Departamento de Informática

PROYECTO FIN DE GRADO

Diseño y evaluación de un complemento para refactorización paralela de código C usando OpenMP

Autor: Miguel Olmedo Camacho

Tutor: Luis Miguel Sánchez García

Colmenarejo, 4 de Septiembre de 2012

Resumen

En este proyecto de fin de grado se ha desarrollado un complemento para el IDE Eclipse que servirá para facilitar a los desarrolladores de software el llevar a cabo la creación de aplicaciones que utilicen paralelismo a través de directivas OpenMP. Para ello, ha sido necesario aprender a utilizar la plataforma PDE que proporciona Eclipse para desarrollar complementos. Esta plataforma es muy importante ya que es la manera que tiene la comunidad de Eclipse de colaborar con el IDE y proporcionar nuevas funcionalidades a través de complementos.

El complemento desarrollado dispondrá de una vista donde se podrán añadir ficheros que se quieran refactorizar en el futuro y de una serie de comandos que permitirán al usuario llevar a cabo la refactorización de ficheros de código fuente escritos en lenguaje C. El usuario podrá escoger entre dos modos de refactorización de manera que podrá personalizar algunos aspectos de la misma.

Por otro lado se han analizado una gran cantidad de modelos de programación paralela actuales y se han visto las ventajas y desventajas de cada uno de ellos. El modelo de programación escogido para llevar a cabo la refactorización de ficheros ha sido OpenMP debido a su simplicidad y a que proporciona un gran aumento del rendimiento gracias a las directivas que dispone para ejecutar bucles de forma paralela.

También se han estudiado los distintos patrones que existen para aprovechar el paralelismo en código secuencial y así sacar provecho del hardware actual.

Por último se han redactado una serie de guías para facilitar al usuario la instalación y el uso del complemento desarrollado. También se muestra, de manera resumida, como colaborar con el IDE creando nuevos complementos y así proporcionar nuevas funcionalidades a Eclipse.

Contenido

Índice de figuras

Índice de tablas

Índice de gráficos

1. INTRODUCCIÓN

A continuación se expone una breve introducción donde se explicarán los diferentes conceptos utilizados a lo largo del proyecto de fin de grado, la motivación que ha llevado al desarrollo de la aplicación, los objetivos que se persiguen, la estructura que seguirá el documento en las páginas posteriores y los conceptos básicos utilizados.

1.1 DEFINICIONES Y ACRÓNIMOS

A continuación, se definen las nomenclaturas y los tecnicismos que se utilizan a lo largo del proyecto de fin de grado.

DEFINICIONES

- **Benchmark:** técnica utilizada para medir el rendimiento de un sistema o componente del mismo.
- **Compilador:** programa que traduce un código fuente escrito en un lenguaje de programación a lenguaje máquina.
- **Complemento**: aplicación que se relaciona con otra y que le aporta una funcionalidad nueva, normalmente muy específica.
- **Interprete**: programa que analiza y ejecuta código fuente escrito en lenguaje de alto nivel, traduciéndolo a lenguaje máquina según lo va necesitando.
- **Operación atómica:** modificación de una zona de memoria realizada en una sola operación de bus, siendo imposible que otro proceso escriba o lea dicha zona de memoria hasta que la operación se haya completado.
- **Pragma:** sentencias insertadas en el código fuente que permiten modificar el comportamiento del compilador.
- **RCP**: mínimo conjunto de complementos necesarios en el IDE Eclipse para desarrollar una aplicación con interfaz gráfica.
- **Refactorización:** técnica de ingeniería del software consistente en modificar el código fuente de una aplicación sin cambiar su comportamiento externo.

- **Requisito de usuario funcional:** característica requerida del sistema que expresa una capacidad de acción del mismo.
- **Requisito de usuario no funcional:** característica requerida del sistema, del proceso de desarrollo, del servicio prestado o de cualquier otro aspecto del desarrollo que señala una restricción del mismo.
- **Sección crítica:** parte de código de un programa que accede a un recurso compartido entre varios procesos ligeros y por lo tanto no debe ser accedido por más de un hilo a la vez.

ACRÓNIMOS

Acrónimo	Significado
API	***Application Programming Interface***
AST	***Abstract Syntax Tree***
BCP	***Bloque de Control del Proceso***
CUDA	***Compute Unified Device Architecture***
ICV	***Internal Control Variable***
IDE	***Integrated Development Environment***
JDT	***Java Development Tools***
JVM	***Java Virtual Machine***
MPI	***Message Passing Interface***
NUMA	***Non Uniform Memory Acces***
OpenCL	***Open Computing Language***
PDE	***Plugin Development Environment***
RCP	***Rich Client Platform***
SMP	***Symmetric Multi-Processing***
SIMD	***Single Instruction, Multiple Data***
SISD	***Single Instruction, Single Data***

Tabla 1 - Acrónimos

1.2 MOTIVACIÓN

El desarrollo del hardware está sufriendo un gran cambio en pro de la creación de ordenadores personales de bajo precio pero gran potencia. Hasta hace unos años, la mejora de la velocidad de reloj era el método más utilizado por las industrias para crear nuevos procesadores con capacidades de cálculo cada vez mayores. Por otro lado, se han ido reduciendo los costes de producción de todos los componentes hardware necesarios a medida que los ordenadores personales se convertían en una herramienta cada vez más presente en el día a día de las personas, como puede apreciarse en el Gráfico 1.

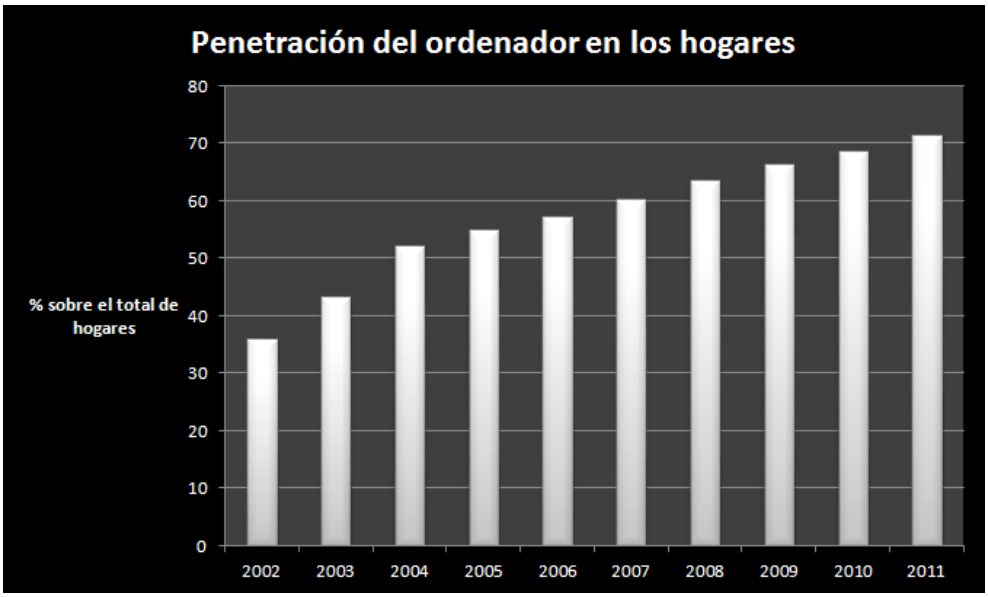


Gráfico 1 - Utilización de los ordenadores personales en los hogares [55]

En el año 2003 el aumento de la capacidad de cálculo de los procesadores se encontró con un muro infranqueable: no se podía seguir aumentando la velocidad de reloj para aumentar su rendimiento. Las causas que impedían continuar con la tendencia hasta entonces era el gran calor que se producía en el procesador, el cuál era difícil de disipar, el consumo de energía cada vez mayor y problemas de estabilidad [55].

En el año 2004 AMD planteó una nueva posibilidad: en vez de tener un procesador ejecutándose a una velocidad de reloj muy rápida, tener varios núcleos idénticos menos potentes en un solo chip los cuales se repartan equitativamente la carga

de trabajo que es necesario realizar en cada instante. De esta manera, se sobrecargan menos los procesadores y se consigue un aumento teórico del rendimiento notable [55]. Este aumento del rendimiento puede apreciarse en el Gráfico 2 donde se comparan distintos tipos de procesadores utilizando el Benchmark PassMark.

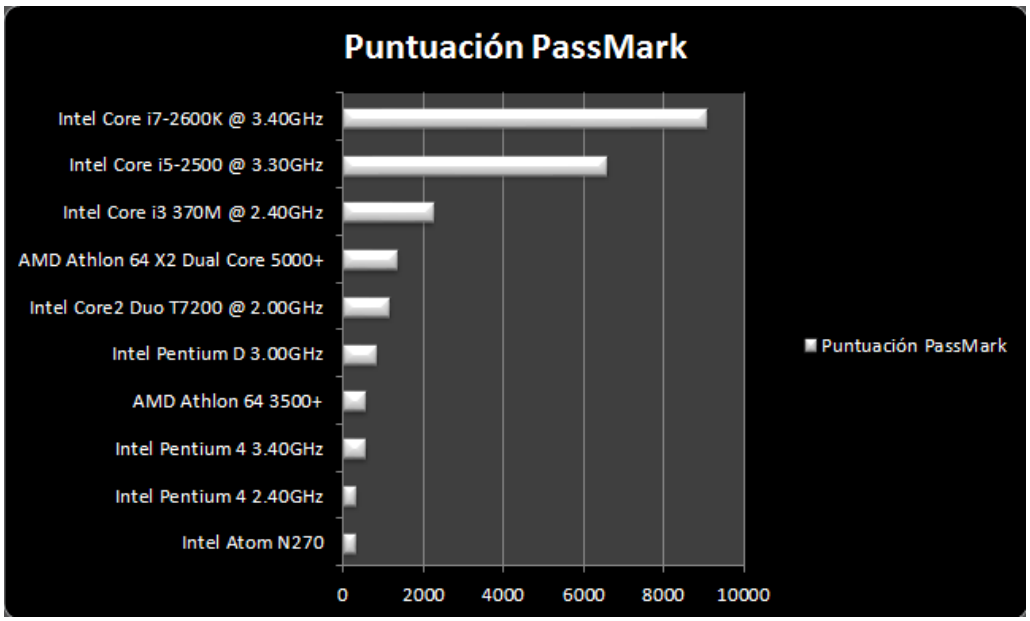


Gráfico 2 - Rendimiento utilizando el Benchmark PassMark [56]

El mayor problema de este nuevo planteamiento para conseguir un rendimiento óptimo es que cualquier cambio de hardware necesita un software que lo acompañe y que esté a la altura. Toda la mejora de rendimiento obtenida gracias a las arquitecturas con varios núcleos no sirve de nada cuando se quiere ejecutar un programa pesado en un solo procesador; la velocidad de ejecución del mismo será igual o peor que en una arquitectura mono núcleo. Por ello fue necesario cambiar la forma de desarrollar software para aprovechar las nuevas características disponibles.

