calcular el hash de la entrada en la tabla de símbolo, `string key` por lo general será el *name* del Simbolo.

Clase SymbolTable

La gestión de la tabla de símbolos principal a lo largo del programa se realiza mediante la clase *SymbolTable*, que mantiene un puntero al *ScopeTable* actual y proporciona métodos para gestionar los distintos *scope* y símbolos.

Atributos

- `currScopeTable`: Un puntero al *ScopeTable* actual, que representa el ámbito en el que se están definiendo y utilizando los símbolos.
- `tableSize`: El tamaño de la tabla hash utilizada en cada *ScopeTable* para almacenar los símbolos.

Métodos

- `SymbolTable(int tableSize)`: Constructor que inicializa la tabla de símbolos principal con un tamaño de tabla hash especificado.
- `enterScope()`:Crea un nuevo *scope* y lo establece como el *scope* actual. Este método incrementa la profundidad de anidamiento de los *scope*.
- `exitScope()`: Sale del *scope* actual y vuelve al *scope* padre, eliminando el *scope* hijo de la jerarquía de *scope*s.
- `insert(SymbolInfo *newSymbol)`: Inserta un nuevo símbolo en el *scope* actual. Si el símbolo ya existe en el *scope* actual, se muestra un error de inserción y se devuelve *false*.
- `remove(string symbol)`: Elimina un símbolo del *scope* actual. Si el símbolo se encuentra y se elimina correctamente, se devuelve true.
- `lookup(string symbol)`: Busca un símbolo en el *scope* actual y, si no se encuentra, continúa la búsqueda en los *scope* padres. Devuelve true si el símbolo se encuentra.
- `getSymbolInfo(string symbol)`: Devuelve la información del símbolo si se encuentra en el *scope* actual o en sus *scope* padres.
- `printCurrScopeTable()`: Imprime la tabla de símbolos del *scope* actual, mostrando todos los símbolos definidos en ese *scope*. (Clave para el DEBUG de la tabla de símbolos)
- `printAllScopeTables()`: Imprime todas las tablas de símbolos desde el *scope* actual hasta la raíz, mostrando la jerarquía completa de *scopes*. (Clave para el DEBUG de la tabla de símbolos)

3.3 Ampliación de Directivas de OpenMP para incorporar MPI

De la ampliación de directivas de OpenMP se encargaría otro compañero de nuestro equipo. Nuestro compañero añadió las reglas encargadas de la memoria distribuida de OMPD a los archivos *omplexer.ll* y *ompparser.yy*.

Como modelo para la estructura de las nuevas sentencias utilizó las producciones de *target* debido a su similitud con los pragmas a implementar. Codificó los nuevos pragmas añadiendo los tokens que se generaban en las sentencias nuevas y escribiendo las producciones de gramática correspondientes para que se comprobara correctamente el orden de dichos tokens.

Una vez conseguida la ampliación de la gramática se requería fusiona ambas gramáticas OMPD con C99. Esta tarea se le fue encargada a otro de nuestros compañeros. Este tuvo dos opciones: una opción era unificar el analizador léxico de C con el de OMPD y lo mismo con el analizador sintáctico, quedando como resultado solo dos archivos. La otra opción, que es la que se acabó realizando debido a que era más sencilla, fue la de tener los dos analizadores sintácticos y los dos léxicos separados. Tras realizar esto, en el analizador léxico de C99 (*C99-scanner.lex*), cuando detecta un pragma, llama al analizador léxico de OMPD (*omplexer.ll*) para que realice la compilación de la línea del pragma. Tras hacer estos cambios, solo queda compilar todo junto y ya estaría realizada la fusión.

```
%%
"#"             {
                  char * line = get_pragma();
                  char * pragma = strstr(line, "pragma");

                  if (pragma != NULL) {
                      parseOpenMP(pragma+7, NULL);
                  }
                  line_count++;
                  column = 0;
                }
%%
```

Ilustración 2: Check "pragma" en scanner C99-scanner.lex

3.4 Desarrollo del Parser y Scanner

Como ya hemos comentado anteriormente, para el desarrollo del parser y scanner, hacemos uso de Bison y Flex tomando como base la gramática de la arquitectura C99 de C. Tendremos que integrar la tabla de símbolos dentro de la gramática, para así facilitar el manejo de identificadores y sus atributos asociados durante el análisis.

El parser C99 utilizado como referencia es una gramática bien documentada, conocida por su uso en el análisis y compilación de programas en lenguaje C. La estructura de esta gramática incluye definiciones detalladas de expresiones, declaraciones, especificadores de tipo, entre otros elementos esenciales del lenguaje.

Desarrollo del Scanner

El scanner es responsable de la tokenización del código fuente. Flex permite definir patrones de expresiones regulares que se corresponden con los diferentes tokens del lenguaje C. A continuación, se presenta un extracto del código del *C99-scanner.lex*:

```
D           [0-9]
L           [a-zA-Z_]
---------------------------------<SNIP>---------------------------------
"/*"            { comment(); }
"//"[^\n]*              { /* consume //-comment */ }
---------------------------------<SNIP>---------------------------------
"if"            { count(); return(IF); }
"continue"      { count(); return(CONTINUE); }
"double"        { count(); return(DOUBLE); }
"else"          { count(); return(ELSE); }
"float"         { count(); return(FLOAT); }
---------------------------------<SNIP>---------------------------------
{D}*"."{D}+{E}?{FS}?    { count(); return(CONSTANT); }
---------------------------------<SNIP>---------------------------------
L?\"(\\.|[^\\"\n])*\"   { count(); return(STRING_LITERAL); }
---------------------------------<SNIP>---------------------------------
"<"         { count(); return('<'); }
">"         { count(); return('>'); }
"|"         { count(); return('|'); }
---------------------------------<SNIP>---------------------------------
[ \t\v\n\f]     { count(); }
```

Ilustración 3: Código Scanner Tokens

Este código define varios patrones básicos, como números constantes, palabras clave (e.g., *if*, *double*, *float*), *strings*, y otros símbolos del lenguaje.

La función `count()` recogerá información de la posición en que nos encontramos dentro del archivo, contando el número de líneas y columnas, ya función comment() consumirá los comentarios del programa.

Existen casos especiales dentro del scanner en los que querremos devolver el valor de un token. Para estos casos especiales, podemos crear un objeto

SymbolInfo que encapsule tanto el valor como el tipo de simbolo. Este objeto SymbolInfo se puede almacenar en `yylval` y luego ser utilizado por el parser para realizar acciones semánticas específicas.

Por ejemplo, cuando el scanner reconoce un identificador, puede crear un objeto *SymbolInfo* con el nombre del identificador y su tipo ("IDENTIFIER"), y almacenar este objeto en `yylval`:

```
{L}({L}|{D})*       {
    count();
    SymbolInfo *s = new SymbolInfo(yytext, (char *)"IDENTIFIER");
    yylval.sym = s;
    return IDENTIFIER;
}
```

Ilustración 4: Código Scanner Identificador

De manera similar, cuando se reconoce una constante numérica, el scanner puede crear un objeto SymbolInfo que almacene el valor de la constante y su tipo ("CONSTANT"):

```
[1-9]{D}*{IS}?      {
    count();
    SymbolInfo *s = new SymbolInfo(yytext, (char *)"CONSTANT");
    yylval.sym = s;
    return(CONSTANT);
}
```

Ilustración 5: Código Scanner Constante

Estos objetos *SymbolInfo* proporcionan una manera conveniente de transportar información adicional desde el *scanner* al *parser*, facilitando el análisis semántico y la construcción del árbol de análisis sintáctico. En el *parser*, estos objetos se pueden utilizar para diversas tareas, como la inserción en la tabla de símbolos o la comprobación de atributos de un token.

Desarrollo del Parser

El archivo *C99-parser.yacc* contiene las especificaciones de la gramática del lenguaje, las reglas de producción y las acciones semánticas asociadas a cada regla. Estas acciones se utilizan para construir el árbol de análisis sintáctico y ayudar a realizar las transformaciones necesarias en el código.

Reglas de Producción

Las reglas de producción definen la estructura del lenguaje y especifican cómo se pueden combinar los distintos tokens para formar construcciones válidas. A continuación, se presentan unas partes del código, ilustrando algunas de estas reglas:

```
translation_unit
    : external_declaration
    | translation_unit external_declaration
    ;

external_declaration
    : function_definition
    | declaration
    ;
```

Ilustración 6: Comienzo árbol sintáctico

Este primer fragmento de código describe el comienzo del árbol sintáctico:

`translation_unit`: Esta regla define la unidad de traducción en C, que puede ser una única declaración externa o una secuencia de declaraciones externas. En términos prácticos, esto representa el contenido global de un archivo de código fuente en C.

`external_declaration`: Esta regla define lo que constituye una declaración externa en C, que puede ser una definición de función o una declaración de variable.

```
function_definition
    : declaration_specifiers declarator {
        $2->setIsFunction(true);
        $2->setVariableType($1->getSymbolType());
        table.insert($2);
        table.enterScope();
    } declaration_list {
        table.getSymbolInfo($2->getSymbolName())->setParamList($4);
    } compound_statement {
        $2->setIsDefined(true);
        table.exitScope();
    }
```

Ilustración 7: Declaración de función

Este fragmento de código define la regla de producción `function_definition`, describe una de las formas en la que se estructura la definición de una función en el lenguaje C.

`declaration_specifiers`: Especifica el tipo de la función (por ejemplo, *static int*, *float*, *void*...).

`declarator`: Especifica el nombre de la función y sus parámetros.

`declaration_list`: Lista de declaraciones que pueden incluir variables locales o parámetros de la función.

Actualiza la tabla de símbolos con la lista de parámetros de la función.

`compound_statement`: Define el cuerpo de la función, que puede contener múltiples declaraciones y sentencias.

En primer lugar, marcamos el `declarator` como una función con *setIsFunction*. Asignamos el tipo de función al `declarator`, insertamos la función en la tabla de símbolos y entramos en un nuevo *scope*. Este *scope* corresponderá al cuerpo de la función que incluirá la lista de parámetros.

Al final de la declaración marcamos la función como definida y salimos del *scope* de la función. volviendo al *scope* global.

```
statement
    : labeled_statement
    | compound_statement
    | expression_statement
    | { table.enterScope(); } selection_statement { table.exitScope(); }
    | { table.enterScope(); } iteration_statement { table.exitScope(); }
    | jump_statement
    ;
```

Ilustración 8: Sentencias

En el caso de entrar en iteraciones como *if*, *else*, *switch*, *for*, *while* y do-while, entraremos en un nuevo scope antes de procesar la sentencia. Esto asegura que cualquier variable definida dentro de la sentencia esté aislada del ámbito externo. Una vez procesada la sentencia se saldrá del *scope*.

```
declaration
    : declaration_specifiers init_declarator_list ';' {
       $$ = new vector<SymbolInfo*>();
       for(std::vector<SymbolInfo*>::size_type i = 0; i < $2->size(); i++){
        $2->at(i)->setVariableType($1->getSymbolType());
        SymbolInfo* symbol = new SymbolInfo(*$2->at(i));
        $$->push_back(symbol);
        table.insert($2->at(i));
      }
 }
```

Ilustración 9: Declaración de variables

Este fragmento de código define la regla de producción `declaration`, que describe cómo se estructura una declaración de variables en el lenguaje C.

`declaration_specifiers`: Como ya hemos comentado antes, especifica el tipo de las variables.

`init_declarator_list`: Especifica la lista de variables a ser declaradas. Devuelve un vector de *SymbolInfo*.

Se itera sobre cada *SymbolInfo* del vector de `init_declarator_list`, esta será nuestra lista de variables. Se asigna el tipo de la variable a cada símbolo a partir de la salida en `declaration_specifiers`. Se inserta cada variable en la tabla de símbolos y se crea una copia de cada variable para devolver a la siguiente regla.

```
declarator
    : pointer direct_declarator { $2->setIsPointer(true); $$ = $2; }
    | direct_declarator { $$ = $1; }
    ;

direct_declarator
    : IDENTIFIER        { $$ = $1; }
    | direct_declarator '[' assignment_expression ']' {
        if(!$1->isArray()){
            $1->setIsArray(true);
        }
        $1->addArrSize(($3->getSymbolName()));
        $$ = $1;
    }
    | direct_declarator '(' parameter_type_list ')' {
        $1->setParamList($3);
        $$ = $1;
    }
```

Ilustración 10: Sentencias de declaración en la gramática

Pueden existir distintos tipos de sentencias de declaración que nos podemos encontrar a lo largo del programa, pero para resumir presentaremos tres casos distintos. Estas reglas permiten especificar si un identificador es un puntero, una matriz, o una función, y también manejan listas de parámetros para funciones.

`pointer direct_declarator`: Si la sentencia de declaración incluye un puntero, se establece la propiedad *isPointer* true para el `direct_declarator` y se lleva a la siguiente regla.

`direct_declarator`: Esta regla define cómo se estructura una declaración que puede ser un identificador, una matriz o una función entre otros.

Si la declaración es una matriz se guarda el tamaño de esa matriz en la lista de tamaños que puede tener un array con la función *`addArrSize`* de *SimbolInfo*. Marcamos el símbolo como array en caso de que no esté ya marcado.

En el caso de que la sentencia de declaración sea una función, se asigna la lista de parámetros a la declaración con `setParamList`, permitiendo que esta contenga información sobre los parámetros de la función.

Si es un identificador simplemente lo asignamos a “$$”, es decir al resultado que devolverá la regla de producción.

```
declaration_specifiers
    : storage_class_specifier { $$ = $1; }
    | storage_class_specifier declaration_specifiers {
        std::ostringstream oss;
        oss << $1->getSymbolType() << "_" << $2->getSymbolType();
        $2->setSymbolType(oss.str());
        $$ = $2;
    }
    | type_specifier { $$ = $1; }
    | type_specifier declaration_specifiers {
        std::ostringstream oss;
        oss << $1->getSymbolType() << "_" << $2->getSymbolType();
        $2->setSymbolType(oss.str());
        $$ = $2;
    }
    | type_qualifier { $$ = $1; }
    | type_qualifier declaration_specifiers {
        std::ostringstream oss;
        oss << $1->getSymbolType() << "_" << $2->getSymbolType();
        $2->setSymbolType(oss.str());
        $$ = $2;
    }
    | function_specifier { $$ = $1; }
    | function_specifier declaration_specifiers {
        std::ostringstream oss;
        oss << $1->getSymbolType() << "_" << $2->getSymbolType();
        $2->setSymbolType(oss.str());
        $$ = $2;
    }
    ;
```

Ilustración 11: Reglas de Tipos y Especificadores

Este fragmento de código define la regla de producción `declaration_specifiers`, que especifica cómo se estructuran tipos o especificadores en el lenguaje C. Estos pueden incluir clases de almacenamiento, tipos, calificadores y especificadores de funciones, también pueden combinarse de varias maneras.