Surgieron nuevas formas de crear programas basadas en la idea de proceso ligero (también denominado hilo) en los cuales se paralelizaban ciertas zonas de código de un mismo proceso. Al principio cada fabricante presentó su propia implementación para aprovechar los hilos [9], por lo que el desarrollo de este tipo de programas no era intuitivo ni portable y los programadores continuaron creando versiones secuenciales, desaprovechando las nuevas arquitecturas.

Posteriormente surgieron unas nuevas bibliotecas y API que pretendían incentivar a los desarrolladores a realizar programas paralelos de una forma mucho más sencilla pero igual de efectiva. De esta manera surgió MPI en el año 1994, Pthreads en el año 1995, OpenMP en el año 1997 y otras interfaces parecidas.

Aun facilitando la vida a los desarrolladores para que con apenas esfuerzo puedan crear software paralelo efectivo, muy poco porcentaje de los programas aprovechan esta capacidad, ya que supone un extra de trabajo. En ocasiones es complicado pensar que zonas de código se pueden realizar en paralelo y cuales no debido a dependencias de datos. Además los desarrolladores deben de tomar precauciones adicionales para que no se produzca un comportamiento del programa inadecuado debido a la compartición entre hilos de variables. Sin embargo, el incremento de velocidad de ejecución obtenida a través de la programación paralela es muy alto y por lo tanto muy recomendable.

En el Gráfico 3 puede apreciarse el incremento del rendimiento a medida que va aumentando la zona de código que se ejecuta de forma paralela. Existe un límite donde la ejecución de un programa no puede repartirse entre más hilos y el rendimiento deja de aumentar.

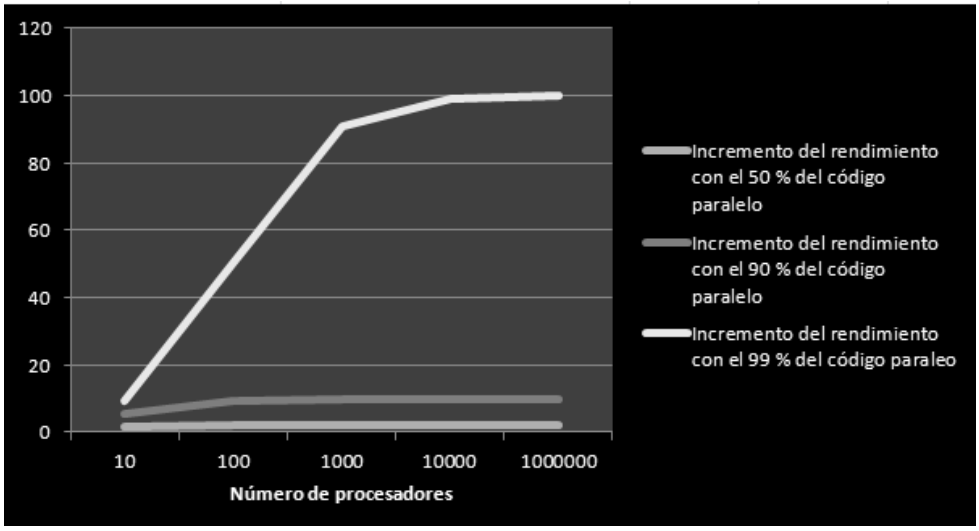


Gráfico 3 - Incremento del rendimiento en programas paralelos [57]

En el proyecto de fin de grado desarrollado se va a crear un complemento para el IDE Eclipse, el más utilizado actualmente. Dicho complemento pretende facilitar aún más la vida a los desarrolladores de software, permitiéndoles desarrollar programas secuenciales en lenguaje C y luego convertirlos en una versión paralela de forma transparente para el usuario sin apenas esfuerzo.

Con el complemento desarrollado las dificultades de crear software paralelo no podrán ser una excusa de nuevo y se podrá incentivar a los desarrolladores a que con un mínimo esfuerzo realicen todas las aplicaciones de forma paralela en vista al futuro.

1.3 OBJETIVOS

El objetivo de este proyecto de fin de grado es **diseñar e implementar un complemento para el IDE Eclipse que permita realizar un análisis estático de código fuente escrito en lenguaje C e insertar las directivas adecuadas de OpenMP que permitan la ejecución paralela de bucles en dicho código.**

Dentro del proyecto se encuentran los siguientes subobjetivos:

- Obtener una visión general de la historia de la programación paralela desde sus inicios hasta la actualidad.
- Estudiar los procesadores de nueva generación y su funcionamiento.
- Analizar y estudiar el funcionamiento interno del IDE Eclipse y de la plataforma de desarrollo de complementos PDE.
- Comprender el funcionamiento interno de OpenMP y de las distintas directivas por las que está formado.
- Estudiar las distintas opciones que existen para paralelizar la ejecución de código.

1.4 ESTRUCTURA DEL DOCUMENTO

En este apartado se detalla cada uno de los capítulos de los que se compone este documento.

En primer lugar se encontrará el capítulo *introducción*, donde se explicará la necesidad de un complemento como el creado en este proyecto para facilitar el desarrollo de software de nueva generación de manera transparente para el programador. También se incluirá un apartado de definiciones y acrónimos utilizados y la definición de los conceptos básicos más importantes que conciernen al proyecto.

En segundo lugar se encontrará el capítulo *estado de la cuestión*, donde se mostrará el estado actual de la programación paralela y las diferentes opciones que existen para ejecutar código concurrente.

En tercer lugar se encontrará el capítulo *análisis y diseño*, donde se describirán los modelos y estructuras que se han seguido en el desarrollo del complemento.

En cuarto lugar se encontrará el capítulo *fases de desarrollo*, donde se explicarán la planificación del proyecto junto con las distintas etapas por las que ha ido pasando el desarrollo del complemento.

En quinto lugar se encontrará el capítulo *conclusiones,* donde se expondrá el conocimiento que se ha obtenido del desarrollo de este proyecto de fin de grado, las futuras mejoras que se podrían implementar en una futura versión del complemento y el presupuesto.

En sexto lugar se encontrará el anexo *guía de instalación*, donde se explicará cómo llevar a cabo la instalación del complemento en el IDE Eclipse.

En séptimo lugar se encontrará el anexo *guía de usuario*, donde se explicará al usuario como utilizar el complemento para realizar la refactorización de ficheros fuente.

En octavo lugar se encontrará el anexo *guía de creación de complementos en Eclipse,* donde se explicará de forma breve como llevar a cabo la creación de complementos para el IDE eclipse.

1.5 CONCEPTOS BÁSICOS

A continuación se describirán diversos conceptos que se han de tener en cuenta a lo largo del proyecto.

Eclipse

Eclipse es un entorno de desarrollo integrado (IDE) de código abierto multiplataforma para desarrollar aplicaciones de cliente enriquecido. Originalmente fue desarrollado en el año 2001 por IBM Canadá como el sucesor de su familia de herramientas para VisualAge.

En noviembre de 2001 se creó el Eclipse Consortium formado por las siguientes empresas: Borland, IBM, Merant, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft y WebGain. La idea del consorcio era promover el desarrollo de Eclipse como software de código abierto.

El Eclipse Consortium se centró en tres proyectos, todos relacionados con el IDE que se quería desarrollar:

- **El proyecto Eclipse**: fue el responsable de desarrollar el banco de trabajo de Eclipse, el complemento de desarrollo de lenguaje Java (JDT) y el entorno de desarrollo de nuevos complementos (PDE) utilizado para ampliar la plataforma.
- **El proyecto de herramientas de Eclipse:** se centró en el desarrollo de nuevas funcionalidades para el IDE.
- **El proyecto de tecnología de Eclipse:** se centró en investigación tecnológica, incubación y educación utilizando la plataforma desarrollada.

En el año 2003, el número de empresas participantes en el consorcio ascendía a 80. En el año 2004 se fundó la fundación Eclipse [42] que actuaba en nombre de la enorme comunidad que se había creado en torno a Eclipse.

Eclipse se diferencia de los demás IDE en varios aspectos fundamentales, que han hecho que se haya desmarcado del resto de entornos que existen en la actualidad como puede apreciarse en el Gráfico 4. Una de ellas es que Eclipse es completamente independiente de la plataforma y del lenguaje. Actualmente existen complementos que permiten utilizar Eclipse como plataforma de desarrollo de lenguaje Java, C/C++, Python, Cobol, PHP, Ruby, C#, HTML, CSS, J2EE, etc. y cada día es compatible con más lenguajes.

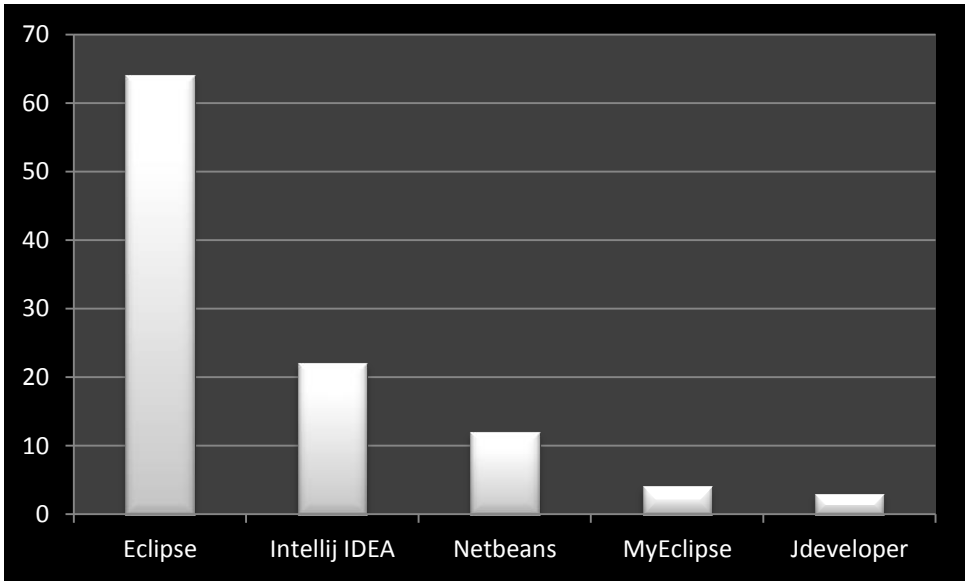


Gráfico 4 - IDE más utilizados por desarrolladores en el año 2011 [26]

Eclipse se ha desarrollado de acuerdo a las necesidades de los usuarios gracias a que son ellos mismos los que colaboran con la plataforma, creando nuevos complementos, resolviendo dudas, mostrando los errores encontrados, etc. La plataforma de desarrollo de complementos es una de la parte más importante de Eclipse y por ello está en continua evolución, intentando facilitar la creación de nuevas funcionalidades para Eclipse a todos los desarrolladores que quieran colaborar [40].

Lenguaje Java

La historia de Java comienza el 15 de enero de 1991, cuando Bill Joy, Andy Bechtolsheim, Wayne Rosing, Mike Sheridan, James Gosling y Patrick Naughton se reúnen en Aspen, Colorado para discutir hacia donde se dirigirá la computación en los años futuros.

Después de hablar acerca de las tecnologías actuales, llegan a la conclusión de que una de las tendencias será el acercamiento de sistemas digitales y electrónica de consumo. Por ello, deciden desarrollar un entorno único que pueda ser utilizado por todos los dispositivos electrónicos de consumo.