Para cada tipo de especificador individual (`storage_class_specifier`, `type_specifier`, `type_qualifier`, `function_specifier`), la acción simplemente asigna el valor del especificador a “$$”.

Cuando se combinan especificadores, se utiliza un *ostringstream* para concatenar los tipos de los especificadores con un guion bajo entre ellos. El tipo combinado se asigna al segundo especificador con `setSymbolType`, y este especificador combinado se devolverá para la siguiente línea de producción.

Este podría ser un ejemplo de cómo quedaría la definición de un especificador: “EXTERN_CONST_LONG_INT”

```
type_specifier
    : VOID          { $$ = new SymbolInfo("void", "VOID"); }
    | CHAR          { $$ = new SymbolInfo("char", "CHAR"); }
    | SHORT         { $$ = new SymbolInfo("short", "SHORT"); }
    | INT           { $$ = new SymbolInfo("int", "INT"); }
    | LONG          { $$ = new SymbolInfo("long", "LONG"); }
    | FLOAT         { $$ = new SymbolInfo("float", "FLOAT"); }
    | DOUBLE        { $$ = new SymbolInfo("double", "DOUBLE"); }
    | SIGNED        { $$ = new SymbolInfo("signed", "SIGNED"); }
    | UNSIGNED      { $$ = new SymbolInfo("unsigned", "UNSIGNED"); }
    | BOOL          { $$ = new SymbolInfo("bool", "BOOL"); }
    | COMPLEX       { $$ = new SymbolInfo("complex", "COMPLEX"); }
    | IMAGINARY     { $$ = new SymbolInfo("imaginary", "IMAGINARY"); }
    | struct_or_union_specifier  { $$ = $1; }
    | enum_specifier             { $$ = $1; }
    ;
```

Ilustración 12: Especificador de Tipos

Aquí se presentan las distintas formas en las que se puede especificar un tipo en el lenguaje C. Cada alternativa de la regla maneja un tipo específico o un especificador de tipo compuesto, como es en el caso de los *structs*, *unión* o *enum*.

```
struct_or_union_specifier
    : struct_or_union IDENTIFIER '{' struct_declaration_list '}'
    {
        $2->setIsStruct(true);
        $2->setVariableType($1->getSymbolType());
        $2->setParamList($4);
        $$ = $2;

    }
    | struct_or_union '{' struct_declaration_list '}'{
        $1->setIsStruct(true);
        $1->setParamList($3);
        $$ = $1;
    }
    | struct_or_union IDENTIFIER
    {
        $2->setIsStruct(true);
        $2->setVariableType($1->getSymbolType());
        $$ = $2;
    }
    ;
```

Ilustración 13: Tipo Struct o Union

La siguiente regla de producción `struct_or_union_specifier` maneja diferentes formas de especificar estructuras y uniones, incluyendo definiciones completas o parciales.

En el primer caso, cuando se define una estructura o unión con un nombre (IDENTIFICADOR) y una lista de declaraciones (`struct_declaration_list`), se manejan las definiciones completas. Marcamos el identificador como una estructura, asignamos el tipo (*struct* o *union*) al identificador y añadimos la lista de parámetros al identificador.

En el segundo caso la definición es parcial, no incluye nombre (IDENTIFICADOR). Devolvemos a la siguiente regla el símbolo (*struct* o *union*) con la lista de parámetros.

En el último caso la declaración vuelve a ser parcial, tenemos el nombre, pero no tenemos la lista de parámetros de la estructura. En este caso asignamos el tipo (*struct* o *unión*) al identificador y lo llevamos a la siguiente regla.

```
declaration
    : declaration_specifiers ';'{
        if($1->isStruct()){
            $$ = new vector<SymbolInfo*>();
            SymbolInfo* symbol = new SymbolInfo(*$1);
            $$->push_back(symbol);
            table.insert($1);
        }
    }
```

Ilustración 14: Declaración completa Struct o Union

Podemos añadir dentro de una variante de la regla de producción `declaration`, que en el caso de que los especificadores de una declaración sea una estructura y acabe en “;” esa estructura esta completa y la podemos añadir a la tabla de símbolos.

```
| declaration_specifiers init_declarator_list ';' {
  $$ = new vector<SymbolInfo*>();
  if($1->isStruct()){
    if($1->getParamList() != nullptr){
     for(std::vector<SymbolInfo*>::size_type i = 0; i < $2->size(); i++){
         $2->at(i)->setVariableType($1->getSymbolType());
         $2->at(i)->setIsStruct(true);
         $2->at(i)->setParamList($1->getParamList());
         SymbolInfo* symbol = new SymbolInfo(*$2->at(i));
         $$->push_back(symbol);
         table.insert($2->at(i));
         }
      }
    else {
     for(std::vector<SymbolInfo*>::size_type i = 0; i < $2->size(); i++){
          std::ostringstream oss = std::ostringstream();
          oss << $1->getVariableType() << "_" << $1->getSymbolName();
          $2->at(i)->setVariableType(oss.str());
          table.insert($2->at(i));
          $$->push_back($2->at(i));
         }
      }
   }
  else{
    for(std::vector<SymbolInfo*>::size_type i = 0; i < $2->size(); i++){
       $2->at(i)->setVariableType($1->getSymbolType());
       if(!$2->at(i)->isFunction()){
          SymbolInfo* symbol = new SymbolInfo(*$2->at(i));
          $$->push_back(symbol);
          table.insert($2->at(i));
        }
     }
   }
}
```

Ilustración 15: Declaración con estructuras

Para la declaración de variables tendremos que incluir el caso en el que el especificador de la declaración sea una estructura. Dentro de que sea una estructura existen dos variantes. La primera opción sería una estructura con parámetros en la declaración, es decir, por ejemplo:

```
typedef struct{
    unsigned char bl,gr,re;
} color;
```

Ilustración 16: Ejemplo estructura 1

Para este caso se itera sobre cada *SimbolInfo* en la lista de variables de `init_declarator_list, a cada variable` se asigna el tipo, se marca como estructura y se incluye la lista de parámetros de la estructura. Se crea una copia del objeto se inserta a la tabla de símbolos y se añade a "$$".

En el siguiente caso se pueden declarar múltiples variables de tipo *struct* o *enum*, lo cual es útil para la declaración de variables relacionadas que comparten el mismo tipo de datos. Este sería un ejemplo:

```
    struct timeval t1, t2;
```

Ilustración 17: Ejemplo estructura 2

El fragmento `struct timeval t1, t2;` declara dos variables `t1` y `t2` del tipo `struct timeval`. Para estos casos en los que la estructura no tiene una lista de parametros, se construye un tipo combinado y se asigna a cada variable; en el ejemplo expuesto sería "`STRUCT_timeval`". También como en el resto de los casos, cada símbolo se inserta en la tabla de símbolos y se añade al vector "$$".

4 Implementación y Funcionalidades del Traductor

4.1 Traducción de Directivas de OMPD a MPI

Llegamos a la parte crucial del proyecto, la traducción de código. En este apartado explicaremos detalladamente el proceso de traducción de directivas OMPD, de generación de código de inicialización para MPI, finalización, declaración de variables, etc. Para ello primero describiremos el flujo de traducción, usando el diagrama adjunto como referencia. Una vez descrito el proceso de traducción mostraremos como se gestionan estas conversiones utilizando la clase *MPIUtils* y sus componentes en el archivo *MPIUtils.h*.

Diagrama de flujo del Proceso de Traducción

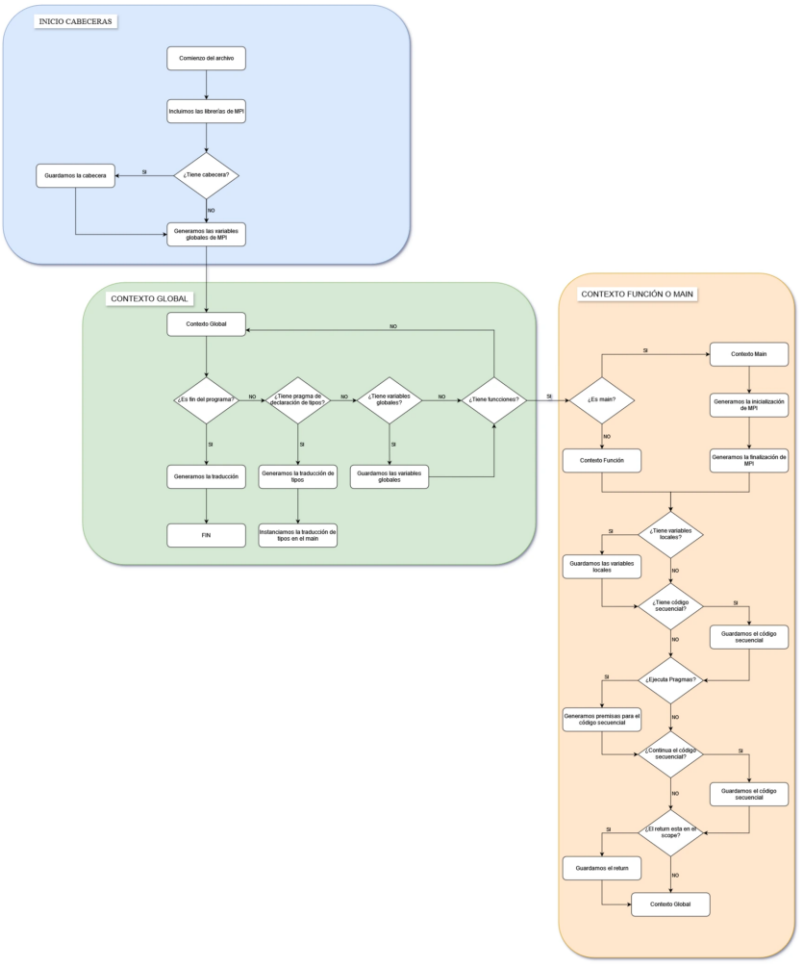

Ilustración 18: Diagrama de flujo del Proceso de Traducción

Dividiremos el diagrama de flujo en 3 secciones “Inicio Cabeceras”, “Contexto Global” y “Contexto Función o Main”. Esto nos ayudará a simplificar y organizar el proceso de traducción, asegurando que cada aspecto del código fuente se maneje de manera eficiente y estructurada.

Inicio Cabeceras

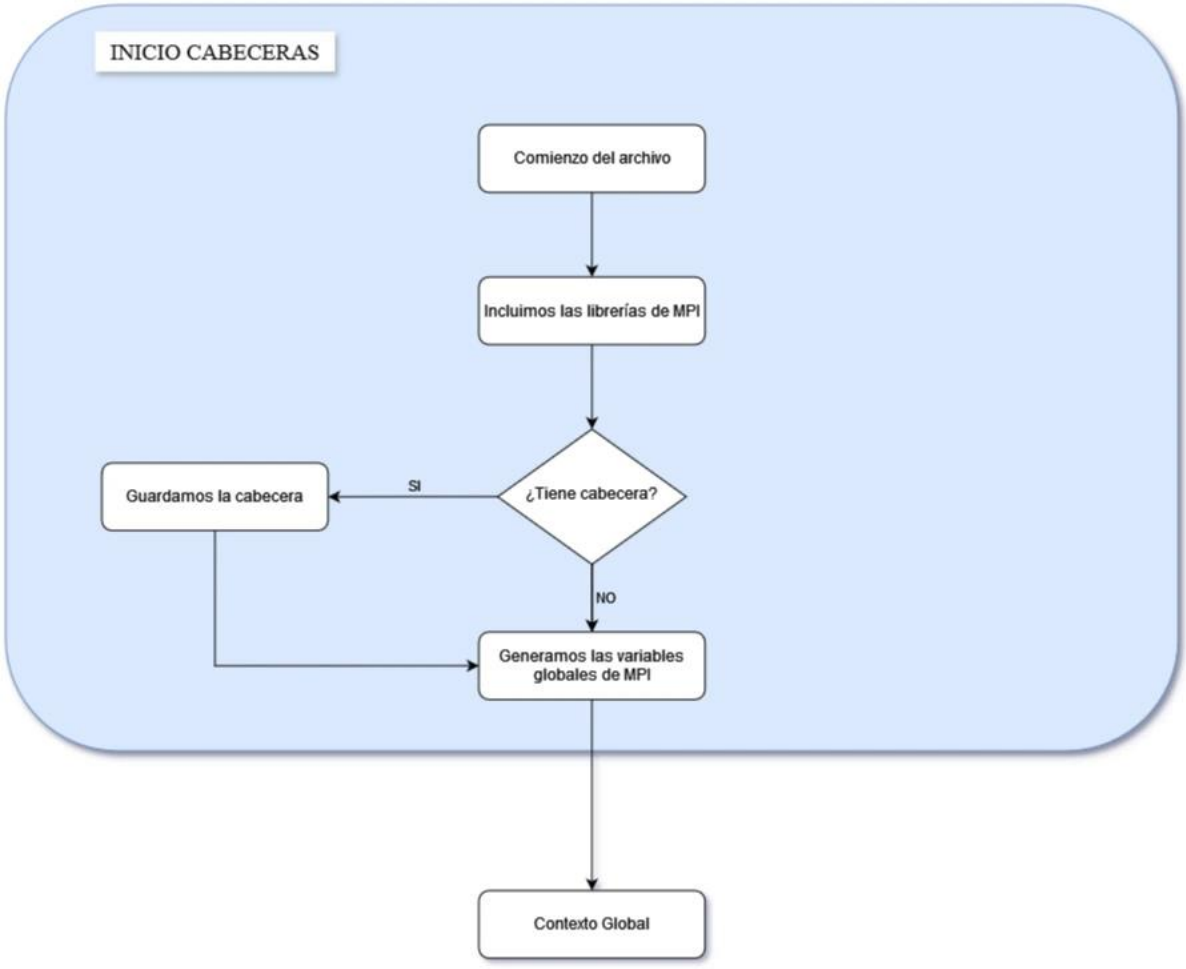

Ilustración 19: Flujo Inicio Cabeceras

Esta sección del diagrama se encarga de las operaciones iniciales necesarias para preparar el entorno de MPI en el archivo de código. El flujo comienza con la inclusión de las cabeceras necesarias para MPI (*assert.h*, *mpi.h*). En este punto, el traductor verifica si el archivo ya contiene una cabecera. Si existe, se guarda y se incluye en la cabecera del fichero de traducción. A continuación, se generan las variables globales necesarias para el uso de MPI, tales como `__taskid` y `__numprocs`, asegurando que el código resultante tenga toda la información necesaria para la ejecución en un entorno MPI.