Se discutió que lenguaje de programación utilizar en el proyecto, siendo C++ el preferido. Sin embargo, el lenguaje tenía algunas carencias, por lo que se intentó extender y modificar C++, creando el lenguaje C++ ++ --. La idea fue descartada y se decidió crear un lenguaje nuevo desde cero, resultando el lenguaje Oak.

El nuevo lenguaje tendría que ser independiente de la plataforma por lo que se decidió en un principio que sería un lenguaje interpretado. El lenguaje también tendría que ser robusto y sencillo para evitar en lo posible errores humanos en el código. Por ello, se eliminaron las características de otros lenguajes que los hacían más propensos a errores, como la herencia múltiple.

Al final Oak presentaba características parecidas a los lenguajes C, C++ y Objective C pero no estaba ligado a ningún tipo de CPU. Más tarde, se le cambió el nombre debido a que ya existía un lenguaje con ese nombre. Se le llamo Java.

En junio de 1994 se comienza a desarrollar el proyecto “Live Oak”, un pequeño sistema operativo para estudiar las posibilidades de Internet. Al mismo tiempo se comienza a desarrollar un navegador web, denominado en primera instancia “WebRunner” y más tarde “HotJava”.

El 23 de mayo de 1995 en la conferencia SunWorld'95 el cofundador y vicepresidente de Netscape anunciaba que su navegador, el más utilizado en Internet, iba a incorporar Java.

Con la segunda *alpha* de Java, se añade soporte para Windows NT y en la tercera para Windows 95. En enero de 1996, Sun crea JavaSoft y aparece la versión 1.0 de la JDK [23].

Una de las características más importantes de Java es la máquina virtual (JVM). La JVM es el entorno donde se ejecutan los programas Java y es la herramienta que permite que sea totalmente portable. Las funciones principales de la JVM son las siguientes:

- Reservar espacio en memoria para los objetos creados.
- Liberar la memoria no usada.
- Asignar variables a registros y pilas.
- Realizar llamadas al sistema operativo.
- Vigilar que se cumplen todas las normas de seguridad de las aplicaciones.

Una de las principales ventajas de Java reside en que es un lenguaje compilado e interpretado. En un primer lugar, se compilan los ficheros con el código fuente en un conjunto de instrucciones denominados *bytecodes*. A continuación estas instrucciones son interpretadas por la JVM en un ordenador específico. El proceso puede apreciarse en la Figura 1.

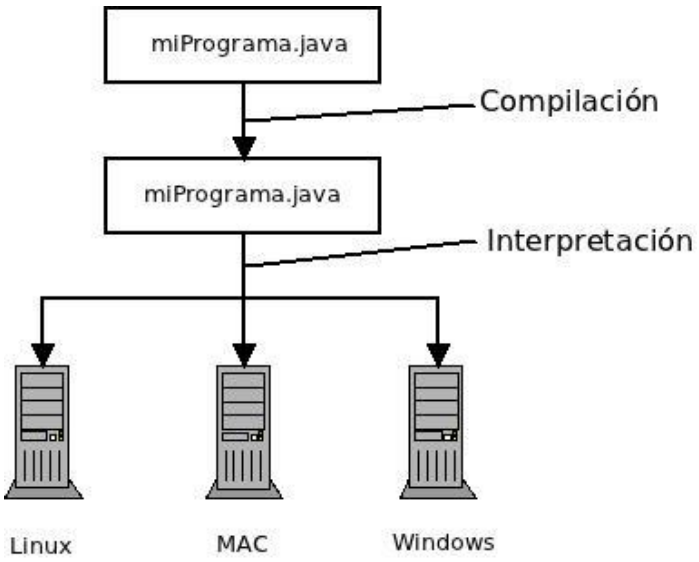


Figura 1 – Compilación e interpretación de lenguaje Java

Lenguaje C

El lenguaje C fue creado a partir del lenguaje B desarrollado por Ken Thompson en los Laboratorios Bell en 1970. El lenguaje B surgió para recodificar el sistema operativo UNIX que hasta esa fecha se programaba en ensamblador, lo que provocaba que estuviese vinculado a máquinas concretas según el juego de instrucciones utilizado. Era necesario un lenguaje que permitiese al programador abstraerse de la capa hardware y conseguir una mayor portabilidad del código.

Entre los años 1970 y 1972 Ritchie estuvo desarrollando junto con Brian Kernighan un lenguaje que permitiese realizar una programación estructurada con un gran conjunto de operadores y tipos de datos para programar tanto en alto como en bajo nivel sin necesidad de cambiar de lenguaje. Surgió el lenguaje C.

Las características del lenguaje C son las siguientes:

- Su núcleo está escrito en lenguaje simple e incluye funciones matemáticas y de manejo de archivos a través de bibliotecas.
- Es un lenguaje flexible que permite programar de manera estructurada y no estructurada.
- Permite el acceso a la memoria de bajo nivel mediante el uso de punteros.
- Presenta un conjunto reducido de palabras reservadas.

- El paso de parámetros a funciones se realiza por valor no por referencia.
- Contenía tipos de datos agregados que permitían combinar bajo un mismo tipo datos de distintos tipos y manejarlos como uno solo.

En 1978 Kernighan y Ritchie publican el libro *The C Programming Language*, el cual contenía la descripción del lenguaje C y todas las posibilidades que presentaba.

El lenguaje C fue muy popular a partir de los años 80 porque fue desplazando a BASIC. Además, cada vez eran más populares los compiladores de C y comenzaron a usarse en los IBM PC. A su vez, Bjarne Stroustrup comenzó a desarrollar C++ complementando C con clases, funciones virtuales, tipos genéricos y expresiones de ADA. Este lenguaje permitía combinar la programación estructurada de C con la orientada a objetos.

El lenguaje C se ha utilizado mucho para la programación en sistemas UNIX aunque también se ha utilizado en el desarrollo de sistemas operativos como Windows o GNU/Linux y en el desarrollo de aplicaciones de escritorio. También es muy común utilizar este lenguaje en sistemas empotrados de tiempo real o como base de kits de desarrollo de micro controladores.

Proceso

Un proceso se define como un programa en ejecución. Se puede definir de una manera más precisa como la unidad de procesamiento gestionada por el sistema operativo. No hay que confundirlo con el concepto de programa, el cual solamente es un conjunto de instrucciones máquina.

Durante la ejecución de un proceso se van modificando los registros del modelo de programación de la computadora de acuerdo a las instrucciones máquina por las que está formado el programa.

Por cada proceso que ejecuta el sistema operativo se mantiene una serie de estructuras de información que permiten identificar cada proceso de manera unívoca y

obtener los recursos que tiene asignado cada uno de ellos. Dentro de las estructuras de información almacenadas se encuentra el BCP, el cual contiene la siguiente información:

- **Información de identificación:** permite identificar al usuario y al proceso.
- **Estado del procesador:** contiene los valores iniciales del estado del procesador o su valor en el momento en que fue interrumpido el proceso.
- **Información de control del proceso:** contiene información que permite gestionar el proceso. Se encuentran los siguientes datos:
    - Estado del proceso.
    - Evento por el que espera el proceso.
    - Prioridad del proceso.
    - Información de planificación.
    - Descripción de los segmentos de memoria asignados al proceso.
    - Recursos asignados entre los que se encuentran los archivos abiertos y los puertos de comunicación.
    - Punteros para estructurar los procesos.
    - Comunicación entre procesos.

Cuando se inicia el sistema operativo se crea un proceso a partir del cual se procederá a crear el resto de procesos cuando sea necesario. De esta manera, se mantiene una jerarquía de procesos en forma de árbol que en sistemas UNIX se mantiene de forma explícita, como puede verse en la Figura 2.

Los procesos pasan por diferentes estados en función de si están ejecutándose, están esperando a que el sistema operativo los ejecute, se encuentran realizando una operación de entrada/salida o están suspendidos. Se puede observar las diferentes transiciones entre estados en la Figura 3. Los estados son los siguientes:

- **En ejecución**: el proceso está siendo ejecutado por el procesador. En esta fase el estado del proceso reside en los registros del procesador.

- **Bloqueado**: el proceso está bloqueado esperando a que ocurra un evento que haga que cambie de estado.
- **Listo**: el proceso está listo para ejecutar pero el procesador está siendo ocupado por otro proceso.
- **Suspendido**: los datos del proceso se mantienen en la zona de intercambio para dejar suficiente memoria a los procesos no suspendidos para ejecutarse correctamente.

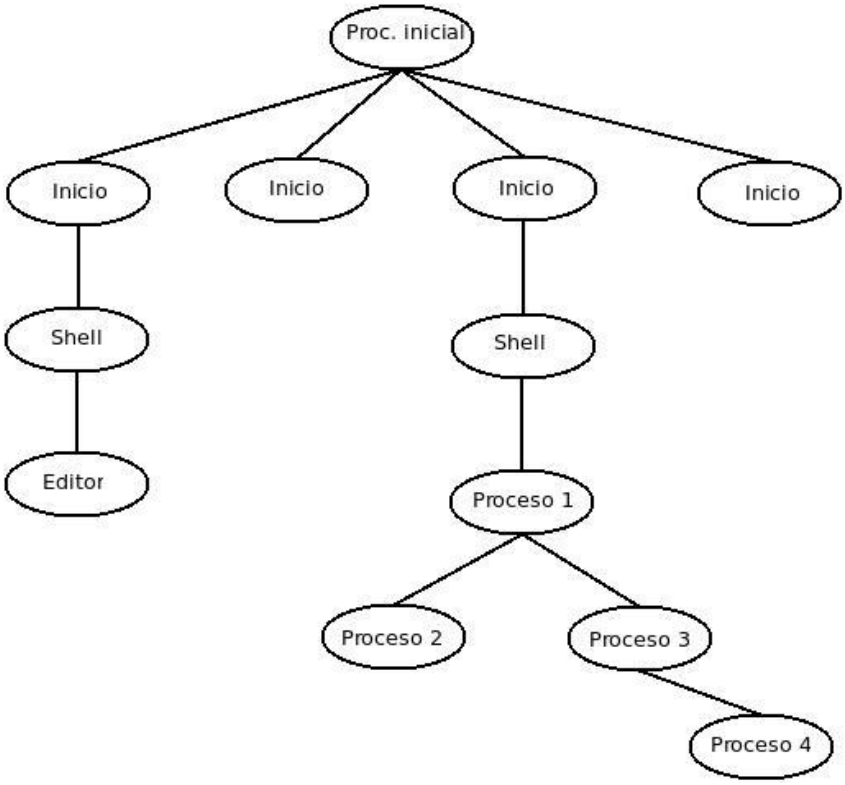


Figura 2 – Jerarquía de procesos [13]

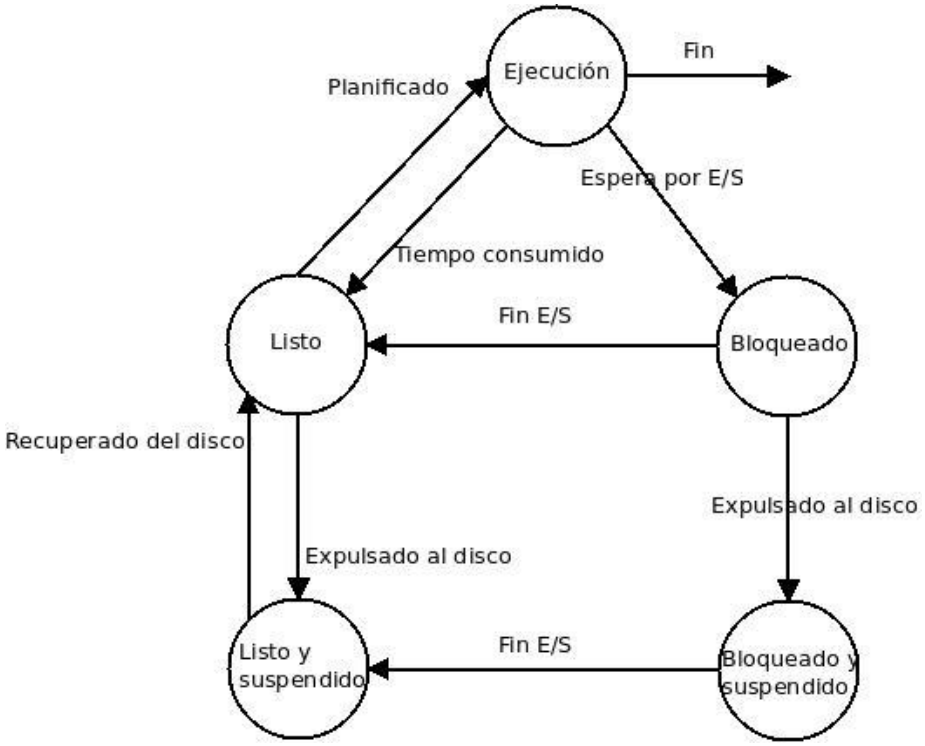


Figura 3 – Estado de los procesos y transiciones entre ellos [13]

Hilo

Un hilo, proceso ligero, subproceso o *thread* es la unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo. Normalmente un proceso está formado por uno o más hilos y todos ellos comparten los siguientes recursos [13]:

- Espacio de memoria.
- Variables globales.
- Archivos abiertos.
- Procesos hijos.
- Temporizadores.
- Señales y semáforos.
- Contabilidad.

Por otro lado, cada hilo tiene información propia entre la que destaca los siguientes recursos:

- Contador de programa.

- Pila.
- Registros.
- Estado del hilo.

El estado de un proceso será el resultado de la combinación de todos los hilos que lo forman. Por lo tanto, si un hilo se encuentra en estado de ejecución, el proceso estará en ejecución; si ninguno está en ejecución y uno está listo para ejecutar, estará listo.

Los hilos permiten paralelizar un programa. Para ello, hay que dividir el programa principal en tareas que puedan ejecutarse de forma independiente y asignarlas a los procesos ligeros. La concurrencia existe al disponer de procesadores con varios núcleos, pudiéndose ejecutar un proceso ligero en cada uno de ellos.

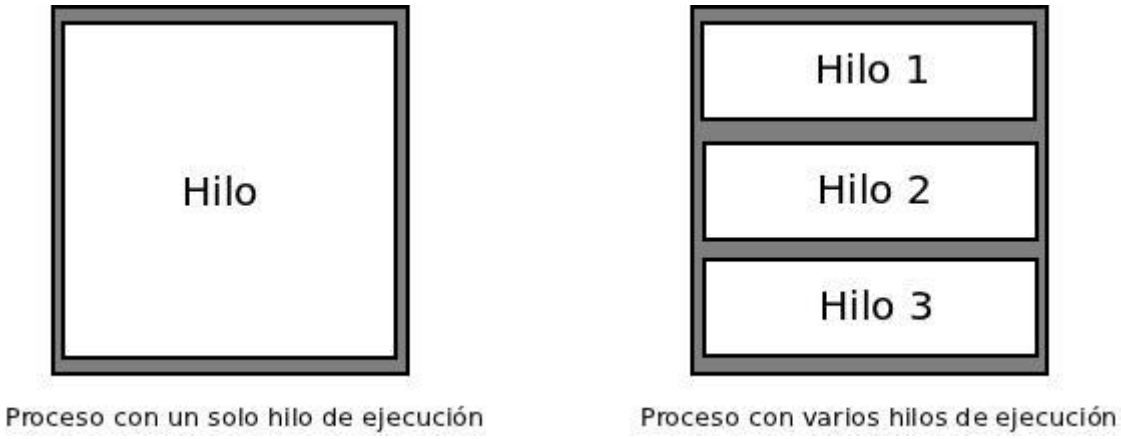


Figura 4 – Proceso con varios hilos

Los hilos presentan muchas ventajas de diseño respecto a los procesos:

- Permite la separación de tareas, encapsulando cada una de ellas en un hilo diferente.
- Facilita la modularidad al dividir los trabajos más complejos en pequeñas tareas independientes.
- Aumenta la velocidad de ejecución, aprovechando los tiempos de bloqueo de algunos procesos ligeros para ejecutar otros cuando se encuentran en un

mismo procesador, cosa que sería imposible si un proceso estuviese formado por un solo hilo.

Sin embargo, la programación concurrente presenta complejidad, ya que los distintos hilos existentes accederán a variables compartidas y es necesario garantizar que el acceso se realiza de forma ordenada. Para controlar el acceso a variables compartidas, existen mecanismos de sincronización de hilos, entre los que se encuentran los cerrojos, los semáforos y las variables condicionales.

Los semáforos son tipos abstractos de datos que pueden tener como valor inicial cualquier número positivo y que tienen asociados una lista de procesos [18].

Los hilos pueden ejecutar dos operaciones atómicas sobre los semáforos:

- wait()
  - Reduce el valor del semáforo. Si el valor se hace negativo, el hilo que ejecuta la operación se bloquea.
- signal():
  - Incrementa el valor del semáforo. Si el valor es positivo, se desbloquea un proceso.

Los cerrojos son iguales que los semáforos pero solo pueden tomar dos valores: 0 y 1. Las operaciones atómicas que pueden ejecutar los hilos sobre los cerrojos son las siguientes [13]:

- lock():
  - El hilo que ejecuta la operación adquiere el cerrojo, garantizando que ningún otro hilo ejecute el código que se encuentra posteriormente hasta que el cerrojo sea liberado.
- unlock():
  - El hilo que ejecuta la operación libera el cerrojo, permitiendo a otro proceso ligero adquirirlo.

Cuando un proceso ligero quiere ejecutar una sección crítica la cual está bloqueada por otro hilo, tendrá que esperar a que el hilo que llegó en primer lugar mande una señal. Con este objetivo se utilizan las variables condicionales. Existen dos operaciones atómicas que se pueden ejecutar sobre variables condicionales [13]:

- c_wait():
  - Bloquea al proceso que ejecuta la llamada.
- c_signal():
  - Desbloquea uno o varios procesos ligeros suspendidos en una variable condicional. Los procesos despertados competirán por el cerrojo según la política utilizada.
  -

Tarea

Una tarea es una unidad de trabajo que se asigna a un hilo determinado en el sistema y cuya ejecución puede ser aplazada hasta que se disponga de los recursos necesarios para ejecutarse. Las tareas están compuestas por los siguientes elementos:

- Código a ejecutar.
- Datos de entorno.
- Variables internas de control (ICV).

Un proceso formado por varios hilos estará formado a su vez por múltiples tareas que tienen que dividirse entre los procesos ligeros. Si existen más tareas que hilos, cada proceso ligero deberá de ejecutar más de una tarea. Los hilos son capaces de suspender la tarea que se encuentra actualmente en ejecución, ejecutar otra y más tarde terminar de ejecutar la primera tarea.

2. ESTADO DE LA CUESTIÓN

A continuación, se describe la situación actual en la que se encuentra los diferentes estándares de programación paralela que han surgido a lo largo de los últimos años. También se comentan los intérpretes y compiladores para código paralelo que existen.

2.1 MODELOS DE PROGRAMACIÓN PARALELA

Los modelos de programación paralela son un conjunto de herramientas software que permiten realizar la ejecución paralela de procesos utilizando bibliotecas invocadas desde los programas tradicionales, extendiendo lenguajes de programación o utilizando modelos de ejecución completamente nuevos.

Los modelos de programación paralela se dividen en los siguientes tipos:

2.1.1 MEMORIA COMPARTIDA

Los sistemas de memoria compartida están formados por múltiples procesadores que comparten un mismo espacio de memoria y que se comunican entre ellos escribiendo y leyendo variables compartidas [54].

Dentro de los sistemas de memoria compartida existen dos tipos diferentes: los SMP y los sistemas NUMA.

En los sistemas SMP todos los procesadores comparten una conexión común a la memoria y el acceso a cualquier zona de memoria se realiza a la misma velocidad. Estos sistemas son los más simples a la hora de realizar ejecuciones paralelas de programas debido a que no hay que preocuparse de la localización de los datos dentro de la memoria compartida. Sin embargo, los sistemas SMP no escalan bien y están limitados a un pequeño número de procesadores. El esquema de un sistema SMP puede verse en la Figura 5.

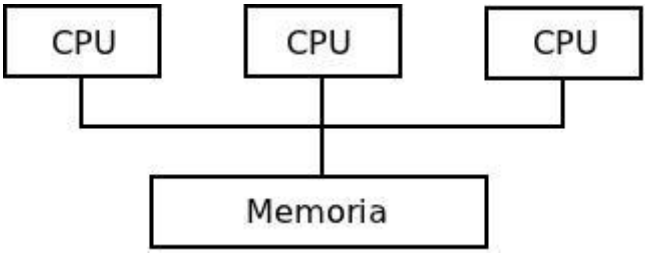


Figura 5 – Sistema SMP

En los sistemas NUMA la memoria es compartida pero está distribuida físicamente de manera que la velocidad de acceso a un determinado bloque de memoria por parte de un procesador dependerá de la lejanía o de la cercanía de dicha memoria. De esta manera se consigue disminuir el cuello de botella generado por el ancho de banda de la memoria, pudiéndose utilizar un mayor número de procesadores que en el caso de SMP. Sin embargo, el tiempo de acceso de un procesador a una determinada posición de memoria puede variar mucho en función de su posición. En la Figura 6 puede verse de forma esquemática un sistema NUMA.