Contexto Global

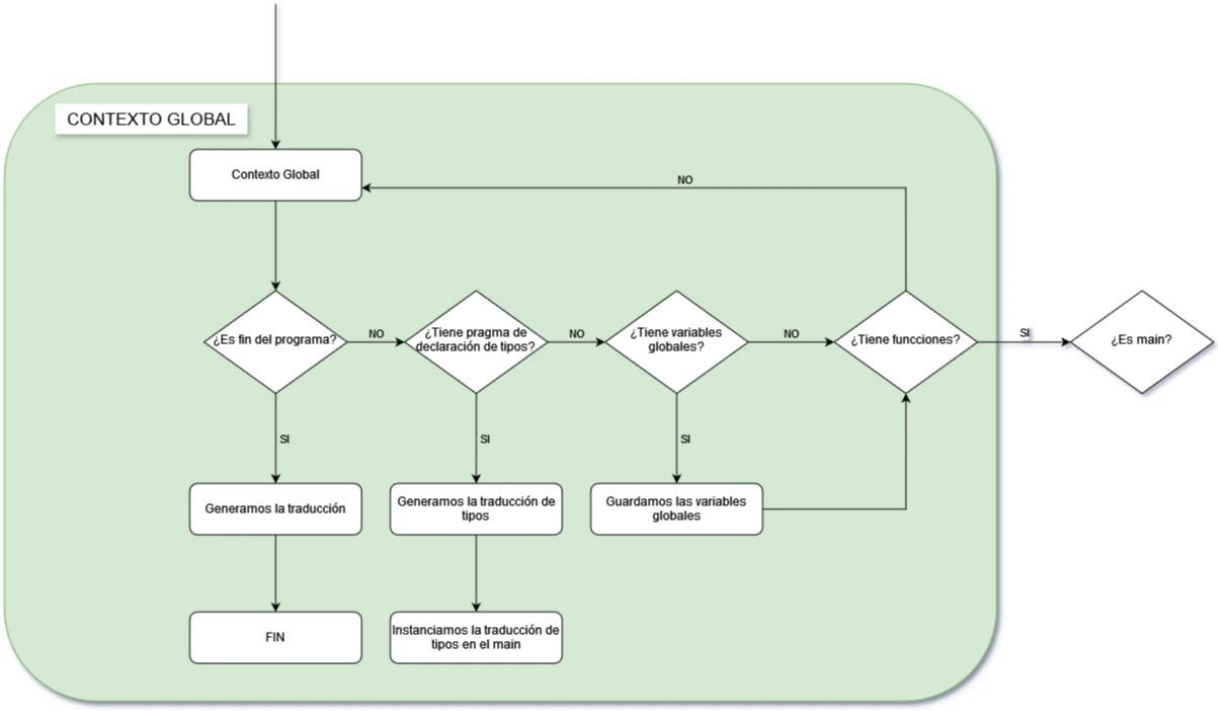

Ilustración 20: Diagrama Contexto Global

En esta sección, se analiza el contexto global del archivo de código, lo que incluye las definiciones de variables globales, funciones y los pragmas de traducción de tipos. El flujo de trabajo se divide en varios pasos clave:

1. **Detección del Contexto Global**: Aquí, el traductor determina que se encuentra en el contexto global del programa.
2. **Fin del archivo**: Si nos encontramos ante el final del archivo detenemos la traducción y generamos el archivo de traducción *generated.c*.
3. **Pragmas de Traducción de Tipos**: Si se detectan pragmas específicos que definen tipos de datos para OpenMPD, estos se traducen a los tipos de datos equivalentes en MPI. Esto implica generar declaraciones de tipos de datos MPI y las funciones necesarias para manejarlos. También tendremos que generar la llamada a la generación de tipos en la función *main*.
4. **Variables Globales**: Las variables globales encontradas en el código original se guardan en la traducción.
5. **Funciones Globales**: Si hay funciones definidas en el contexto global, el traductor prepara estas funciones para su inclusión en el flujo de MPI. Esto incluye la identificación y manejo de la función *main*, que es crucial para la inicialización y finalización de MPI.

Contexto Función o Main

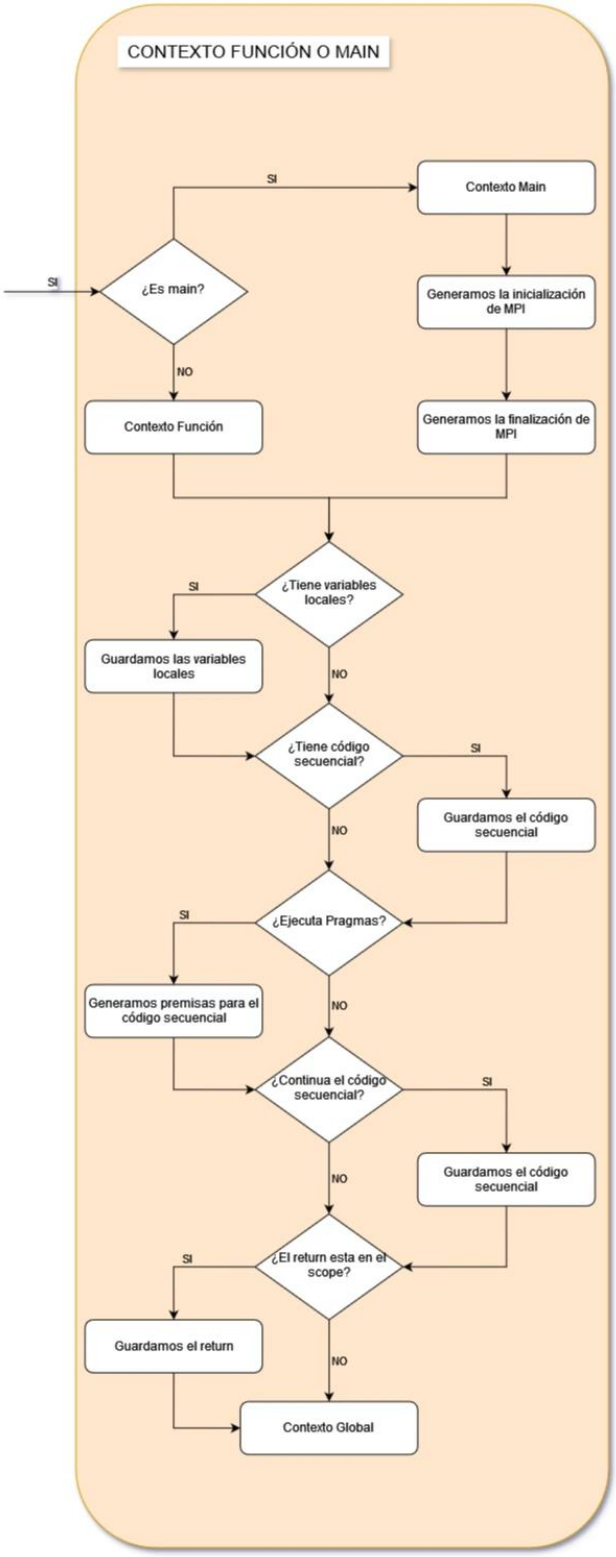

Ilustración 21: Diagrama Función o Main

Esta sección maneja la traducción de las funciones y la función principal *main* del código. El flujo de trabajo incluye:

1. **Identificación del Contexto Main**: Si el traductor se encuentra en la función *main*, genera el código necesario para inicializar y finalizar MPI. Esto incluye las llamadas a *MPI_Init*, *MPI_Comm_size*, *MPI_Comm_rank* al inicio y *MPI_Finalize* al final del programa.
2. **Traducción de Funciones**: Si el contexto actual es una función, el traductor maneja las variables locales, cualquier código secuencial presente, y las directivas pragma. El flujo asegura que se generen los permisos y estructuras necesarias para que el código secuencial funcione correctamente dentro del entorno MPI.
3. **Manejo de Return**: Si se encuentra una sentencia *return*, esta se procesa adecuadamente para asegurar que la función se complete correctamente dentro del marco de MPI.

Al dividir el diagrama de flujo en estas tres secciones, logramos una traducción organizada del código. Cada sección se enfoca en un aspecto específico del programa. A través de este proceso, se mantiene la funcionalidad del código original mientras se aprovechan las capacidades de MPI.

Descripción de MPIUtils.h

El archivo *MPIUtils.h* proporciona una serie de clases y métodos diseñados para facilitar la generación de código MPI. Las clases principales en este archivo son *Funcion*, *MainFuncion* y *MPIUtils*.

Clase Función

La clase *Funcion* maneja la traducción de funciones individuales dentro del código fuente.

```
class Funcion
{

private:
    std::ostringstream variables, ini_seq_pre, c_seq_pre, end_seq_pre,
    pragmas, ini_seq_pos, c_seq_pos, end_seq_pos, ret;

public:
    void write_MPI_sec_pre();
    void write_MPI_seq_pos();
    void insert_MPI_Funcion(std::string line, int state);
    std::string print();
    ~Funcion();
}
```

Ilustración 22: Clase Funcion

Atributos: La clase utiliza varios buffers en memoria (*ostringstream*) para almacenar diferentes partes del código de una función, incluyendo variables (*variables*), codigo secuencial previo a la declaración de pragmas (*c_seq_pre*), declaración de pragmas (*pragmas*), codigo secuencial después de los pragmas *(c_seq_pos)*, el *return* de la función (*ret*), y las condiciones que envolverán el código secuencial en caso de ejecución de pragmas (*ini_seq_pre*, *end_seq_pre*, *ini_seq_pos*, *end_seq_pos*).

Métodos:

- `write_MPI_sec_pre()`, `write_MPI_seq_pos()`: Estas funciones generan el código necesario para que el código secuencial previo y posterior a un pragma se ejecuten solo en el nodo maestro (`__taskid == 0`), imitando el comportamiento de las directivas de tareas de OpenMPD.

```
    void write_MPI_sec_pre()
    {
        if (ini_seq_pre.str().empty() && end_seq_pre.str().empty())
        {
            ini_seq_pre << "\nif (__taskid == 0) {\n";
            end_seq_pre << "}\n";
        }
    }
    void write_MPI_seq_pos()
    {
        if (ini_seq_pos.str().empty() && end_seq_pos.str().empty())
        {
            ini_seq_pos << "\nif (__taskid == 0) {\n";
            end_seq_pos << "}\n";
        }
    }
```

Ilustración 23: Funcion pre y pos secuencial

- `insert_MPI_Funcion(std::string line, int state)`: Inserta una línea de código en la sección correspondiente de una función, dependiendo del estado proporcionado. La función toma dos parámetros: una cadena *line* que contiene la línea de código a insertar y un entero *state* que indica en qué parte de la función se debe insertar esa línea.

```
void insert_MPI_Funcion(std::string line, int state)
{
    switch (state)
    {
    case 1:
        variables << line;
        break;
    case 2:
        c_seq_pre << line;
        break;
    case 3:
        write_MPI_sec_pre();
        pragmas << line;
        write_MPI_seq_pos();
        break;
    case 4:
        c_seq_pos << line;
        break;
    case 5:
        ret << line;
        break;
    }
}
```

Ilustración 24: Funcion Insert

Si *state* es 1, la línea de código se inserta en el buffer de `variables`. Esto significa que la línea corresponde a una declaración de variable local dentro de la función.

Si *state* es 2, la línea de código se inserta en el buffer `c_seq_pre`. Esto indica que la línea pertenece al código secuencial que se ejecuta antes de las directivas paralelas dentro de la función.

Si *state* es 3, la línea de código se inserta en el buffer `pragmas`. Esto generalmente corresponde a directivas pragmas que controlan el comportamiento paralelo. También se llaman a las funciones `write_MPI_sec_pre()` y `write_MPI_seq_pos()`.

Si *state* es 4, la línea de código se inserta en el buffer `c_seq_pos`. Esto indica que la línea corresponde al código secuencial que se ejecuta después de las directivas paralelas dentro de la función.

Si *state* es 5, la línea de código se inserta en el buffer `ret`. Esto significa que la línea es una sentencia de retorno (*return*) de la función.

- `print()`: Devuelve el código completo de la función como una cadena de caracteres.

```
    std::string print()
    {
        return variables.str() + ini_seq_pre.str() + c_seq_pre.str() +
        end_seq_pre.str() + pragmas.str() + ini_seq_pos.str() +
        c_seq_pos.str() + end_seq_pos.str() + ret.str();
    }
```

Ilustración 25: Funcion Print

Clase MainFuncion

La clase *MainFuncion* maneja la traducción de la función principal (*main*) del programa.

```
class MainFuncion
{

private:
    std::ostringstream variables, inic, inic_type_def, ini_seq_pre,
    c_seq_pre, end_seq_pre, pragmas, ini_seq_pos, c_seq_pos, end_seq_pos,
    c_fi, ret;

public:
    MainFuncion();
    void write_type_def();
    void write_MPI_sec_pre();
    void write_MPI_seq_pos();
    void insert_MPI_Main(std::string line, int state);
    std::string print();
    ~MainFuncion();
}
```

Ilustración 26: Clase MainFuncion

Como se puede apreciar, esta clase maneja prácticamente las mismas variables y funciones que la clase *Funcion*. Sin embargo, maneja algunas diferencias importantes que comentaremos a continuación.

Atributos: La clase también utiliza varios buffers en memoria (*ostringstream*) para almacenar diferentes partes del código de la función *main*. A diferencia de la clase *Función* se incluye un buffer que guardara las inicializaciones necesarias para MPI, como *MPI_Init*, *MPI_Comm_size*, y *MPI_Comm_rank* (*inic*).