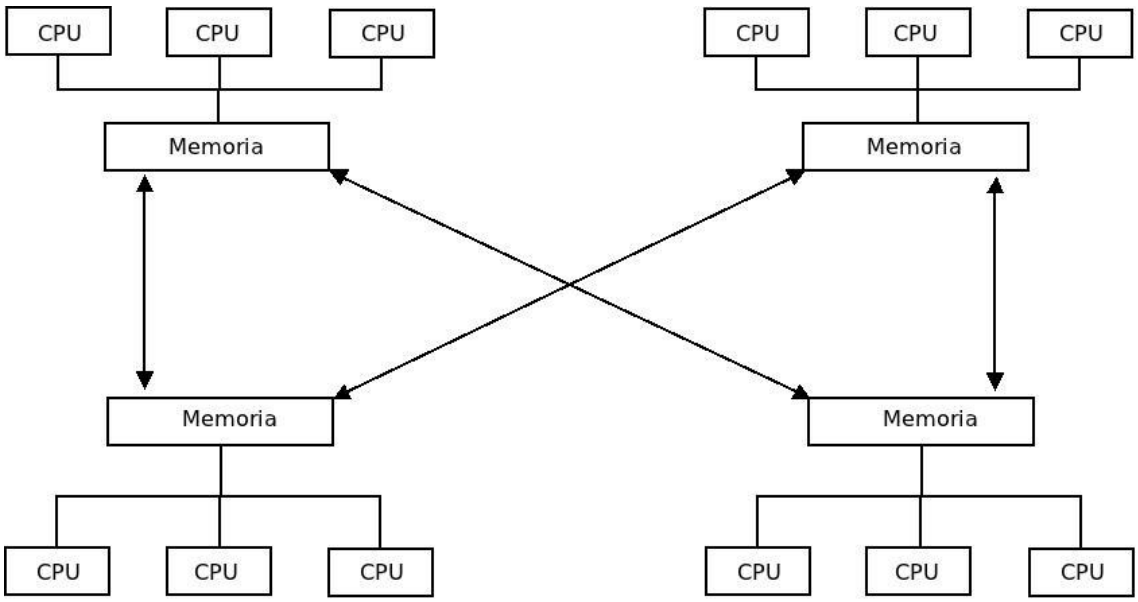


Figura 6 – Sistema NUMA

Existen multitud de formas de crear y ejecutar programas paralelos en este tipo de sistemas:

Pthreads

Pthreads es una interfaz creada en el año 1995 para el lenguaje de programación C en sistemas UNIX que permite utilizar procesos ligeros en sistema SMP para implementar paralelismo. Al inicio de la programación paralela cada vendedor sacó al mercado su propia API para el manejo de hilos, de manera que la portabilidad entre los programas era nula. Pthreads surgió como un intento de estandarización del manejo de los procesos ligeros, según el estándar IEEE POSIX 1003.1c.

La API está formada por diversas subrutinas que se pueden agrupar en cuatro grupos principales:

- Manejo de hilos:
  - Pertenecen a esta categoría las rutinas que trabajan directamente con los hilos (creación, manejo y destrucción).
- Exclusión mutua:
  - Pertenecen a esta categoría las rutinas que tienen que ver con la sincronización de hilos y el acceso a secciones críticas.
- Variables de condición:
  - Pertenecen a esta categoría las rutinas que tienen que ver con la comunicación entre hilos que comparten una zona de exclusión mutua.
- Sincronización:
  - Pertenecen a esta categoría las rutinas que tienen que ver con el manejo de cerrojos y barreras.

Al crear los hilos se le deberá asignar a cada uno una función a ejecutar. A partir de ese momento, los hilos se convertirán en independientes, de manera que podrán crear otros hilos y asignarles nuevas tareas.

Se pueden crear dos tipos diferentes de hilos a través de esta API: los hilos de tipo *joinable* y los hilos *detached*. Los hilos de tipo joinable es necesario que el padre que los ha creado recoja el estado de terminación de los hijos; hasta que no terminen él no terminará. Por lo tanto, los hilos de tipo joinable tienen un punto de sincronización en el hilo padre. Sin embargo, el padre de los hilos de tipo detached no necesita recoger el estado de terminación de sus hijos; de esta forma el padre podrá terminar mientras sus hijos siguen ejecutando [9]. En la Figura 7 se ve un pequeño ejemplo de creación y terminación de hilos:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
	long tid;
	tid = (long)threadid;
	printf("Hola Mundo! Soy el hilo %ld!\n", tid);
	pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
	pthread_t threads[NUM_THREADS];
	int t;
	for(t = 0;t < NUM_THREADS; t++){
		pthread_create(&threads[t], NULL, PrintHello, (void *)t);
	}
	for(t = 0; t < NUM_THREADS; t++) {
		pthread_join(threads[t], NULL);
	}
	pthread_exit(NULL);
}
```

Figura 7 - Ejemplo de código de Pthreads

OpenMP

OpenMP surgió como una API creada por parte de numerosas empresas con la idea de estandarizar y facilitar el desarrollo de programas paralelos escritos en lenguajes estructurados (Fortran en su inicio, C/C++ y Python actualmente). La idea era facilitar la adaptación de los desarrolladores de software tradicional a las nuevas tecnologías emergentes, como los nuevos procesadores de varios núcleos o los ordenadores de memoria compartida distribuida.

Las ventajas de OpenMP respecto a otros tipos de programación paralela son las siguientes:

- Es fácil de usar.
- Está ampliamente adaptado.
- Es escalable.

Esto hace que esta API sea una de las mejores opciones para llevar a cabo la ejecución paralela de programas. En este proyecto de fin de grado se ha centrado la atención en la ejecución concurrente de bucles, ya que son fácilmente identificables y suelen adaptarse a los patrones típicos de paralelismo.

Para realizar la ejecución concurrente, OpenMP deja en manos del compilador toda la lógica y se basa en el uso de unas directivas (denominadas *pragmas*) para marcar las zonas que se quieran realizar de forma paralela.

El usuario deberá elegir que variables quiere que sean compartidas entre todos los hilos que ejecutarán una zona paralela y que variables quiere que sean privadas.

Cuando una variable se marca como privada, cada proceso ligero realizará una copia local de la misma y solo tendrá valor dentro del hilo. La mayor ventaja de las variables privadas es que el acceso a las mismas será óptimo, ya que normalmente cada hilo guardará la copia en su caché local. Como desventaja, las variables privadas en la zona paralela no tendrán valor inicial y al terminar la ejecución de la zona paralela tampoco tendrán valor final.

Cuando una variable se marca como compartida, cada hilo accede a la posición de memoria donde se encuentra dicha variable de forma ordenada, de manera que se garantiza que no se sobrescriban los valores. Cada hilo no podrá crear una copia local en la caché de las variables compartidas, por lo que el acceso se realizará a memoria principal y por lo tanto será lento. Se recomienda marcar todas las variables posibles como privadas para que el acceso a las mismas sea óptimo.

OpenMP dispone de una cláusula de reducción que es muy útil cuando dentro del bucle se llevan a cabo operaciones de acumulación en una variable, como por ejemplo almacenar las sumas parciales de cada iteración en una variable común. Normalmente dicha variable se marcaría como compartida y los hilos irían accediendo a ella de forma ordenada.

Con la cláusula de reducción lo que se consigue es que cada hilo se cree una copia privada de la variable de reducción, lleve a cabo las operaciones necesarias y una vez terminada la zona paralela se sumarán todos los resultados parciales. La mayor ventaja que presenta esta cláusula es que el acceso que realiza cada hilo a la variable es a la caché local, por lo que es mucho más rápido que si tuviesen que acceder a una variable compartida [14].

La Figura 8 es un pequeño ejemplo de código paralelo en un bucle simple con directivas OpenMP:

```
int main(int argc, char *argv[]) {
	double local, w, pi;
	long i, N;
	N = 100000;
	w = 1.0 / N;
	pi = 0.0;
	long j, k;

#pragma omp parallel for default(none) \
	schedule(static) \
	shared(N,w) private(i,local) \
	reduction(+:pi)
	for(i = 0; i < N; i++) {
		local = (i + 0.5) * w;
		pi += 4.0 / (1.0 + local * local);
	}
	pi *= w;
	return 0;
}
```

Figura 8 - Ejemplo de código de OpenMP

La última versión publicada de OpenMP es la versión 3.0 y fue presentada en mayo del año 2008. Las novedades más importantes de esta versión respecto a la 2.5 son las siguientes [58]:

- Se ha creado un nuevo constructor denominado *task* el cual permite crear tareas de forma explícita.
- Se ha creado un nuevo constructor denominado *taskwait* el cual permite a una tarea esperar a que terminen todos sus hijos.
- Se han modificado las normas a seguir para determinar el número de hilos a ejecutar en una región paralela.
- Se permite que un constructor de bucle paralelo se asocie con más de un bucle interno. El número de bucles asociados se controla a través de la cláusula *collapse.*
- Se ha añadido la nueva directiva de reparto de trabajo *auto*, que permite al constructor asignar cualquier número de iteraciones a los hilos que estén ejecutando el bucle paralelo.
- Se ha añadido la nueva variable de entorno *thread-limit-var* que permite definir el máximo número de hilos que ejecutarán una zona paralela.
- Se ha añadido la nueva variable de entorno *max-active-levels* que permite controlar el máximo número de zonas activas paralelas anidadas.
- En esta versión los cerrojos pertenecen a tareas no a hilos.

StarSS

StarSS es un modelo de programación a nivel de nodo que cuenta con una interfaz natural y está formado por un modelo de ejecución subyacente de flujo de datos cuyo objetivo es permitir la ejecución de un mismo código fuente en cualquier arquitectura, ya sea sistemas SMP, GPU, NUMA, etc.

StarSS está basado en la división del programa en pequeñas tareas y opera sobre una sola dirección de memoria lógica. Las directivas de direccionamiento aportan información acerca de los argumentos de las tareas. El compilador usará dichos

argumentos de forma dinámica en tiempo de ejecución para calcular dependencias y para manejar el espacio físico de memoria.

La Figura 9 muestra un pequeño código en el que se usan las directivas de StarSS [68]:

```
#pragma css task input (n, nt) inout (A[n][n])
void cholesky(int n, float *A, int nt ) {
      if (n < SMALL) { spotrf(…); return;}
      float **Ah;
      int bs= n/nt
                  Ah = allocate_block_matrix();
      convert_to_blocks(n, nt, A, Ah);
      for (k=0; k<NT; k++) {
            cholesky (bs, A[k*NT+k], 2) ;
            for (i=k+1; i<NT; i++) strsm (bs, A[k*NT+k], A[k*NT+i]);
            for (i=k+1; i<NT; i++) {
                  for (j=k+1; j<i; j++) sgemm( bs, A[k*NT+i], A[k*NT+j],
A[j*NT+i]);
                  ssyrk (bs, A[k*NT+i], A[i*NT+i]);
            }
            convert_to_linear(Ah);
#pragma css barrier
            free_block_matrix(Ah)
}
```

Figura 9 – Ejemplo de código de StarSS

Intel Threading Building Blocks

Intel TBB es una biblioteca de C++ desarrollada por Intel que permite simplificar la creación, manejo y destrucción de hilos para mejorar el rendimiento de los ordenadores personales de última generación. La última versión de TBB es la 3.0, publicada el 4 de mayo de 2010.

El objetivo de TBB se basa en la definición de tareas de manera simple, sin tener que preocuparse de la creación explícita de hilos. Al igual que OpenMP, TBB tiene un nivel de abstracción muy alto, de manera que el usuario no tiene que preocuparse de los detalles de bajo nivel como la sincronización de hilos [36].

La nueva versión de TBB incluye los siguientes elementos:

- Uso de C++ 0x.
- Uso de variables de condición.
- Manual de diseño de patrones.
- Nuevo mapeo desordenado concurrente.
- Soporte de Windows para el Visual Studio 2010.
- Más herramientas de depuración.
- Intento de evitar inanición de tareas para trabajos encolados.
- Mayor rendimiento.
- Mejor Usabilidad.

La Figura 10 muestra un ejemplo sencillo de un programa paralelizado a través de Intel TBB [62]:

```
class ApplyFoo {
	float *const my_a;
public:
	ApplyFoo( float *a ) : my_a(a) {}
	void operator()( const blocked_range<size_t>& range ) const {
		float *a = my_a;
		for( size_t i=range.begin(); i!=range.end(); ++i )
			Foo(a[i]);
	}
};
void ParallelApplyFoo( float a[], size_t n ) {
	parallel_for( blocked_range<size_t>( 0, n ),
			ApplyFoo(a),
			auto_partitioner());
}
```

Figura 10 – Ejemplo de código de Intel TBB

Cilk Plus

Cilk plus es una extensión para el lenguaje C y C++ desarrollado por Intel cuya mayor ventaja reside en su simplicidad. Ofrece una forma fácil, rápida y segura de optimizar el rendimiento de programas en procesadores con varios núcleos [5].

Cilk plus permite lo siguiente:

- Escribir programas paralelos usando un modelo simple.
    - Cilk plus utiliza solamente tres palabras clave para realizar la ejecución paralela, de forma que es muy fácil de aprender.
- Utilizar paralelismo de datos usando anotaciones simples en *arrays* las cuales incluyen capacidades de funciones elementales.
- Depurar utilizando un depurador familiar y serial.
- Escalar a sistemas con más de cien procesadores.

En la Figura 11 se encuentra un código muy simple de una función en C que aprovecha el paralelismo creado gracias a Cilk Plus [61]:

```
int fib (int n)
{
	if (N <= 2)
		return n;
	else {
		int x, y;
		x = _Cilk_spawn fib(n-1);
		y = fib(n-2);
		_Cilk_sync;
		return x+y;
	}
}
```

Figura 11 – Ejemplo de código utilizando Cilk Plus

SIMD

Es una técnica mediante la cual los procesadores pueden conseguir paralelismo a nivel de datos. El paralelismo que se consigue mediante SIMD se explota cuando se dispone de un conjunto de datos grande sobre los que hay que aplicar la misma instrucción. La Figura 12 ilustra la diferencia entre la técnica de una instrucción para cada dato y la de una instrucción para múltiples datos.

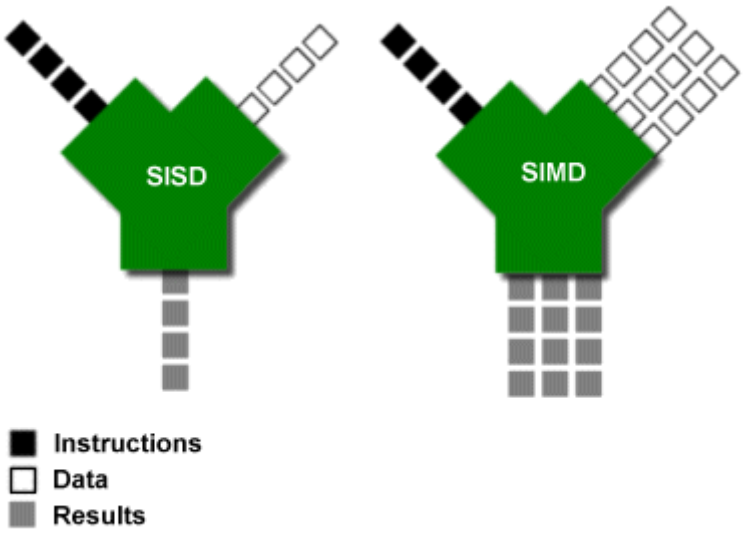


Figura 12 – Arquitectura SISD y SIMD

La unidad básica de datos de un procesador SIMD es el vector. Un vector es un conjunto de datos escalares sobre los cuales se quiere utilizar una misma instrucción. El tamaño de los vectores determina el grado de paralelismo que se realizará en el procesador.

Las operaciones más típicas que se realizan en los procesadores SIMD son las siguientes [39]:

Operaciones entre elementos de diferentes vectores

Esta operación es una de las más básicas. SIMD permite realizar operaciones aritméticas y no aritméticas entre diferentes elementos de varios vectores y almacenar el resultado en un tercer vector. Para conseguir un paralelismo a nivel de datos, todas las

operaciones aritméticas realizadas sobre los diferentes elementos de los vectores deben de ser la misma, de manera que la misma instrucción se aplicará a todos los datos.

Operaciones entre elementos de un mismo vector

SIMD permite realizar operaciones aritméticas y no aritméticas entre diferentes elementos de un mismo vector, almacenando el resultado en un segundo vector acumulativo. El paralelismo se consigue al ejecutar la misma operación sobre todos los datos existentes.

Existen múltiples modelos de programación paralela que utilizan la técnica de SIMD, entre los que se encuentran los siguientes:

CUDA

CUDA es una arquitectura de cálculo paralelo desarrollada por NVIDIA que aprovecha la potencia de la GPU de las tarjetas gráficas para obtener un incremento del rendimiento del sistema.

En los últimos años se ha pasado de solo utilizar la CPU para realizar cálculos de propósito general a utilizar conjuntamente la CPU y la GPU de las tarjetas gráficas como unidades de procesamiento, consiguiendo un aumento considerable de la potencia de cálculo.

CUDA existe actualmente para los lenguajes de programación C, C++ y Fortran y permite mandar bloques de código a la GPU sin necesidad de utilizar lenguaje ensamblador [31].

La Figura 13 muestra un ejemplo de un programa sencillo escrito en lenguaje C utilizando CUDA [60]:

```
#include "stdafx.h"
#include <stdio.h>
#include <cuda.h>

__global__ void square_array(float *a, int N) {
	int idx = blockIdx.x * blockDim.x + threadIdx.x;
	if (idx<N) a[idx] = a[idx] * a[idx];
}

int main(void) {
	float *a_h, *a_d;
	const int N = 10;
	size_t size = N * sizeof(float);
	a_h = (float *)malloc(size);
	cudaMalloc((void **) &a_d, size);
	for (int i=0; i<N; i++) a_h[i] = (float)i;
	cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
	int block_size = 4;
	int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
	square_array <<< n_blocks, block_size >>> (a_d, N);
	cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
	for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
	free(a_h); cudaFree(a_d);
}
```

Figura 13 – Ejemplo de código utilizando CUDA

OpenCL

OpenCL fue propuesto inicialmente por Apple y fue desarrollado en conjunto con AMD, IBM, Intel y NVIDIA.

Es una API de bajo nivel que permite a los desarrolladores de software crear núcleos de cálculo utilizando un lenguaje similar a C y aprovechar la capacidad de cálculo de las GPU de las tarjetas gráficas. Con OpenCL se consigue un paralelismo de datos y de tareas que permite aprovechar el hardware disponible.

OpenCL está formado por los siguientes elementos [64]:

- Núcleos de cómputo.
  - Unidad básica de cálculo. Similar a una función en C.
  - Paralelismo de datos o de tareas.
- Programa de cómputo.
  - Colección de núcleos de cómputo y funciones internas.
  - Análogo a una biblioteca dinámica.

- Cola de aplicaciones de instancias de ejecución del núcleo de cómputo.
    - Encolados en orden.
    - Ejecución en orden o fuera de orden.
    - Se utilizan eventos.

La Figura 14 se corresponde con un ejemplo de código que utiliza OpenCL [59]:

```
__kernel void vector_add_gpu (__global const float* src_a,
             __global const float* src_b,
             __global float* res,
             const int num)
{
       const int idx = get_global_id(0);
       if (idx < num)
             res[idx] = src_a[idx] + src_b[idx];
}

int main() {
       size_t src_size = 0;
       const char* path = shrFindFilePath("vector_add_gpu.cl", NULL);
       const char* source = oclLoadProgSource(path, "", &src_size);
       cl_program program = clCreateProgramWithSource(context, 1, &source,
&src_size, &error);
       assert(error == CL_SUCCESS);

       error = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
       assert(error == CL_SUCCESS);

       char* build_log;
       size_t log_size;

       clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0, NULL,
&log_size);
       build_log = new char[log_size+1];

       clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, log_size,
build_log, NULL);
       build_log[log_size] = '\0';
       cout << build_log << endl;
       delete[] build_log;

       cl_kernel vector_add_kernel = clCreateKernel(program,
"vector_add_gpu", &error);
       assert(error == CL_SUCCESS);
}
```

Figura 14 – Ejemplo de código utilizando OpenCL

OpenACC

OpenACC es una API desarrollada por Cray, CAPS, NVIDIA y PGI que describe una colección de directivas de compilador para especificar zonas paralelas en código C, C++ y Fortran. La API fue desarrollada para simplificar el desarrollo de programas paralelos.

La principal diferencia que presenta OpenACC frente a otras API similares como OpenMP reside en que las directivas utilizadas permiten ejecutar código paralelo no solo en la CPU de los ordenadores sino también en las GPU de las tarjetas gráficas, consiguiendo un aumento del rendimiento significativo [29].

La Figura 15 muestra un pequeño código donde se utilizan algunas de las directivas de OpenACC [63]:

```
#pragma acc data copy(A, Anew)
while ( error > tol && iter < iter_max )
{
      error = 0.f;

#pragma acc kernels
      for( int j = 1; j < n-1; j++)
      {
            for( int i = 1; i < m-1; i++ )
            {
                  Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                                       + A[j-1][i] + A[j+1][i]);
                  error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
            }
      }
#pragma acc kernels
for( int j = 1; j < n-1; j++)
{
      for( int i = 1; i < m-1; i++ )
      {
            A[j][i] = Anew[j][i];
      }
}