También se incluye un buffer que guardara la inicialización de tipos personalizados de MPI en caso de que se hayan creado (*inic_type_def*). Y por último se incluirá un buffer para las finalizaciones necesarias para MPI, como *MPI_Finalize* (*c_fi*).

Métodos:

- Constructor `MainFuncion()`: Guarda el código de inicialización y finalización de MPI.

```
MainFuncion()
{
    inic << "MPI_Init(&argc, &argv);\n";
    inic << "MPI_Comm_size(MPI_COMM_WORLD,&__numprocs);\n";
    inic << "MPI_Comm_rank(MPI_COMM_WORLD,&__taskid);\n\n";

    c_fi << "MPI_Finalize();\n";
}
```

Ilustración 27: Constructor MainFuncion

- `write_type_def()`: Guarda la inicialización de tipos personalizados de MPI.

```
void write_type_def()
{
    if (inic_type_def.str().empty())
        inic_type_def << "\nDeclare_MPI_Types();\n";
}
```

Ilustración 28: MainFuncion Inicialización de Tipos

- `print()`: Devuelve el código completo traducido del *main* como una cadena de caracteres.

```
std::string print()
{
    return variables.str() + inic.str() + inic_type_def.str() +
    ini_seq_pre.str() + c_seq_pre.str() + end_seq_pre.str() +
    pragmas.str() + ini_seq_pos.str() + c_seq_pos.str() +
    end_seq_pos.str() + c_fi.str() + ret.str();
}
```

Ilustración 29: MainFuncion print

Clase MPIUtils

La clase *MPIUtils* gestiona la generación global del código MPI, incluyendo la cabecera, las variables globales, las funciones y el *main*.

```
class MPIUtils
{

private:
    std::ostringstream includes, var_glob;
    std::vector<Funcion *> *funciones = new std::vector<Funcion *>();
    MainFuncion *mainFuncion = new MainFuncion();

    std::ostringstream line;

public:
    void write_MPI_Type_struct(SymbolInfo *symbol);
    void write_MPI_header();
    void write_MPI_new_func();
    void insert_MPI_token(std::string token, int level, int state);
    void insert_MPI_buffer_line(int level, int state);
    void insert_MPI(std::string line, int level, int state);
    void insert_MPI_Global(std::string line, int state);
    void generate_MPI_all();
    ~MPIUtils();
}
```

Ilustración 30: MPIUtils clase

La clase *MPIUtils* es la principal encargada de manejar la traducción global del código fuente. Proporciona las funcionalidades generales necesarias para gestionar la creación de tipos de datos MPI, la inserción de código MPI en las secciones adecuadas del programa, y la generación del archivo final traducido.

Atributos

- `includes`: Este buffer almacena las inclusiones de cabeceras necesarias para MPI, tales como `#include <mpi.h>`. Estas inclusiones son esenciales para que cualquier parte del programa traducido pueda utilizar las funcionalidades de MPI.
- *var*`_glob`: Almacena las declaraciones de variables globales necesarias para MPI. Por ejemplo, variables como `__taskid` y `__numprocs` se declaran aquí. Estas variables son fundamentales para gestionar la identificación de tareas y el número de procesos en MPI.
- `funciones`: Es un vector que contiene punteros a instancias de la clase *Funcion*. Este vector almacena todas las funciones del programa traducidas a MPI. Cada *Funcion* maneja una función individual del programa original.
- `mainFuncion`: Un puntero a la instancia de *MainFuncion*. Esta instancia maneja la traducción de la función *main* del programa, que es crucial para inicializar y finalizar el entorno MPI.

- `line`: Este buffer se utiliza para almacenar temporalmente una línea de código antes de determinar en qué sección del programa debe ser insertada. Facilita la manipulación y el procesamiento de líneas de código antes de su ubicación final.

Métodos:

- `write_MPI_Type_struct(SymbolInfo *symbol)`: Este método genera el código necesario para definir un nuevo tipo de dato MPI basado en una estructura definida en el programa original. Utiliza la información de *SymbolInfo* para crear y registrar el tipo de dato en MPI. Mas adelante haremos una explicación detallada sobre esta función y el papel que juega en la traducción de tipos.

```
void write_MPI_Type_struct(SymbolInfo *symbol)
{
    insert_MPI_buffer_line(0, 1);
    var_glob << "\n\nMPI_Datatype MPI" << symbol->getSymbolName() << "_t;\n\n";
    var_glob << "void __Declare_MPI_Type_" << symbol->getSymbolName() << "() {\n";
    var_glob << "    int blocklengths[" << symbol->getParamList()->size() << "];\n";
    var_glob << "    MPI_Datatype old_types[" << symbol->getParamList()->size() << "];\n";
    var_glob << "    MPI_Aint disp[" << symbol->getParamList()->size() << "];\n";
    var_glob << "    MPI_Aint lb;\n";
    var_glob << "    MPI_Aint extent;\n";
    for (std::vector<SymbolInfo *>::size_type i = 0; i < symbol->getParamList()->size(); i++)
    {
        var_glob << "    blocklengths[" << i << "] = 1;\n";
    }
    for (std::vector<SymbolInfo *>::size_type i = 0; i < symbol->getParamList()->size(); i++)
    {
        var_glob << "    old_types[" << i << "] = MPI_" << symbol->getParamList()->at(i)->getVariableType() << ";\n";
    }
    for (std::vector<SymbolInfo *>::size_type i = 0; i < symbol->getParamList()->size(); i++)
    {
        var_glob << "    MPI_Type_get_extent(MPI_" << symbol->getParamList()->at(i)->getVariableType() << ", &lb, &extent);\n";
        if (i == 0)
            var_glob << "    disp[" << i << "] = lb;\n";
        else
            var_glob << "    disp[" << i << "] = disp[" << i - 1 << "] + extent;\n";
    }
    var_glob << "    MPI_Type_create_struct(" << symbol->getParamList()->size() << ", blocklengths, disp, old_types, &MPI" << symbol->getSymbolName() << "_t);\n";
    var_glob << "    MPI_Type_commit(&MPI" << symbol->getSymbolName() << "_t);\n";
    var_glob << "}\n\n";
    var_glob << "void Declare_MPI_Types() {\n";
    var_glob << "    __Declare_MPI_Type_" << symbol->getSymbolName() << "();\n";
    var_glob << "    return;\n";
    var_glob << "}\n";

    mainFuncion->write_type_def();
}
```

Ilustración 31: MPIUtils insert_MPI_buffer_line

- `write_MPI_header()`: Escribe las cabeceras necesarias para MPI en el buffer de `includes`. Asegura que las librerías necesarias de MPI estén disponibles en el código traducido. Además, incluimos las variables globales `__taskid` y `__numprocs` en el buffer `var_glob`.

```
void write_MPI_header()
{
    includes << "#include <assert.h>\n";
    includes << "#include <mpi.h>\n";
    var_glob << "\nint __taskid = -1, __numprocs = -1;\n\n";
}
```

Ilustración 32: MPIUtils write_MPI_header

- `write_MPI_new_func()`: Crea y almacena una nueva instancia de *Funcion* en el vector funciones. Esto prepara el traductor para manejar una nueva función del programa original.

```
void write_MPI_new_func()
{
    Funcion *newFuncion = new Funcion();
    funciones->push_back(newFuncion);
}
```

Ilustración 33: MPIUtils write_MPI_new_func

- `insert_MPI_token(std::string token, int level, int state)`: Inserta un token de código a traducir en la sección correspondiente, dependiendo del nivel y el estado especificado. Utiliza `line` para almacenar el token temporalmente hasta determinar su ubicación final.

```
void insert_MPI_token(std::string token, int level, int state)
{
    if (token == "\n")
    {
        line << token;
        insert_MPI(line.str(), level, state);
        line = std::ostringstream();
    }
    else
    {
        line << token;
    }
}
```

Ilustración 34: MPIUtils insert_MPI_token

- `insert_MPI_buffer_line(int level, int state)`: Inserta la línea de código almacenada en `line` en la sección correspondiente del programa, según el nivel y el estado especificado. Luego, `line` se reinicia para almacenar una nueva línea.

```
void insert_MPI_buffer_line(int level, int state)
{
    insert_MPI(line.str(), level, state);
    line = std::ostringstream();
}
```

Ilustración 35: MPIUtils insert_MPI_buffer_line

- `insert_MPI(std::string line, int level, int state)`: Inserta una línea de código (*line*) directamente en la sección correspondiente, dependiendo del nivel y el estado especificado.

```
void insert_MPI(std::string line, int level, int state)
{
    switch (level)
    {
    case 0:
        insert_MPI_Global(line, state);
        break;
    case 1:
        if (!funciones->empty())
        {
            Funcion *lastFuncion = funciones->back();
            lastFuncion->insert_MPI_Funcion(line, state);
        }
        break;
    case 2:
        mainFuncion->insert_MPI_Main(line, state);
        break;
    }
}
```

Ilustración 36: MPIUtils insert_MPI

El parametro *level* es un entero que indica el contexto en donde se debe insertar la línea de código. Los niveles posibles son: Contexto Global (0), Contexto de una función (1), Contexto del *main* (2).

Si *level* es 0, la línea de código se inserta en el contexto global llamando a la función `insert_MPI_Global(line, state)`. Esta función maneja la inserción de *includes* en la cabeceras y declaraciones de variables globales.

Si *level* es 1, la línea de código se inserta en el contexto de una función. Primero, se verifica si el vector *funciones* no está vacío, asegurando que haya al menos una función en la lista. Si hay funciones en la lista, se obtiene un puntero a la última función (*lastFuncion*) utilizando `funciones->back()`. Luego, se llama al método `insert_MPI_Funcion(line, state)` de *lastFuncion*, que inserta la línea de código en la sección correspondiente de esa función según el *state* proporcionado.

Si *level* es 2, la línea de código se inserta en el contexto del *main*. Se llama al método `insert_MPI_Main(line, state)` de `mainFuncion`, que inserta la línea de código en la sección correspondiente del *main* según el *state* proporcionado.

- `insert_MPI_Global(std::string line, int state)`: Inserta una línea de código en el contexto global del programa. Dependiendo del estado, puede ser un *include* o una declaración de variable global.

```
    void insert_MPI_Global(std::string line, int state)
    {
        switch (state)
        {
        case 0:
            includes << line;
            break;
        case 1:
            var_glob << line;
            break;
        }
    }
```

Ilustración 37: MPIUtils insert_MPI_Global

- `generate_MPI_all()`: Genera el archivo final de código traducido, combinando todas las partes procesadas. Escribe el contenido de includes, var_glob, todas las funciones almacenadas en funciones, y por último la función *main* en *generatedFile*.

```
void generate_MPI_all()
{
generatedFile << includes.str() << var_glob.str();
for (std::vector<Funcion *>::size_type i = 0; i < funciones->size(); i++)
    {
        generatedFile << funciones->at(i)->print();
    }
    generatedFile << mainFuncion->print();
}
```

Ilustración 38: MPIUtils generate_MPI_all

- `Destructor ~MPIUtils()`: Libera los recursos utilizados por la clase. Elimina la instancia de *MainFuncion* y todas las instancias de *Funcion* almacenadas en el vector `funciones`.

```
    ~MPIUtils()
    {
        delete mainFuncion;
        for (auto funcion : *funciones)
        {
            delete funcion;
        }
        delete funciones;
    }
```

Ilustración 39: Destructor ~MPIUtils

La clase *MPIUtils* proporciona una infraestructura organizada para la traducción de directivas de OMPD a MPI. Maneja la complejidad de la traducción y garantiza que todas las secciones del programa, desde las cabeceras hasta las funciones individuales y el bloque *main*, se traduzcan correctamente. Esta clase es esencial para mantener la coherencia y funcionalidad del código original en el entorno distribuido de MPI.

4.2 Traductor en el Parser y Scanner

Para el correcto funcionamiento del traductor tendremos que integrar las funciones de traducción de *MPIUtils.h* con el proceso de análisis léxico y sintáctico. Además, describiremos las variables globales utilizadas para controlar el flujo de la traducción. En primer lugar describiremos estas variables y como los declararemos en ambos archivos *C99-scanner.lex* y *C99-parser.yacc*.

Variables Globales

```
%{
#include "MPIUtils.h"

// Declaración de un objeto MPIUtils
extern MPIUtils mpi_utils;

// Declaración de variables globales
extern bool declarePragma, otherPragma;
int state = 0, level = 0;
%}
```

Ilustración 40: Declaración de variables globales Parser

```
%{
#include "MPIUtils.h"

// Declaración de un objeto MPIUtils
MPIUtils mpi_utils;

// Declaración de variables globales
bool declarePragma, otherPragma;
Extern int state = 0, level = 0;
%}
```

Ilustración 41: Declaración de variables globales Scanner

`level`: Indica el contexto; global (0), función (1), main (2) en el que se está insertando el código.

`state`: Indica el estado actual del traductor. En contexto de funciones y *main*: inserción de variables (1), inserción de código secuencial anterior a pragmas (2), inserción de pragmas (3), inserción de código secuencial posterior a pragmas (4), inserción de sentencia *return* (5). En el contexto global: inserción de cabecera (0), inserción de variables globales (1).

`declarePragma`: Marca si se ha encontrado una directiva pragma de declaración de tipos de datos (true o false).

`otherPragma`: Marca si se ha encontrado cualquier otra directiva pragma (true o false).

Traductor en el Scanner

Como ya sabemos el *scanner* es responsable de identificar tokens en el código fuente y enviarlos al *parser*. Cada vez que hacemos una lectura de un token el *scanner* hace una llamada a la función `count()`. Podemos incluir en la función `count()` la inserción de tokens para la generación de código y así permitir una lectura completa del archivo fuente.

```
void count(void)
{
    int i;

    for (i = 0; yytext[i] != '\0'; i++)
        if (yytext[i] == '\n'){
            line_count++;
            column = 0;
        }
        else if (yytext[i] == '\t')
            column += 8 - (column % 8);
        else
            column++;

    mpi_utils.insert_MPI_token(yytext, level, state);

    ECHO;
}
```

Ilustración 42: Insertar Tokens en el Scanner

El siguiente fragmento de código en el scanner se encarga de procesar líneas que comienzan con el carácter "#", como las directivas `#pragma` e `#include`. Este código detecta estas directivas y hace uso del objeto `mpi_utils` para insertar las líneas de código en las secciones correspondientes del programa.

```
"#"             {
                  char * line = get_pragma();
                  char * pragma = strstr(line, "pragma");
                  char * include = strstr(line, "include");

                  if (pragma != NULL) {
                   if (level > 0) { state = 3; }
                   if (strstr(pragma, "end") != NULL){
                       declarePragma = false;
                   }
                   else if (strstr(pragma, "declare") != NULL){
                       declarePragma = true;
                   }

                   else{
                    otherPragma = true;
                    mpi_utils.insert_MPI("// pragma aqui\n", level, state);
                   }

                   parseOpenMP(pragma+7, NULL);
                  }
                  else {
                     std::ostringstream oss;
                     oss << "#" << line << endl;
                     if ( level > 0 && state == 3){
                         state = 4;
                     }
                     if(include != NULL){
                         mpi_utils.insert_MPI(oss.str(), 0, 0);
                     }
                     else{
                         mpi_utils.insert_MPI(oss.str(), level, state);
                     }
                     line_count++;
                     column = 0;
                  }
                }
```

Ilustración 43: Inserción y traducción de pragmas y otros

Primero obtenemos la línea completa que comienza con “#” utilizando la función `get_pragma`.

Si *linea* incluye la subcadena “pragma” nos encontramos ante una declaración de pragmas. En el caso en que estemos fuera del contexto global (0) cambiaremos el estado a declaración de pragmas (3). Si *linea* incluye la subcadena “end” significa que estamos en el final de la declaración de tipos y

desactivamos *declarePragma.* Si *linea* incluye la subcadena "declare" significa que estamos en el principio de la declaración de tipos y activaremos *declarePragma.* Si nos encontramos ante otro tipo de pragma simplemente insertamos un comentario "// pragma aquí" como indicador en la traducción y activamos *otherPragma*. Finalmente, se llama a *parseOpenMP* para procesar la directiva OpenMP.

En el caso de que la línea no incluya "pragma" lo trataremos como si fuera código secuencial. Si nos encontramos fuera del contexto global (>0) y el estado actual es declaración de pragmas (3), cambiaremos el estado a "código secuencial después de pragma" (4) y lo insertamos a la traducción mediante *insert_MPI* de *mpi_utils*. En caso de que sea una sentencia "#include" lo insertaremos en la cabecera de la traducción.

Traductor en el Parser

El *parser* analizará la estructura del programa y según se procesen nuevas directivas se cambiará el contexto en el que se encuentra el programa (*nivel*) y el estado (*state*). Uno de los primeros pasos en este proceso es asegurar que las librerías de MPI se añadan al comienzo de la traducción.

```
translation_unit
    : { mpi_utils.write_MPI_header(); state = 1; } external_declaration
    | translation_unit external_declaration
    ;
```

Ilustración 44: Comienzo e inserción de librerías MPI

Como ya hemos visto, está llamada a `mpi_utils.write_MPI_header()` agrega las inclusiones necesarias para MPI al comienzo del archivo traducido. A continuación, cambiaremos el estado a inserción de variables (1).

```
function_definition
    : declaration_specifiers declarator {
        $2->setIsFunction(true);
        $2->setVariableType($1->getSymbolType());
        table.insert($2);
        table.enterScope();
        table.setIsScopeReturn(true);
        if ($2->getParamList() != nullptr) {
        for(std::vector<SymbolInfo*>::size_type i = 0; i < $2->getParamList()->size(); i++){
                SymbolInfo* symbol = new SymbolInfo(*$2->getParamList()->at(i));
                table.insert(symbol);
            }
        }
    } compound_statement {
        $2->setIsDefined(true);
        table.exitScope();
        mpi_utils.insert_MPI_buffer_line(level, state);
        level = 0;
        state = 1;
        if(otherPragma){
            SymbolInfo* symbol = table.getSymbolInfo($2->getSymbolName());
            symbol->setHasPragma(true);
            otherPragma = false;
        }
    }
    ;
```

Ilustración 45: Declaración de funciones con traducción

El fragmento de código corresponde a la regla `function_definition`, que define cómo se traduce una definición de función en el análisis sintáctico. Tendremos que modificar esta regla para que maneje tanto la inserción de la función en la tabla de símbolos como la manipulación de los niveles y estados necesarios para la correcta generación de código MPI.

Las sentencias que hemos añadido a esta regla son:

`setIsScopeReturn(true)`: Indica que la tabla *scope* actual puede tener una sentencia de retorno. Esto nos ayudara a encontrar el final del programa en caso de múltiples *returns*.

Después de alcanzar el final de la función procesaremos la línea que haya quedado en el buffer de *MPIUtils* con *insert_MPI_buffer_line* y cambiamos el contexto a global (0) y el estado a inserción de variables (1). Si la función tiene ejecución de pragmas (*otherPragma* es true) lo marcamos en la tabla de símbolos con `symbol->setHasPragma(true);`.

```
direct_declarator
    : direct_declarator '(' parameter_type_list {
        state = 1;
        if($1->getSymbolName() == "main"){
            level = 2;
        }
        else{
            mpi_utils.write_MPI_new_func();
            level = 1;
        }

    } ')' {
        $1->setParamList($3);
        $1->setIsFunction(true);
        $$ = $1;

    }
```

Ilustración 46: Declaración de parametros y traducción

Antes de procesar la función y después de leer los parámetros de la función cambiaremos el estado a declaración de variables (1) y dependiendo de si la función es *main* o no cambiaremos *level* a contexto main (2) o contexto función (1).

```
block_item
    : declaration
    | { statement_MPI(); } statement
    ;

void statement_MPI(){
    if (state == 1){
        state = 2;
    }
    else if ( state == 3){
        state = 4;
    }
}
```

Ilustración 47: Statements en traducción MPI

Conocemos el flujo de traducción, si estamos dentro de una función y el estado actual es “declaración de variables” (0) y nos encontramos con código secuencial (*statements*) cambiaremos el estado a “código secuencial antes de pragma” (2). Si el estado actual es “ejecución de pragmas” (3) y no encontramos con código secuencial (*statements*) cambiaremos el estado a “código secuencial posterior a pragma” (4).

```
jump_statement
    : GOTO IDENTIFIER ';'
    | CONTINUE ';'
    | BREAK ';'
    | RETURN { if (table.getIsScopeReturn()) { state = 5; };  } ';'
    | RETURN { if (table.getIsScopeReturn()) { state = 5; };  }
expression ';'
    ;
```

Ilustración 48: Traducción y Returns

En el caso de que nos encontremos una sentencia *return* tendremos que comprobar si estamos en el *scope* de la función. Si ese es el caso significa que es el final del programa y cambiamos el estado a "sentencia return" (5). De esta forma la sentencia *return* quedará fuera del código secuencial posterior a la ejecución de pragmas.

```
postfix_expression
    : postfix_expression '(' argument_expression_list ')' {
        SymbolInfo* symbol = table.getSymbolInfo($1->getSymbolName());
        if(symbol != nullptr && symbol->getHasPragma()){
            state = 3;
            otherPragma = true;
        }
      };
```

Puede ocurrir el caso en que nos encontremos una llamada a una función externa que ejecuta pragmas, en ese caso cambiaremos el estado a "ejecución de pragmas" (3) y activaremos *otherPragma*. Nos ayudaremos de la función `symbol->getHasPragma()` para comprobar si la función ejecuta pragmas.

4.3 Gestión y Traducción de Tipos de Datos Definidos por el Usuario

Como he explicado al comienzo de este proyecto, dentro de los objetivos a cumplir estaría la traducción de tipos definidos por el usuario. En específico sería la traducción de los tipos definidos como *struct*. Un ejemplo sería el siguiente:

```
#pragma omp declare cluster
typedef struct{
    unsigned char bl,gr,re;
} color;
#pragma omp end declare cluster
```

Ilustración 49: Declaración de tipo OMPD

Para el comienzo de la declaración de un tipo de dato definido por el usuario en OMPD se escribe al comienzo de la sentencia `#pragma omp declare cluster`. En este caso, indica que el siguiente *typedef* define un tipo de dato que se utilizará en clústeres paralelos.

El *typedef* crea un alias "color" para esta estructura, permitiendo su uso en el código como un tipo de dato definido por el usuario.

Para marcar el final de la declaración del tipo de dato se escribe la sentencia `#pragma omp end declare cluster`. Esta sentencia indica al compilador que la definición del tipo "color" ha terminado y que debe ser tratado como un tipo de dato MPI para fines de comunicación y sincronización en memoria distribuida.

En primer lugar, tendremos que procesar los tipos definidos por el usuario sin que genere errores en el compilador, para ello tendremos que modificar el *scanner* y *parser* de nuevo.

Una vez hecho esto tendremos que generar la traducción de los tipos definidos por usuario.

Definición de Tipos con Typedef

Dentro del Scanner crearemos una nueva palabra reservada para los tipos definidos por el usuario. Dentro de [IDENTIFICADOR] si el valor del identificador es un tipo definido por usuario devolveremos el token [USER_DEFINED] con el valor del identificador:

```
{L}({L}|{D})*       {
    count();
    SymbolInfo *s = table.getSymbolInfo(yytext);
    if(s != NULL && s->getSymIsType()){
        yylval.sym = new SymbolInfo(yytext, (char *) yytext);
        return USER_DEFINED;
    }
    else{
        SymbolInfo *s = new SymbolInfo(yytext, (char *)"IDENTIFIER");
        yylval.sym = s;
        return IDENTIFIER;
    }
}
```

Ilustración 50: USER_DEFINED token

Dentro del Parser añadiremos el tipo [USER_DEFINED] a la regla `type_specifier` de la gramática:

```
type_specifier
    : VOID          { $$ = new SymbolInfo("void", "VOID"); }
    | CHAR          { $$ = new SymbolInfo("char", "CHAR"); }
    | SHORT         { $$ = new SymbolInfo("short", "SHORT"); }
    | INT           { $$ = new SymbolInfo("int", "INT"); }
    | LONG          { $$ = new SymbolInfo("long", "LONG"); }
    | FLOAT         { $$ = new SymbolInfo("float", "FLOAT"); }
    | DOUBLE        { $$ = new SymbolInfo("double", "DOUBLE"); }
    | SIGNED        { $$ = new SymbolInfo("signed", "SIGNED"); }
    | UNSIGNED      { $$ = new SymbolInfo("unsigned", "UNSIGNED"); }
    | BOOL          { $$ = new SymbolInfo("bool", "BOOL"); }
    | COMPLEX       { $$ = new SymbolInfo("complex", "COMPLEX"); }
    | IMAGINARY     { $$ = new SymbolInfo("imaginary", "IMAGINARY"); }
    | USER_DEFINED  { $$ = $1; }
    | struct_or_union_specifier  { $$ = $1; }
    | enum_specifier             { $$ = $1; }
```

Ilustración 51: Definición de Tipos con USER_DEFINED

Y por último dentro de la declaración de variables tendremos que marcar las variables definidas con *typedef* como tipo definido por usuario:

```
| declaration_specifiers init_declarator_list ';' {
    $$ = new vector<SymbolInfo*>();
    bool hasTypedef = (strstr($1->getSymbolType().c_str(), "TYPEDEF") != NULL);
    if($1->isStruct()){
        if($1->getParamList() != nullptr){
            for(std::vector<SymbolInfo*>::size_type i = 0; i < $2->size(); i++){
                if(hasTypedef) $2->at(i)->setSymIsType(true);
                $2->at(i)->setVariableType($1->getSymbolType());
                $2->at(i)->setIsStruct(true);
                $2->at(i)->setParamList($1->getParamList());
                if(declarePragma){
                    mpi_utils.write_MPI_Type_struct($2->at(i));
                }
                SymbolInfo* symbol = new SymbolInfo(*$2->at(i));
                $$->push_back(symbol);
                table.insert($2->at(i));
            }
        }
        else {
            for(std::vector<SymbolInfo*>::size_type i = 0; i < $2->size(); i++){
                std::ostringstream oss = std::ostringstream();
                oss << $1->getVariableType() << "_" << $1->getSymbolName();
                $2->at(i)->setVariableType(oss.str());
                table.insert($2->at(i));
                $$->push_back($2->at(i));
            }
        }
    }
    else{
        for(std::vector<SymbolInfo*>::size_type i = 0; i < $2->size(); i++){
            $2->at(i)->setVariableType($1->getSymbolType());
            if(hasTypedef) $2->at(i)->setSymIsType(true);
            if(!$2->at(i)->isFunction()){
                SymbolInfo* symbol = new SymbolInfo(*$2->at(i));
                $$->push_back(symbol);
                table.insert($2->at(i));
            }
        }
    }
}
```

Ilustración 52: Declaración de Tipos

Nos centraremos en las siguientes sentencias añadidas a las reglas de declaración de variables:

```
bool hasTypedef = (strstr($1->getSymbolType().c_str(), "TYPEDEF") != NULL);
```

Comprobaremos que el tipo de la variable contiene un substring “TYPEDEF”. Si este es el caso se marcará el identificiador como tipo definido por usuario: `if(hasTypedef) $2->at(i)->setSymIsType(true);`Esta sentencia se aplicará tanto para variables normales como de tipo *struct* o *enum*.

Si estamos dentro de la declaración de tipos *struct* y *declarePragma* está activado, generaremos la traducción de tipos llamando a la funcion `write_MPI_Type_struct` de *mpi_utils*.