if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
iter++;
}
double runtime = GetTimer();
printf(" total: %f s\n", runtime / 1000.f);
}
```

Figura 15 – Ejemplo de código con OpenACC

2.1.2 MEMORIA COMPARTIDA DISTRIBUIDA

En los sistemas que utilizan memoria compartida distribuida cada nodo que forma parte del sistema está formado por módulos de memoria local independientes que están interconectados entre si.

La comunicación entre los procesos que se encuentran ejecutando en los distintos procesadores se lleva a cabo a través del modelo de paso de mensajes, lo cuál requiere llamadas explícitas a funciones de enviar/recibir y dificulta la programación.

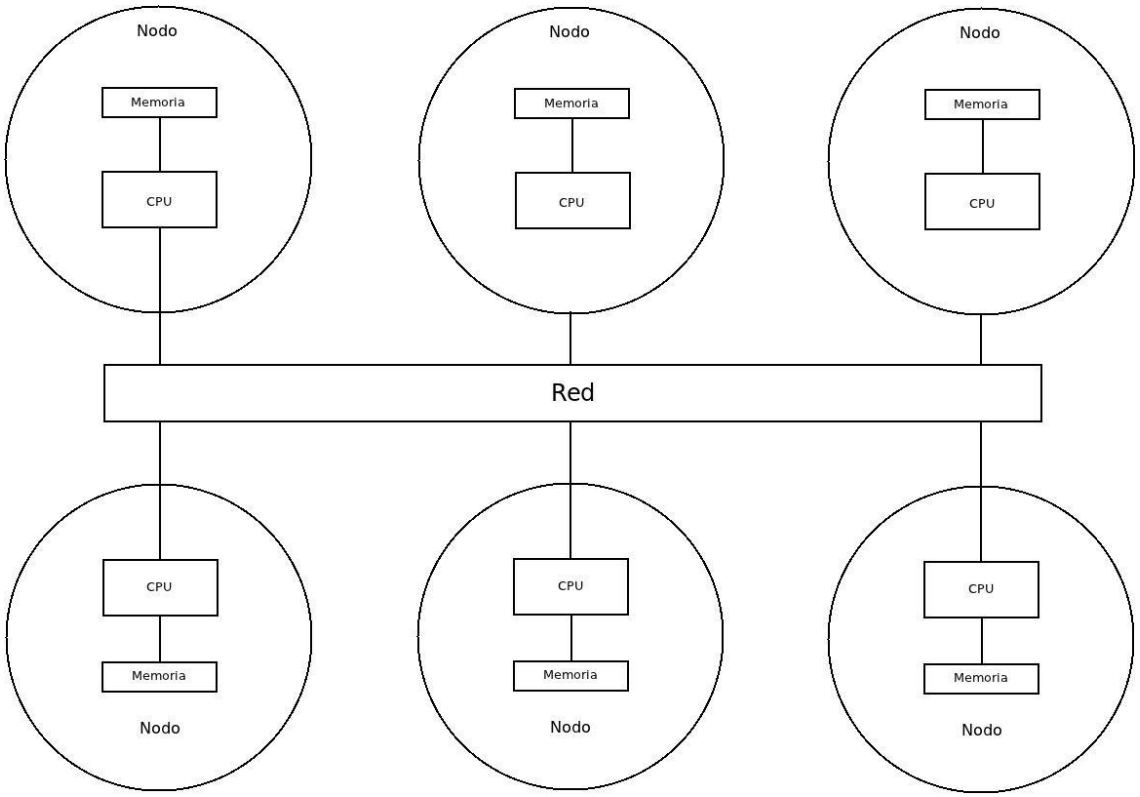


Figura 16 – Memoria compartida distribuida

Para facilitar la programación paralela en este tipo de sistemas existe un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes.

MPI

La interfaz de paso de mensajes es un protocolo de comunicación entre computadoras. Las implementaciones de MPI consisten en un conjunto de bibliotecas que pueden ser utilizadas en programas escritos en lenguaje C, C++, Fortran y Ada. MPI presenta ventajas sobre otras bibliotecas de paso de mensajes, como que los programas que la utilizan son portables y rápidos.

En MPI el número de procesos que se van a utilizar en el programa se calcula antes de la ejecución del mismo, estando cada uno identificado por una variable. Existen cuatro categorias de funciones que se ejecutan sobre MPI:

- Llamadas para inicializar, administrar y finalizar comunicaciones.
- Llamadas para transferir datos entre dos procesos.
- Llamadas para transferir datos entre más de dos procesos.
- Llamadas para crear tipos de datos definidos por el usuario.

MPI presenta las siguientes ventajas:

- Estandarización.
- Portabilidad.
- Buenas prestaciones.
- Amplia funcionalidad.
- Existencia de implementaciones libres.

La Figura 17 muestra un pequeño ejemplo de un programa que utiliza MPI para el paso de mensajes en un sistema de memoria compartida distribuida [67]:

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

#define BUFSIZE 128
#define TAG 0

int main(int argc, char *argv[])
{
	char idstr[32];
	char buff[BUFSIZE];
	int numprocs;
	int myid;
	int i;
	MPI_Status stat;
	MPI_Init(&argc,&argv);
	MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
	MPI_Comm_rank(MPI_COMM_WORLD,&myid);

	if(myid == 0)
	{
		printf("%d: We have %d processors\n", myid, numprocs);
		for(i=1;i<numprocs;i++)
		{
			sprintf(buff, "Hello %d! ", i);
			MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG,
MPI_COMM_WORLD);
		}
		for(i=1;i<numprocs;i++)
		{
			MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD,
&stat);
			printf("%d: %s\n", myid, buff);
		}
	}
	else
	{
		MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD,
&stat);
		sprintf(idstr, "Processor %d ", myid);
		strncat(buff, idstr, BUFSIZE-1);
		strncat(buff, "reporting for duty\n", BUFSIZE-1);
		MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
	}
	MPI_Finalize();
	return 0;
}
```

Figura 17 – Ejemplo de utilización de MPI

2.1.3 MODELOS HÍBRIDOS

Existe otro tipo de sistema que combina los dos tipos de memoria vistas anteriormente: los modelos híbridos. Estos sistemas están formados por varios nodos interconectados a través de una red y a su vez cada nodo está formado por varios procesadores que comparten una misma memoria local.

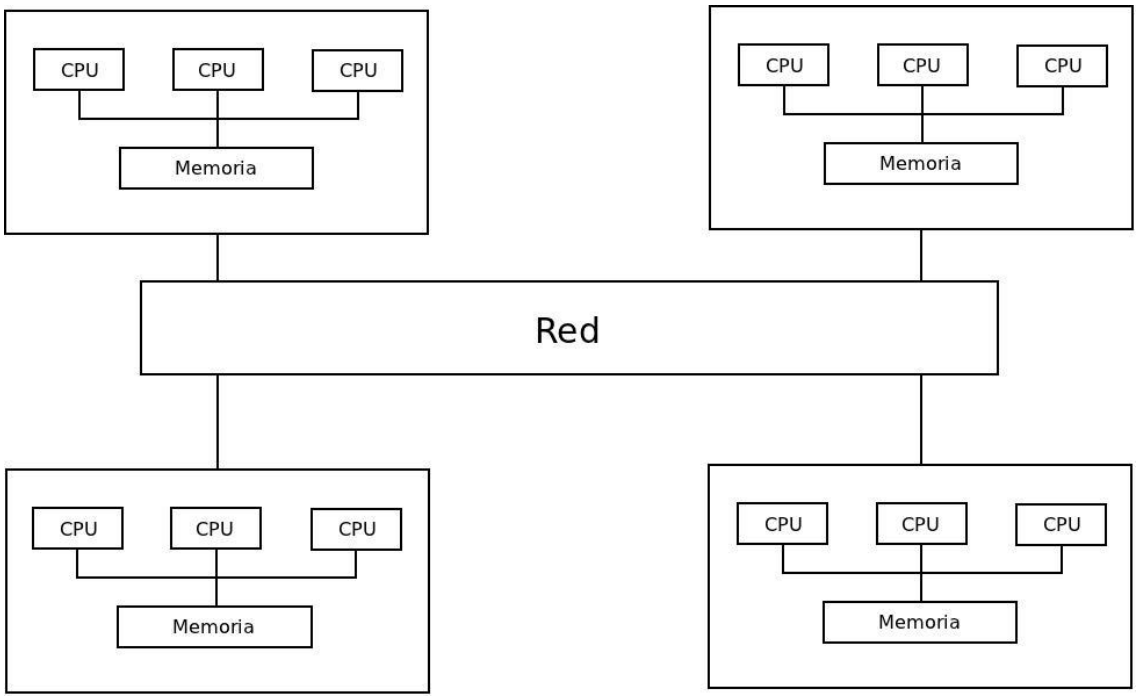


Figura 18 – Modelo híbrido

Este tipo de sistemas complican bastante el desarrollo de software para aprovechar las características que poseen. Es necesario utilizar un modelo de paso de mensajes combinado con un modelo de programación paralela.

Las ventajas que proporciona este modelo son las siguientes:

- Gran escalibilidad gracias a la utilización de modelos de paso de mensajes.
- Uso eficiente de las memorias locales, reduciendo la necesidad de comunicación entre los nodos.
- Alto grado de concurrencia.
- Mejora de la eficiencia.

La Figura 19 muestra un ejemplo de modelo híbrido donde se utiliza MPI junto con las directivas de OpenMP [69]:

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

int main(int argc, char *argv[]) {
	int numprocs, rank, namelen;
	char processor_name[MPI_MAX_PROCESSOR_NAME];
	int iam = 0, np = 1;

	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Get_processor_name(processor_name, &namelen);

#pragma omp parallel default(shared) private(iam, np)
	{
		np = omp_get_num_threads();
		iam = omp_get_thread_num();
		printf("Hello from thread %d out of %d from process %d out of
%d on %s\n",
				iam, np, rank, numprocs, processor_name);
	}