Traducción de Tipos Definidos por Usuario

Para el ejemplo definido anteriormente la traducción sería:

```
typedef struct{
  unsigned char bl,gr,re;
} color;

MPI_Datatype MPIcolor_t;

void __Declare_MPI_Type_color () {
    int blocklengths[3];
    MPI_Datatype old_types[3];
    MPI_Aint disp[3];
    MPI_Aint lb;
    MPI_Aint extent;
    blocklengths[0]= 1;
    blocklengths[1]= 1;
    blocklengths[2]= 1;
    old_types[0]= MPI_UNSIGNED_CHAR;
    old_types[1]= MPI_UNSIGNED_CHAR;
    old_types[2]= MPI_UNSIGNED_CHAR;
    MPI_Type_get_extent(MPI_UNSIGNED_CHAR, &lb, &extent);
    disp[0]= lb;
    MPI_Type_get_extent(MPI_UNSIGNED_CHAR, &lb, &extent);
    disp[1]= disp[0] + extent;
    MPI_Type_get_extent(MPI_UNSIGNED_CHAR, &lb, &extent);
    disp[2]= disp[1] + extent;
    MPI_Type_create_struct(3,blocklengths, disp, old_types, &MPIcolor_t);
    MPI_Type_commit(&MPIcolor_t);
}

void Declare_MPI_Types () {
  __Declare_MPI_Type_color ();
  return;
}
```

Ilustración 53: Traducción de Tipos Definidos por Usuario

La función `__Declare_MPI_Type_color` define el tipo de dato MPI para la estructura "color". Declara *arrays blocklengths*, old_*types* y *disp* para almacenar la longitud de los bloques, los tipos de los elementos y los desplazamientos, respectivamente. Declara variables *lb* (lower bound) y *extent* para calcular los desplazamientos. La traducción de esta parte sería:

```
    void write_MPI_Type_struct(SymbolInfo *symbol)
    {
        insert_MPI_buffer_line(0, 1);
        var_glob << "\n\nMPI_Datatype MPI" << symbol->getSymbolName() << "_t;\n\n";
        var_glob << "void __Declare_MPI_Type_" << symbol->getSymbolName() << "() {\n";
        var_glob << "    int blocklengths[" << symbol->getParamList()->size() << "];\n";
        var_glob << "    MPI_Datatype old_types[" << symbol->getParamList()->size() << "];\n";
        var_glob << "    MPI_Aint disp[" << symbol->getParamList()->size() << "];\n";
        var_glob << "    MPI_Aint lb;\n";
        var_glob << "    MPI_Aint extent;\n";
```

Ilustración 54: Fragmento 1 de la traducción de tipos

Cada campo de la estructura color tiene una longitud de bloque de 1, ya que cada uno es un solo `unsigned char` y cada campo se corresponde con el tipo [MPI_UNSIGNED_CHAR]. La traducción de esta parte sería:

```
 for (std::vector<SymbolInfo *>::size_type i = 0; i < symbol->getParamList()-
>size(); i++)
 {
  var_glob << "    blocklengths[" << i << "] = 1;\n";
 }

 for (std::vector<SymbolInfo *>::size_type i = 0; i < symbol->getParamList()-
>size(); i++)
 {
 var_glob << "    old_types[" << i << "] = MPI_" << symbol->getParamList()-
>at(i)->getVariableType() << ";\n";
 }
```

Ilustración 55: Fragmento 2 de la traducción de tipos

A continuación, se calcula el desplazamiento de cada campo dentro de la estructura. `MPI_Type_get_extent` obtiene el límite inferior (*lb*) y la extensión (*extent*) de [MPI_UNSIGNED_CHAR]. `disp[0]` se inicializa con *lb* y disp[n] se calcula sumando la extensión al desplazamiento anterior. La traducción de esta parte sería:

```
      for (std::vector<SymbolInfo *>::size_type i = 0; i < symbol-
>getParamList()->size(); i++)
      {
            var_glob << "    MPI_Type_get_extent(MPI_" << symbol-
>getParamList()->at(i)->getVariableType() << ", &lb, &extent);\n";


            if (i == 0)
                var_glob << "    disp[" << i << "] = lb;\n";
            else
                var_glob << "    disp[" << i << "] = disp[" << i - 1 << "] +
extent;\n";
      }
```

Ilustración 56: Fragmento 3 de la traducción de tipos

`MPI_Type_create_struct` crea el tipo de dato MPI *MPIcolor_t* usando los *arrays blocklengths*, *disp* y *old_types*. `MPI_Type_commit` crea el tipo de dato para que pueda ser utilizado en comunicaciones MPI. La traducción de esta parte sería:

```
        var_glob << "    MPI_Type_create_struct(" <<
         symbol->getParamList()->size() << ", blocklengths, disp,
         old_types, &MPI" << symbol->getSymbolName() << "_t);\n";

        var_glob << "    MPI_Type_commit(&MPI" << symbol->getSymbolName()
         << "_t);\n";

        var_glob << "}\n\n";
```

Ilustración 57: Fragmento 4 de la traducción de tipos

Por ultimo la función `Declare_MPI_Types` llama a `__Declare_MPI_Type_color` para declarar el tipo de dato MPI para color. Esta función puede ser llamada durante la inicialización del programa para asegurar que todos los tipos de datos MPI definidos por el usuario estén listos para su uso. La traducción de esta parte sería:

```
        var_glob << "void Declare_MPI_Types() {\n";
        var_glob << "    __Declare_MPI_Type_" << symbol->getSymbolName()
                 << "();\n";
        var_glob << "    return;\n";
        var_glob << "}\n";

        mainFuncion->write_type_def();
}
```

Ilustración 58: Fragmento 5 de la traducción de tipos

En la traducción llamaremos a `write_type_def()` para que cree la llamada a la función `Declare_MPI_Types` dentro de la función *main* del archivo generado.

5 Pruebas y Validación

5.1 Estrategias y Pruebas Implementadas

Para validar el funcionamiento correcto del traductor dispondremos de 3 archivos de prueba distintos para evaluar diferentes aspectos del traductor y asegurar que las directivas sean correctamente traducidas y ejecutadas en un entorno MPI.

Nos enfocaremos exclusivamente en la generación de código para el funcionamiento correcto del programa en MPI y en la traducción de tipos de datos definidos por el usuario mediante las `directivas #pragma omp declare cluster` y `#pragma omp end declare cluster`. Otras directivas de comunicación y sincronización (`#pragma omp cluster`, `#pragma omp teams distribute`, `#pragma omp parallel for`) serán ignoradas en esta fase del proyecto y se añadirán manualmente los códigos MPI correspondientes para completar las pruebas.

Archivos de Prueba

Los tres archivos de prueba serán clasificados según su complejidad de traducción: sencilla, intermedia y avanzada. Presentaremos en el anexo el archivo generado con la información de la tabla de símbolos.

Ejemplo Sencillo: *pi_ompd.c*

Este archivo de prueba calcula el valor de π utilizando el método de integración numérica. El código original usa directivas OMPD para paralelizar el cálculo, y se traduce a MPI para distribuir el trabajo entre múltiples nodos.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define NUM_THREADS 8

    static unsigned long int num_steps = (unsigned long int)((1<<30)-1);
    double step;
```

Ilustración 59: Cabecera y Variables Globales pi_ompd.c

```c
int main (int argc, char** argv) {
    unsigned int i;
    double pi;
    double PI25DT = 3.14159265358979323846264643;
    double x;
    double sum=0.0;
    int factor=1;


    struct timeval t1, t2;
    double segundos;

    if (argc == 1) {
        printf("num_steps %ld, ¿Factor de escala (1..4)?\n", num_steps);
        if (scanf("%d", &factor) <= 0)
        printf("scanf error, factor sin cambio %d\n", factor);
    }
    else factor = atoi(argv[1]);

    num_steps = num_steps * factor;

    printf("%ld num_steps, %25.23f step size i: %ld size num_steps: %ld\n",
num_steps, (double)1.0/(double) num_steps, sizeof(i), sizeof(num_steps));
gettimeofday(&t1, NULL);

step = 1.0/(double) num_steps;
// // omp_set_num_threads(NUM_THREADS);

#pragma omp cluster broad(num_steps, step)
#pragma omp teams distribute reduction(+:sum)
#pragma omp parallel for simd private(x)
// for (i=0;i< num_steps; i++) {
//     x = (i+0.5)*step;
//     sum += 4.0/(1.0+x*x);
// }
pi = step * sum;

gettimeofday(&t2, NULL);
segundos = (((t2.tv_usec - t1.tv_usec)/1000000.0f)  + (t2.tv_sec - t1.tv_sec));

printf("Pi %25.23f, calc con %ld pasos en %f segundos\n", pi,num_steps,segundos);
printf("Pi es %25.23f, Error relativo %10.8e\n", PI25DT, (double)100 * (pi -
PI25DT)/PI25DT);

return(0);
}
```

Ilustración 60: Main pi_ompd.c

La traducción es sencilla, ya que el uso de directivas OMPD no incluye la generación de tipos y se traduce fácilmente a MPI sin necesidad de gestionar funciones externas a *main* ni estructuras complejas.

Traducción *pi_ompd.c*

Se añaden las librerías correspondientes de MPI (`<assert.h>, <mpi.h>`) y se inicializan las variables globales `__taskid` y `__numprocs` al final de la cabecera.

```c
#include <assert.h>
#include <mpi.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int __taskid = -1, __numprocs = -1;

#define NUM_THREADS 8

    static unsigned long int num_steps = (unsigned long int)((1<<30)-1) ;
    double step;
int main (int argc, char** argv) {
    unsigned int i;
    double pi;
    double PI25DT = 3.14159265358979323846264643;
    double x;
    double sum=0.0;
    int factor=1;
    struct timeval t1, t2;
    double segundos;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD,&__numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&__taskid);


if (__taskid == 0) {
    if (argc == 1) {
        printf("num_steps %ld, ¿Factor de escala (1..4)?\n", num_steps);
        if (scanf("%d", &factor) <= 0)
        printf("scanf error, factor sin cambio %d\n", factor);
    }
    else factor = atoi(argv[1]);

    num_steps = num_steps * factor;

    printf("%ld num_steps, %25.23f step size i: %ld size num_steps: %ld\n",
num_steps, (double)1.0/(double) num_steps, sizeof(i), sizeof(num_steps));
gettimeofday(&t1, NULL);

step = 1.0/(double) num_steps;
}
// pragma aqui
// pragma aqui
// pragma aquí
if (__taskid == 0) {
pi = step * sum;

gettimeofday(&t2, NULL);
segundos = (((t2.tv_usec - t1.tv_usec)/1000000.0f)  + (t2.tv_sec - t1.tv_sec));

printf("Pi %25.23f, calc con %ld pasos en %f segundos\n", pi,num_steps,segundos);
printf("Pi es %25.23f, Error relativo %10.8e\n", PI25DT, (double)100 * (pi -
PI25DT)/PI25DT);

}
MPI_Finalize();
return(0);
}
```

Ilustración 61: Traducción Cabecera y Variables Globales pi_ompd.c

Se inicializa y finaliza el entorno MPI en el *main*, se guardan las variables locales y se generan las premisas para el código secuencial. En esta fase actual del proyecto se ignorarán los pragmas no relacionados con la generación de tipos.

Ejemplo Intermedio: *ParticlesOMPD.c*

Este archivo de prueba simula la dinámica de partículas, utilizando estructuras complejas y paralelismo para distribuir el trabajo. El código original usa directivas OMPD para manejar la simulación en paralelo, y se traduce a MPI para su ejecución en un entorno distribuido.

```
#include <stdlib.h>
#include <stdio.h>
#pragma omp declare cluster
typedef struct {
  float x, y, z;
  double velocity;
  int  n;
  char type;
} Particle;
#pragma omp end declare cluster
void init(Particle p[], int nelem);
```

Ilustración 62: Cabecera, Variables Globales y Declaración de Tipos ParticlesOMPD.c

```
int main(int argc, char **argv)
{
  Particle *particles;
  int part_num;
  int step_num;
  float dt;
  int i, t;

  if (argc != 4) {
    printf("Particles part_num step_num dt\n");
    exit(-1);
  }

  part_num = atoi ( argv[1] );
  step_num = atoi ( argv[2] );
  dt = atof ( argv[3] );
  particles = ( Particle * ) malloc ( part_num * sizeof ( Particle ) );
  init(particles, part_num);

#pragma omp cluster broad(part_num, dt, step_num) scatter(particles[part_num])
gather(particles[part_num])
//   for (t=0; t<step_num; t++) {
// #pragma omp teams distribute
// #pragma omp parallel for
//     for (i=0; i<part_num; i++) {
//       particles[i].x += particles[i].velocity * dt;
//     }
//   }

printf("particle[3]:   %3.2f %3.2f %3.2f %3.2f %d %d\n", particles[3].x,
     particles[3].y,particles[3].z,particles[3].velocity,particles[3].n,particles[3].
type);
}
```

Ilustración 63: Función main ParticlesOMPD.c

```
void init(Particle p[], int nelem) {
  int i;

  for (i=0; i<nelem; i++) {
     p[i].x = i * 1.0;
     p[i].y = i * -1.0 * 2;
     p[i].z = i * 1.0 * 3;
     p[i].velocity = 0.25;
     p[i].n = i;
     p[i].type = i % 2;
  }
}
```

Ilustración 64: Funciones externas ParticlesOMPD.c

La complejidad de traducción es intermedia, debido a la necesidad de traducir estructuras definidas por el usuario, manejar la distribución y recolección de datos entre los nodos del clúster y mantener la coherencia del programa. Sin embargo, el archivo no invoca funciones que ejecuten pragmas, por lo que eso reduce su complejidad.

Traducción *ParticlesOMPD.c*

Como en el anterior archivo se añaden las librerías correspondientes de MPI (`<assert.h>`, `<mpi.h>`) y se inicializan las variables globales `__taskid` y `__numprocs` al final de la cabecera.

Se define la estructura *Particle* y su correspondiente tipo de dato MPI *MPIParticle_t*. La función `__Declare_MPI_Type_Particle` crea el tipo de dato MPI usando `MPI_Type_create_struct` y `MPI_Type_commit`.

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

int __taskid = -1, __numprocs = -1;

typedef struct {
  float x, y, z;
  double velocity;
  int  n;
  char type;
} Particle;

MPI_Datatype MPIParticle_t;

void __Declare_MPI_Type_Particle() {
    int blocklengths[6];
    MPI_Datatype old_types[6];
    MPI_Aint disp[6];
    MPI_Aint lb;
    MPI_Aint extent;
    blocklengths[0] = 1;
    blocklengths[1] = 1;
    blocklengths[2] = 1;
    blocklengths[3] = 1;
    blocklengths[4] = 1;
    blocklengths[5] = 1;
    old_types[0] = MPI_FLOAT;
    old_types[1] = MPI_FLOAT;
    old_types[2] = MPI_FLOAT;
    old_types[3] = MPI_DOUBLE;
    old_types[4] = MPI_INT;
    old_types[5] = MPI_CHAR;
    MPI_Type_get_extent(MPI_FLOAT, &lb, &extent);
    disp[0] = lb;
    MPI_Type_get_extent(MPI_FLOAT, &lb, &extent);
    disp[1] = disp[0] + extent;
    MPI_Type_get_extent(MPI_FLOAT, &lb, &extent);
    disp[2] = disp[1] + extent;
    MPI_Type_get_extent(MPI_DOUBLE, &lb, &extent);
    disp[3] = disp[2] + extent;
    MPI_Type_get_extent(MPI_INT, &lb, &extent);
    disp[4] = disp[3] + extent;
    MPI_Type_get_extent(MPI_CHAR, &lb, &extent);
    disp[5] = disp[4] + extent;
    MPI_Type_create_struct(6, blocklengths, disp, old_types, &MPIParticle_t);
    MPI_Type_commit(&MPIParticle_t);
}

void Declare_MPI_Types() {
    __Declare_MPI_Type_Particle();
    return;
}

void init(Particle p[], int nelem);
```

Ilustración 65: Traducción Cabecera, Variables Globales y Declaración de Tipos ParticlesOMPD.c

Como en el anterior caso, se inicializa y finaliza el entorno MPI en el *main*, se guardan las variables locales y se generan las premisas para el código secuencial. En este caso se llama a la función `Declare_MPI_Types()` dentro del main para asegurarnos que todos los tipos de datos MPI definidos por el usuario (*Particle*) estén listos para su uso.

```c
int main(int argc, char **argv)
{
  Particle *particles;
  int part_num;
  int step_num;
  float dt;
  int i, t;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD,&__numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&__taskid);


Declare_MPI_Types();

if (__taskid == 0) {
  if (argc != 4) {
    printf("Particles part_num step_num dt\n");
    exit(-1);
  }

  part_num = atoi ( argv[1] );
  step_num = atoi ( argv[2] );
  dt = atof ( argv[3] );

  particles = ( Particle * ) malloc ( part_num * sizeof ( Particle ) );

  init(particles, part_num);

}

// pragma aqui

if (__taskid == 0) {
printf("particle[3]:   %3.2f %3.2f %3.2f %3.2f %d %d\n", particles[3].x,
     particles[3].y,particles[3].z,particles[3].velocity,particles[3].n,particles[3].type);

}
MPI_Finalize();
}
```

Ilustración 66: Traducción de la Función main ParticlesOMPD.c

La función *init* del programa mantendrá su código original sin ser modificada.

Ejemplo Avanzado: *JuliaOMPD.c*

Este archivo de prueba genera el conjunto de Julia, una conocida fractal matemática, utilizando paralelismo. El código original utiliza directivas OMPD para distribuir la carga de trabajo en diferentes *threads*.

```c
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include<omp.h>
#include<complex.h>
#include<tgmath.h>
#include <sys/time.h>


#define DIM 8192

#pragma omp declare cluster
typedef struct{
  unsigned char bl,gr,re;
} color;
#pragma omp end declare cluster

void tga_write ( int w, int h, color rgb[], char *filename );
```

Ilustración 67: Cabecera, Variables Globales y Declaración de Tipos JuliaOMPD.c

```c
color fcolor(int iter,int num_its){
        color c;
        c.re = (iter*20+0)%255;
        c.gr = (iter*20+85)%255;
        c.bl = (iter*20+170)%255;
        return c;
}

int explode (float _Complex z0, float _Complex c, float radius, int n)
{
int k=1;
float modul;

z0 = (z0*z0)+c;
modul = cabsf(z0);

while ((k<=n) && (modul<=radius)){
    z0 = (z0*z0)+c;
    modul = cabsf(z0);
    k++;
}
return k;
}

float _Complex mapPoint(int width,int height,float radius,int x,int y){
  float _Complex c;
  int l = (width<height)?width:height;
  float re = 2*radius*(x - width/2.0)/l;
  float im = 2*radius*(y - height/2.0)/l;
  c = re+im*I;
  return c;
}

void tga_write ( int w, int h, color rgb[], char *filename )
{
  FILE *file_unit;
  unsigned char header1[12] = { 0,0,2,0,0,0,0,0,0,0,0,0 };
  unsigned char header2[6] = { w%256, w/256, h%256, h/256, 24, 0 };

  file_unit = fopen ( filename, "wb" );

  fwrite ( header1, sizeof ( unsigned char ), 12, file_unit );
  fwrite ( header2, sizeof ( unsigned char ), 6, file_unit );
  fwrite ( rgb, sizeof ( unsigned char ), 3 * w * h, file_unit );
  fclose ( file_unit );
  printf ( "\n" );
  printf ( "TGA_WRITE:\n" );
  printf ( "  Graphics data saved as '%s'\n", filename );

  return;
}
```

Ilustración 68: Funciones Externas JuliaOMPD.c

```
color *juliaSet(int width,int height,float _Complex c,float radius,int iter){
  int x,y,i;
  float _Complex z0;
  int k=0;
  int count=0;
  color *rgb;


  rgb = calloc (width*height, sizeof(color));


#pragma omp cluster map(broad:width, height, c, radius, iter) map(gather:rgb[height*width:chunk(width)])
#pragma omp teams distribute dist_schedule(static,1)
#pragma omp parallel for private (x,y,k,i,z0) shared(rgb,width,height)


  return rgb;
}
```

Ilustración 69: Función juliaSet de JuliaOMPD.c

```
int main(int argc, char* argv[])
{
int width, height;
float _Complex c;
color *rgb;
#ifdef _OPENMP
double start_time, end_time;
#else
struct timeval tv_start, tv_end;
float tiempo_trans;
#endif


  if(argc != 6) {
    printf("Uso : %s\n", "<dim de la ventana, partes real e imaginaria de c, radio, iteraciones>");
    exit(1);
  }
    width = atoi(argv[1]);
    height = width; // La ventana es cuadrada
    if (width >DIM) {
        printf("El tamanyo de la ventana deben ser menor que 1024\n");
        exit(1);
    }
    float re = atof(argv[2]);
    float im = atof(argv[3]);
    c=re+im*I;
  printf("JuliaSet: %d, %d, %f, %f, %f, %d\n", width, height,creal(c),cimag(c),atof(argv[4]),atoi(argv[5]));
#ifdef _OPENMP
  start_time = omp_get_wtime();
#else
  gettimeofday(&tv_start, NULL);
#endif


  rgb = juliaSet(width,height,c,atof(argv[4]), atoi(argv[5]));

#ifdef _OPENMP
  end_time = omp_get_wtime();
  printf ( "Tiempo Julia = %f segundos\n",end_time-start_time);
#else
  gettimeofday(&tv_end, NULL);
  tiempo_trans=(tv_end.tv_sec - tv_start.tv_sec) * 1000000 +
    (tv_end.tv_usec - tv_start.tv_usec); //en us
  printf("Tiempo Julia = %f segundos\n", tiempo_trans/1000000);
#endif
  tga_write ( width, height, rgb, "julia_set.tga" );
    printf ( "\n" );
    printf ( "JULIA_SET. Finalizado\n");
    free(rgb);

    return 0;
}
```

Ilustración 70: Función main de JuliaOMPD.c

La complejidad de traducción es avanzada, debido a la necesidad de gestionar la invocación de funciones anidadas que ejecutan código MPI, múltiples variables privadas y definición de tipos de datos definidos por el usuario.

Traducción *JuliaOMPD.c*

Como en el anterior ejemplo se añaden las librerías correspondientes de MPI (`<assert.h>, <mpi.h>`) y se inicializan las variables globales `__taskid` y `__numprocs` al final de la cabecera.

Se define la estructura *Color* y su correspondiente tipo de dato MPI *MPIColor_t*. La función `__Declare_MPI_Type_Particle` crea el tipo de dato MPI usando `MPI_Type_create_struct` y `MPI_Type_commit`.

```
#include <assert.h>
#include <mpi.h>
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include<omp.h>
#include<complex.h>
#include<tgmath.h>
#include <sys/time.h>

int __taskid = -1, __numprocs = -1;


#define DIM 8192

typedef struct{
        unsigned char bl,gr,re;
} color;

MPI_Datatype MPIcolor_t;

void __Declare_MPI_Type_color() {
    int blocklengths[3];
    MPI_Datatype old_types[3];
    MPI_Aint disp[3];
    MPI_Aint lb;
    MPI_Aint extent;
    blocklengths[0] = 1;
    blocklengths[1] = 1;
    blocklengths[2] = 1;
    old_types[0] = MPI_UNSIGNED_CHAR;
    old_types[1] = MPI_UNSIGNED_CHAR;
    old_types[2] = MPI_UNSIGNED_CHAR;
    MPI_Type_get_extent(MPI_UNSIGNED_CHAR, &lb, &extent);
    disp[0] = lb;
    MPI_Type_get_extent(MPI_UNSIGNED_CHAR, &lb, &extent);
    disp[1] = disp[0] + extent;
    MPI_Type_get_extent(MPI_UNSIGNED_CHAR, &lb, &extent);
    disp[2] = disp[1] + extent;
    MPI_Type_create_struct(3, blocklengths, disp, old_types, &MPIcolor_t);
    MPI_Type_commit(&MPIcolor_t);
}

void Declare_MPI_Types() {
    __Declare_MPI_Type_color();
    return;
}
void tga_write ( int w, int h, color rgb[], char *filename );
```

Ilustración 71: Traducción Cabecera, Variables Globales y Declaración de Tipos JuliaOMPD.c

Se traduce la función *juliaSet* rresponsable de calcular el conjunto de Julia. Utiliza paralelismo para distribuir la carga de trabajo en diferentes *threads*, utilizando directivas OMPD. Por lo que en la traducción tendremos que generar las premisas para el código secuencial y marcarla en la tabla de símbolo como una función que ejecuta pragmas.

```
color *juliaSet(int width,int height,float _Complex c,float radius,int iter){
        int x,y,i;
        float _Complex z0;
        int k=0;
        int count=0;

        color *rgb;

if (__taskid == 0) {
        rgb = calloc (width*height, sizeof(color));

}
// pragma aqui
// pragma aqui
if (__taskid == 0) {
}
        return rgb;
}
```

Ilustración 72: Traducción juliaSet de JuliaOMPD.c

Igual que el resto de los casos, se inicializa y finaliza el entorno MPI en el *main*, se guardan las variables locales y se generan las premisas para el código secuencial. Se llama a la función `Declare_MPI_Types()` dentro del *main* para asegurarnos que todos los tipos de datos MPI definidos por el usuario (*Color*) estén listos para su uso.

```
int main(int argc, char* argv[])
{
int width, height;
float _Complex c;
color *rgb;
#ifdef _OPENMP
double start_time, end_time;
#else
struct timeval tv_start, tv_end;
float tiempo_trans;
#endif

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD,&__numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&__taskid);

Declare_MPI_Types();

if (__taskid == 0) {
        if(argc != 6) {
                printf("Uso : %s\n", "<dim de la ventana, partes real e imaginaria de c, radio,
iteraciones>");
                exit(1);
        }

                width = atoi(argv[1]);
                height = width;
                if (width >DIM) {
                   printf("El tamanyo de la ventana deben ser menor que 1024\n");
                   exit(1);
                }
                float re = atof(argv[2]);
                float im = atof(argv[3]);

                c=re+im*I;

        printf("JuliaSet: %d, %d, %f, %f, %f, %d\n", width,
height,creal(c),cimag(c),atof(argv[4]),atoi(argv[5]));
#ifdef _OPENMP
        start_time = omp_get_wtime();
#else
        gettimeofday(&tv_start, NULL);
#endif
}
        rgb = juliaSet(width,height,c,atof(argv[4]), atoi(argv[5]));


if (__taskid == 0) {
#ifdef _OPENMP
        end_time = omp_get_wtime();
        printf ( "Tiempo Julia = %f segundos\n",end_time-start_time);
#else
        gettimeofday(&tv_end, NULL);
        tiempo_trans=(tv_end.tv_sec - tv_start.tv_sec) * 1000000 +
          (tv_end.tv_usec - tv_start.tv_usec);
        printf("Tiempo Julia = %f segundos\n", tiempo_trans/1000000);
#endif
        tga_write ( width, height, rgb, "julia_set.tga" );
        printf ( "\n" );
        printf ( "JULIA_SET. Finalizado\n");
        free(rgb);

}
MPI_Finalize();
        return 0;
}
```

Ilustración 73: Traduccion main de JuliaOMPD.c

Estos tres archivos de prueba permiten evaluar la capacidad del traductor para manejar diferentes niveles de complejidad en la traducción. Desde la generación del código en el *main* hasta la gestión de estructuras complejas y tipos definidos por el usuario, estas pruebas aseguran que el traductor pueda manejar una variedad de escenarios comunes en la programación paralela y distribuida. Como ya hemos comentado los pragmas que no están relacionados con la traducción de tipos de datos han sido ignorados y el código MPI correspondiente se añade manualmente aceptándose así la compilación y permitiendo el funcionamiento correcto del archivo generado.

6 Resultados y Conclusiones

El objetivo final del Trabajo Fin de Grado fue el de crear una herramienta para traducir código C con directivas OMPD (OpenMP Distribuido) a MPI (Message Passing Interface). Para ello en primer lugar se logró integrar una tabla de símbolos eficiente, utilizando un enfoque basado en *hashmap*, para manejar la información sobre variables, funciones y tipos definidos por el usuario. Esta tabla se incorporó en la gramática extendida, y acabo siendo un elemento esencial del proceso de traducción.

El aspecto clave del proyecto fue la traducción de directivas de definición de tipos de datos de OMPD a MPI. Para ello se tuvo que definir un plan de traducción y generación de código del archivo original, un proceso que permitiera mantener la transparencia del código y su funcionalidad original. Este plan se implementó y desarrollo mediante la creación de una nueva clase *MPIUtils.h* que manejara las directivas de generación y traducción del código, integrándolo en la gramática y escáner.

El traductor genera un archivo de salida que reemplaza con éxito las directivas OMPD con llamadas a MPI, lo que fue validado con éxito mediante varios ejemplos prácticos.

Personalmente, este proyecto fue extremadamente desafiante, ya que implicó aprender desde cero tecnologías y conceptos como OMPD, Flex/Bison y MPI. Empezar con un conocimiento limitado y llegar a desarrollar una solución funcional ha sido una experiencia muy enriquecedora. Este reto me permitió adquirir una comprensión profunda de los compiladores y traductores de código, así como de la programación en sistemas de memoria distribuida.

En resumen, este Trabajo Fin de Grado no solo ha producido una herramienta valiosa para la programación en sistemas de memoria distribuida, sino que también me ha proporcionado una plataforma para aprender y aplicar conocimientos críticos en compiladores, paralelismo y computación distribuida.

Además, trabajar en un equipo donde cada miembro se encargaba de una parte específica del proyecto me enseñó la importancia del trabajo colaborativo en el desarrollo de proyectos complejos. Esta experiencia me ha preparado para enfrentar desafíos similares en mi vida profesional, donde la colaboración y el trabajo en equipo son esenciales.

El código completo del proyecto y más detalles pueden encontrarse en mi repositorio de GitHub: OMPD Translator.

[https://github.com/taponplaza/Tabla_y_Tipos].

7 Análisis de Impacto

Impacto Personal

Como ya he comentado anteriormente, el desarrollo de este TFG ha tenido un impacto significativo a nivel personal. El proyecto ha permitido adquirir nuevos conocimientos y habilidades en tecnologías como OMPD, Flex/Bison y MPI, comenzando desde un nivel básico hasta alcanzar una comprensión profunda. Esta experiencia ha fortalecido mis competencias en programación paralela y distribuida, así como en el desarrollo de compiladores y traductores de código. Además, ha fomentado habilidades de gestión del tiempo y resolución de problemas, fundamentales para enfrentar desafíos técnicos complejos.

Impacto Empresarial

En un contexto empresarial, la herramienta desarrollada puede ser de gran utilidad para empresas que trabajan con computación de alto rendimiento y sistemas de memoria distribuida. La simplificación de la programación paralela mediante OMPD puede mejorar la eficiencia y reducir los costos asociados con el desarrollo de software paralelo y distribuido. Empresas en sectores como la simulación científica, la ingeniería y el análisis de grandes volúmenes de datos pueden beneficiarse significativamente al adoptar esta herramienta, mejorando la productividad y la calidad del software.

Impacto Social

Desde una perspectiva social, la disponibilidad de herramientas que simplifican la programación paralela y distribuida puede democratizar el acceso a tecnologías avanzadas. Esto permite que un mayor número de investigadores, desarrolladores y estudiantes puedan aprovechar las ventajas de la computación de alto rendimiento sin enfrentarse a la complejidad de MPI.

Impacto Económico

Económicamente, la herramienta tiene el potencial de reducir los costos de desarrollo y mantenimiento de software paralelo y distribuido. Al simplificar la programación con directivas claras y fáciles de usar, se disminuyen los tiempos de desarrollo y se mejora la eficiencia operativa. Esto puede traducirse en ahorros significativos para las empresas, especialmente aquellas que dependen de la computación de alto rendimiento para sus operaciones diarias. Además, al mejorar la accesibilidad a estas tecnologías, se pueden abrir nuevas oportunidades de negocio y fomentar la innovación.

Impacto Medioambiental

El impacto medioambiental de este proyecto puede ser indirecto pero significativo. Al mejorar la eficiencia del software paralelo y distribuido, se optimiza el uso de los recursos computacionales, lo que puede llevar a una reducción en el consumo de energía. Dado que los centros de datos y las infraestructuras de computación de alto rendimiento son grandes consumidores de energía, cualquier mejora en la eficiencia puede contribuir a

una disminución en las emisiones de carbono y otros impactos ambientales negativos.

Impacto Cultural

Culturalmente, la herramienta promueve una cultura de innovación y aprendizaje continuo. Al facilitar la programación en sistemas de memoria distribuida, se fomenta la adopción de nuevas tecnologías y prácticas en diversas disciplinas. Esto puede llevar a un mayor intercambio de conocimientos y colaboración entre comunidades científicas y tecnológicas, enriqueciendo el entorno cultural y académico.

Relación con los Objetivos de Desarrollo Sostenible (ODS)

Este proyecto contribuye a varios Objetivos de Desarrollo Sostenible (ODS) de la Agenda 2030. En particular:

- **ODS 4: Educación de Calidad**: Al facilitar el aprendizaje de tecnologías avanzadas, se promueve una educación de calidad y accesible para todos.
- **ODS 9: Industria, Innovación e Infraestructura**: La herramienta impulsa la innovación y mejora la infraestructura tecnológica, fomentando el desarrollo industrial sostenible.
- **ODS 12: Producción y Consumo Responsables**: Al optimizar la eficiencia del software y reducir el consumo energético, se contribuye a prácticas de producción y consumo más responsables.
- **ODS 13: Acción por el Clima**: La reducción en el consumo de energía y las emisiones asociadas contribuye a la acción global contra el cambio climático.

Decisiones Basadas en el Impacto

A lo largo del desarrollo del proyecto, se tomaron varias decisiones considerando el impacto potencial. La elección de utilizar tecnologías abiertas y bien documentadas como OpenMP y MPI asegura que la herramienta sea accesible y utilizable por una amplia comunidad. Además, se optó por un diseño modular y extensible, facilitando futuras mejoras y adaptaciones. Estas decisiones reflejan un compromiso con la sostenibilidad, la inclusión y la promoción de prácticas responsables en el desarrollo de software.

8 Bibliografía

[1] R. Rabenseifner, G. Hager y G. Jost, «Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes», 2009, 10.1109/PDP.2009.43.

[2] L. Dagum y R. Menon, «OpenMP: an industry standard API for shared-memory programming» *IEEE Computational Science and Engineering,* vol. 5, pp. 46-55, 1998.

[3] M. A. P. Cabero, «OpenMP para Memoria Distribuida. Reparto de las iteraciones.» 2024. Trabajo Fin de Grado, ETSI Informáticos UPM.

[4] Á. D. Martín, «OpenMP para Memoria Distribuida. Envío de mensajes.» 2024. Trabajo Fin de Grado, ETSI Informáticos UPM.

[5] F. S. Miguel, «OpenMP para Memoria Distribuida. Recolección de resultados» 2024. Trabajo Fin de Grado, ETSI Informáticos UPM.

[6] Barcelona Supercomputing Center, «Mercurium C/C++/Fortran source-to-source compiler» 2018. [En línea]. Available: https://github.com/bsc-pm/mcxx.

[7] M. J. Djomehri, H. H. Jin y B. Biegel, «Hybrid MPI+ OpenMP programming of an overset CFD solver and performance investigations» 2002.

[8] B. Chapman, Using OpenMP : Portable Shared Memory Parallel Programming., 1st ed. ed., 2007.

[9] R. Shikder, P. Thulasiraman, P. Irani y P. Hu, «An OpenMP-based tool for finding longest common subsequence in bioinformatics» *BMC research notes,* vol. 12, p. 1–6, 2019.

[10] M. J. M. J. Quinn, Parallel programming in C with MPI and OpenMP, New York: McGraw-Hill, 2003.

[11] J. R. Levine, Flex & bison, 1st ed. ed., Sebastopol, Calif.: O'Reilly Media, 2009.

[12] A. Grama, Introduction to parallel computing, 2nd ed. ed., Harlow, England ; New York: Addison-Wesley, 2003.

[13] V. Gloukhov, «Parallel Computations in Problems of Climate Modeling» pp. 301-308, 2004, 10.1016/B978-044451612-1/50038-X.

[14] E. S. Raymond, The cathedral and the bazaar : musings on Linux and Open Source by an accidental revolutionary, Revised and expanded edition. ed., 2001.

[15] J. Lee, M. Sato y T. Boku, «OpenMPD: A Directive-Based Data Parallel Language Extension for Distributed Memory Systems» de *2008 International Conference on Parallel Processing - Workshops*, 2008, 10.1109/ICPP-W.2008.28.

9 Anexo

9.1 Tabla de Símbolos *pi_ompd.c*

```
ScopeTable # 1.1.1.1

ScopeTable # 1.1.1

ScopeTable # 1.1.2

ScopeTable # 1.1

 0 --> | < x , DOUBLE > |

 3 --> | < t1 , STRUCT_timeval > |

 4 --> | < t2 , STRUCT_timeval > |

 6 --> | < argv , Pointer Symbol , CHAR > |

 7 --> | < sum , DOUBLE > |

 8 --> | < PI25DT , DOUBLE > |

 13 --> | < pi , DOUBLE > |

 15 --> | < i , UNSIGNED_INT > |

 16 --> | < segundos , DOUBLE > |

 17 --> | < argc , INT > || < factor , INT > |

ScopeTable # 1

 0 --> | < step , DOUBLE > |

 4 --> | < num_steps , STATIC_UNSIGNED_LONG_INT > |

 7 --> | < main , Function Symbol , INT , Parameter List: < argc , IDENTIFIER , INT
>, < argv , IDENTIFIER , CHAR > > |
```

Ilustración 74: Tabla de Símbolos JuliaOMPD.c

9.2 Tabla de Símbolos *particles_OMPD.c*

```
ScopeTable # 1.1.1

ScopeTable # 1.1

 2 --> | < part_num , INT > |

 3 --> | < particles , Pointer Symbol , Particle > |

 6 --> | < argv , Pointer Symbol , CHAR > || < dt , FLOAT > |

 15 --> | < i , INT > |

 17 --> | < argc , INT > |

 23 --> | < step_num , INT > |

 26 --> | < t , INT > |

ScopeTable # 1.2.1

ScopeTable # 1.2

 15 --> | < i , INT > |

 22 --> | < p , Array Symbol , Particle <  Array Size:  >  > |

 25 --> | < nelem , INT > |

ScopeTable # 1

 7 --> | < main , Function Symbol , INT , Parameter List: < argc , IDENTIFIER , INT
>, < argv , IDENTIFIER , CHAR > > |

 20 --> | < Particle , Struct Symbol , TYPEDEF_STRUCT , < x , IDENTIFIER , FLOAT >, <
y , IDENTIFIER , FLOAT >, < z , IDENTIFIER , FLOAT >, < velocity , IDENTIFIER ,
DOUBLE >, < n , IDENTIFIER , INT >, < type , IDENTIFIER , CHAR > > |

 22 --> | < init , Function Symbol , VOID , Parameter List: < p , IDENTIFIER ,
Particle >, < nelem , IDENTIFIER , INT > > |
```

Ilustración 75: Tabla de Símbolos particles_OMPD.c

9.3 Tabla de Símbolos *JuliaOMPD.c*

```
ScopeTable # 1.1
 4 --> | < iter , INT > |
 9 --> | < num_its , INT > || < c , color > |

ScopeTable # 1.2.1

ScopeTable # 1.2
 2 --> | < radius , FLOAT > |
 9 --> | < c , FLOAT_COMPLEX > |
 17 --> | < k , INT > || < modul , FLOAT > |
 20 --> | < n , INT > |
 26 --> | < z0 , FLOAT_COMPLEX > |

ScopeTable # 1.3
 0 --> | < x , INT > |
 1 --> | < y , INT > |
 2 --> | < radius , FLOAT > |
 4 --> | < im , FLOAT > |
 5 --> | < height , INT > |
 9 --> | < c , FLOAT_COMPLEX > |
 14 --> | < width , INT > |
 17 --> | < re , FLOAT > |
 18 --> | < l , INT > |

ScopeTable # 1.4
 0 --> | < x , INT > |
 1 --> | < y , INT > |
 2 --> | < radius , FLOAT > |
 4 --> | < iter , INT > |
 5 --> | < height , INT > |
 9 --> | < c , FLOAT_COMPLEX > |
 14 --> | < width , INT > |
 15 --> | < i , INT > || < count , INT > || < rgb , Pointer Symbol , color > |
 17 --> | < k , INT > |
 26 --> | < z0 , FLOAT_COMPLEX > |

ScopeTable # 1.5.1

ScopeTable # 1.5.2

ScopeTable # 1.5
 3 --> | < tiempo_trans , FLOAT > |
 4 --> | < im , FLOAT > |
 5 --> | < height , INT > |
 6 --> | < argv , Array Symbol , CHAR <  Array Size:  >  > |
 8 --> | < start_time , DOUBLE > |
 9 --> | < c , FLOAT_COMPLEX > |
 10 --> | < tv_end , STRUCT_timeval > |
 14 --> | < width , INT > |
 15 --> | < rgb , Pointer Symbol , color > |
 17 --> | < argc , INT > || < re , FLOAT > |
 21 --> | < tv_start , STRUCT_timeval > |
 23 --> | < end_time , DOUBLE > |

ScopeTable # 1.6
 1 --> | < filename , Pointer Symbol , CHAR > |
 12 --> | < header1 , Array Symbol , UNSIGNED_CHAR <  Array Size: 12 >  > |
 13 --> | < header2 , Array Symbol , UNSIGNED_CHAR <  Array Size: 6 >  > |
 14 --> | < h , INT > |
 15 --> | < rgb , Array Symbol , color <  Array Size:  >  > |
 29 --> | < w , INT > |

ScopeTable # 1
 5 --> | < fcolor , Function Symbol , color , Parameter List: < iter , IDENTIFIER , INT >, < num_its ,
IDENTIFIER , INT > > || < juliaSet , Function Symbol , color , Parameter List: < width , IDENTIFIER , INT >, <
height , IDENTIFIER , INT >, < c , IDENTIFIER , FLOAT_COMPLEX >, < radius , IDENTIFIER , FLOAT >, < iter ,
IDENTIFIER , INT > > |
 6 --> | < tga_write , Function Symbol , VOID , Parameter List: < w , IDENTIFIER , INT >, < h , IDENTIFIER ,
INT >, < rgb , IDENTIFIER , color >, < filename , IDENTIFIER , CHAR > > |
 7 --> | < main , Function Symbol , INT , Parameter List: < argc , IDENTIFIER , INT >, < argv , IDENTIFIER ,
CHAR > > |
 17 --> | < color , Struct Symbol , TYPEDEF_STRUCT , < bl , IDENTIFIER , UNSIGNED_CHAR >, < gr , IDENTIFIER ,
UNSIGNED_CHAR >, < re , IDENTIFIER , UNSIGNED_CHAR > > |
 24 --> | < mapPoint , Function Symbol , FLOAT_COMPLEX , Parameter List: < width , IDENTIFIER , INT >, <
height , IDENTIFIER , INT >, < radius , IDENTIFIER , FLOAT >, < x , IDENTIFIER , INT >, < y , IDENTIFIER , INT
> > |
 25 --> | < explode , Function Symbol , INT , Parameter List: < z0 , IDENTIFIER , FLOAT_COMPLEX >, < c ,
IDENTIFIER , FLOAT_COMPLEX >, < radius , IDENTIFIER , FLOAT >, < n , IDENTIFIER , INT > > |
```

Ilustración 76: Tabla de Símbolos JuliaOMPD.c

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Mon Jun 03 14:06:58 CEST 2024
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)