	MPI_Finalize();
}
```

Figura 19 – Ejemplo de código con OpenMP

2.2 INTÉRPRETES Y COMPILADORES

En ocasiones los compiladores e intérpretes clásicos como gcc no presentan algunas características que son necesarias para realizar un análisis más exhaustivo del código fuente y de su estructura. Por ello, se han desarrollado nuevos compiladores e intérpretes que permiten, entre otras cosas, realizar un análisis estático del código fuente y crear árboles sintácticos. Entre los compiladores e intérpretes actuales más interesantes se encuentran los siguientes:

2.2.1 CPARSER

CParser es una biblioteca escrita en lenguaje C++ que permite realizar diferentes operaciones con código fuente escrito en C y C++. CParser permite realizar las siguientes acciones:

- Construir un analizador de código.
- Añadir criterios para buscar determinada semántica en los ficheros.
- Reiniciar el analizador.
- Buscar byte a byte en el código fuente e ir comparando con el criterio a buscar.

Para realizar el análisis del código es necesario proporcionar uno o más criterios de búsqueda. Para crear los criterios, bastará con crear un elemento dentro del código que esté formado por un número identificativo. El analizador buscará dentro del código los criterios marcados y se podrá realizar una llamada a una función definida por el usuario para llevar a cabo las operaciones deseadas [65].

2.2.2 CLANG

Clang es un compilador desarrollado originalmente por Apple [72] para lenguajes C, C++, Objective-C y Objective C-++ que utiliza LLVM como parte del proceso de compilación.

En el año 2005 Apple comenzó a utilizar de forma intensiva LLVM en un gran número de sistemas comerciales. Su intención era incrementar el rendimiento de los ordenadores que disponían de chips GMA de Intel, utilizando LLVM para transformar llamadas a funciones de OpenGL en llamadas más simples que aumentaban la velocidad de ejecución del código.

La intención inicial de Apple fue la de utilizar parte del compilador GCC en combinación con LLVM, pero surgieron problemas debido a la gran complejidad que presentaba el código fuente de GCC. Apple decidió desarrollar su propio compilador, surgiendo el proyecto Clang en julio del año 2007.

Clang ha sido diseñado, al igual que las demás partes de LLVM, en forma de bibliotecas que pueden ser fácilmente integradas en otros ficheros fuente. Otra de las ventajas que presenta es que ha sido desarrollado para obtener mayor información que

GCC durante el proceso de compilación, de manera que permite localizar errores de una forma precisa [66].

Existen proyectos relacionados con Clang que proporcionan nuevas funciones al compilador, entre los que se encuentran los siguientes:

Cling

Cling es un intérprete desarrollado sobre Clang que permite realizar la interpretación de lenguaje C++ de forma interactiva. El proyecto se desarrolló para facilitar la realización de pruebas sobre programas desarrollados en C++.

Utilizando Cling bastará con ir escribiendo código en la línea de comandos y en cuanto se desee se ejecutará y se mostrarán los errores encontrados de manera muy específica y con gran detalle. Además Cling dispone de una serie de comandos que se pueden utilizar para realizar pruebas que permiten nuevas acciones al escribir el código [70].

NVIDIA con Clang

El modelo de programación paralela de NVIDIA, conocido como CUDA, en su última versión ha pasado a utilizar LLVM dentro del compilador de C/C++. LLVM será usado para realizar optimizaciones, para generar código ensamblador para las GPU de NVIDIA (conocido como código PTX) y para generar información de depuración.

El nuevo compilador produce mejor código con mejor tiempos de compilación. Se ha modificado el núcleo de LLVM para que entienda modelos de programación paralela y pueda realizar la compilación de manera satisfactoria.

Debido a la estructura modular de LLVM será posible extender CUDA a nuevos lenguajes y a nuevas arquitecturas diferentes a las GPU de NVIDIA, siendo posible en el futuro utilizar CUDA sobre arquitecturas ARM [71].

2.2.3 PYCPARSER

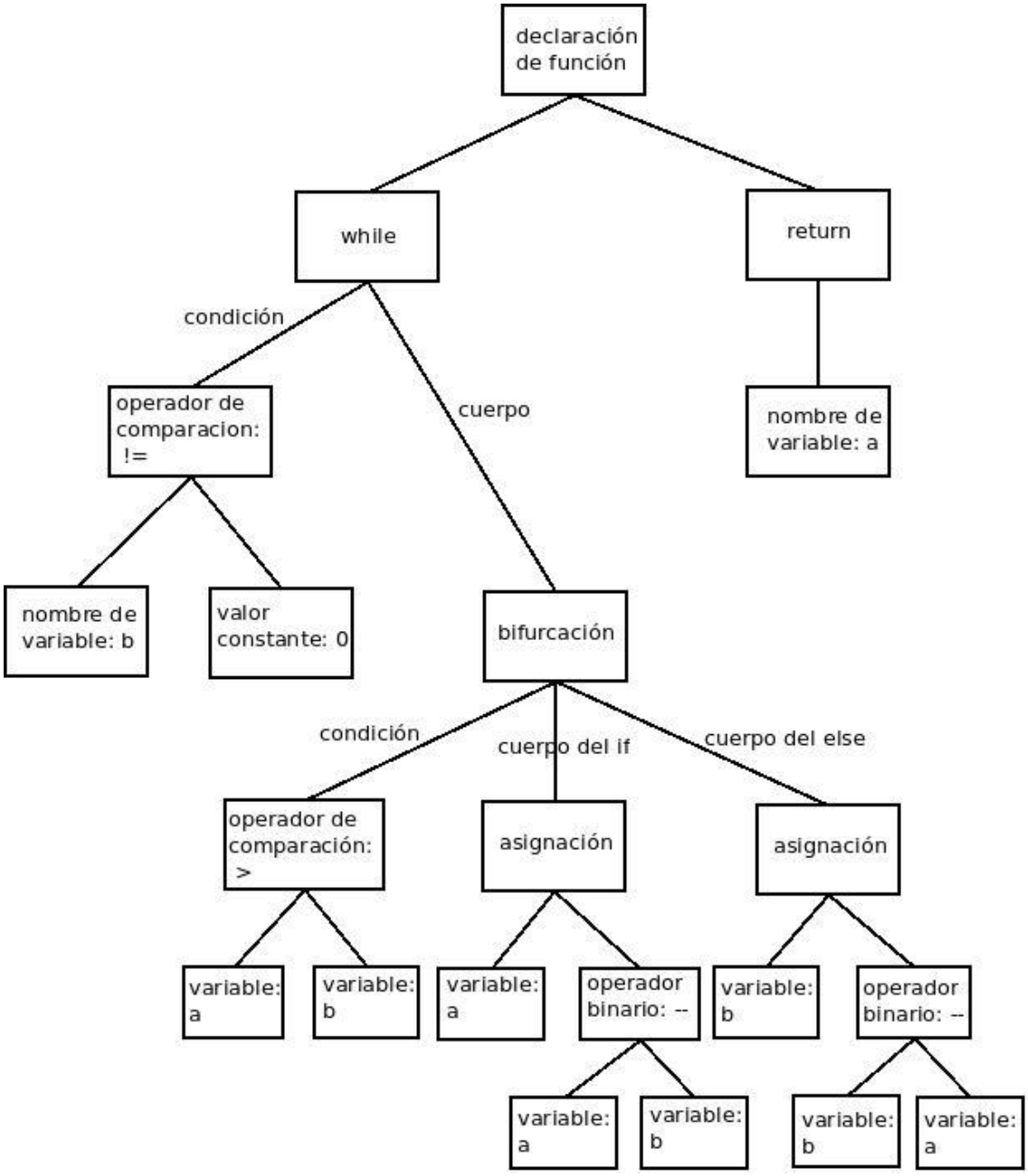
PYCParser es un analizador escrito en Python del lenguaje C. Está diseñado en pequeños módulos para que sea fácilmente integrable dentro de otros programas los cuales necesiten analizar código C en algún momento de su ejecución [10].

2.2.4 AST

Un árbol de sintaxis abstracta es una representación en árbol de la estructura sintáctica de un código fuente escrito en un determinado lenguaje de programación. Cada nodo del árbol se refiere a una sentencia en el código fuente.

La sintaxis es abstracta en el sentido de que no se representa cada detalle que aparece en el código real. Utilizando estas estructuras se puede tener una visión general del flujo de ejecución de un programa y de las diferentes sentencias que lo forman, además de las dependencias entre datos. Es muy común utilizar el árbol de sintaxis abstracta para realizar un análisis estático del código y buscar alguna situación determinada dentro del mismo.

En la Figura 20 se puede apreciar el árbol de sintaxis abstracta del código que se encuentra en la parte inferior de la figura:




```
while b ≠ 0
if a > b
        a := a − b
else
        b := b − a
return a
```

Figura 20 – Árbol de sintaxis abstracta

3. ANÁLISIS Y DISEÑO

A continuación se describen los distintos aspectos de análisis y diseño que se han seguido a lo largo del proyecto para desarrollar el complemento.

3.1 MARCO REGULADOR

Este capítulo recoge las normativas técnicas y legales que afectan al trabajo.

Es necesario contemplar la ley de Protección de Datos de carácter personal [73]. Dado que el proyecto de fin de grado desarrollado es un complemento para el IDE Eclipse que será utilizado en un ámbito personal, la ley no es aplicable en ningún caso.

3.2 REQUISITOS DE USUARIO

En la siguiente sección se van a describir los requisitos de usuario que tiene la aplicación desarrollada. El formato de los requisitos va a ser el siguiente:

Identificador:	
Prioridad: ☐ Alta ☐ Media ☐ Baja	
Necesidad: ☐ Esencial ☐ Deseable ☐ Opcional	
Descripción:	

Identificador:

Para la identificación de los requisitos de usuario se va a usar la siguiente nomenclatura:

RUT-yy, donde:

- **T**: admite los siguientes valores:
  - F: requisito funcional.
  - N: requisito no funcional.
- **yy**: números consecutivos para identificar un requisito.

Prioridad

El campo prioridad será usado para determinar la importancia dentro del desarrollo de los distintos requisitos.

Necesidad

El campo necesidad indicará si el requisito es prescindible para el correcto funcionamiento de la aplicación o si es necesario para que se cumpla con la funcionalidad requerida.

Descripción

En este apartado se definirán de modo sencillo el requisito actual.

A continuación se describen todos los requisitos de usuario del complemento:

Identificador: RUN-01	
Prioridad: ☑ Alta ☐ Media ☐ Baja	
Necesidad: ☑ Esencial ☐ Deseable ☐ Opcional	
Descripción:	El complemento deberá ser compatible con la versión 3.7 de Eclipse (Índigo) y posteriores.

Identificador: RUF-02	
Prioridad: ☑ Alta ☐ Media ☐ Baja	
Necesidad: ☑ Esencial ☐ Deseable ☐ Opcional	
Descripción:	El complemento creará y analizará un árbol sintáctico de los bucles contenidos en el código fuente de ficheros escritos en lenguaje C.

Identificador: RUF-03
Prioridad: ☑ Alta ☐ Media ☐ Baja
Necesidad: ☑ Esencial ☐ Deseable ☐ Opcional

Identificador: RUF-03	
Descripción:	Durante el análisis se observará si existe alguna condición que no permita ejecutar el bucle de forma paralela. Dichas condiciones son las siguientes: • Modificación del número de iteraciones. • Sentencias break. • Falta de llaves en las sentencias compuestas. • Declaración del bucle incorrecta.

Identificador: RUF-04	
Prioridad: ☑ Alta ☐ Media ☐ Baja	
Necesidad: ☑ Esencial ☐ Deseable ☐ Opcional	
Descripción:	En caso de encontrar alguna condición que no permita ejecutar el bucle de forma paralela se informará al usuario a través de una ventana de información. En dicha ventana estará escrita la condición y la línea donde se ha encontrado el fallo.

Identificador: RUF-05	
Prioridad: ☑ Alta ☐ Media ☐ Baja	
Necesidad: ☑ Esencial ☐ Deseable ☐ Opcional	
Descripción:	Después de mostrar al usuario la ventana de información se procederá con el análisis del siguiente bucle encontrado en el fichero.

Identificador: RUF-06
Prioridad: ☑ Alta ☐ Media ☐ Baja
Necesidad: ☑ Esencial ☐ Deseable ☐ Opcional

Identificador: RUF-06	
Descripción:	En caso de que el bucle se pueda ejecutar de forma paralela, se insertará una directiva OpenMP justo antes del bucle. Dicha directiva deberá de contener todas las cláusulas necesarias para que las variables que se encuentren dentro del bucle se asignen a los hilos de forma correcta.

Identificador: RUF-07	
Prioridad: ☑ Alta ☐ Media ☐ Baja	
Necesidad: ☑ Esencial ☐ Deseable ☐ Opcional	
Descripción:	El complemento deberá encargarse de insertar las cláusulas necesarias para que al ejecutar de forma paralela los bucles se aprovechen el máximo de procesadores disponibles.

Identificador: RUF-08	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	Al realizar el análisis del árbol sintáctico, el complemento deberá de buscar posibles operaciones de reducción.

Identificador: RUF-09	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	En caso de encontrar operaciones de reducción, el complemento deberá de incluir la cláusula necesaria para ejecutar dicha operación de forma paralela en la directiva OpenMP.

Identificador: RUF-10

Identificador: RUF-10	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	El complemento dispondrá de un modo de refactorización semiautomático que permitirá al usuario modificar algunos parámetros de la directiva OpenMP.

Identificador: RUF-11	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	El modo de refactorización semiautomático se realizará a través de un asistente formado por varias páginas.

Identificador: RUN-12	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	El modo de refactorización semiautomático permitirá realizar las siguientes acciones: • Modificar el número de hilos que ejecutan la zona paralela. • Modificar el tipo de reparto de trabajo entre los hilos. • Modificar el tamaño de los bloques de iteraciones a repartir entre los hilos. • Activar/Desactivar la búsqueda de operaciones de reducción. • Marcar todas las variables del bucle como compartidas.

Identificador: RUF-13
Prioridad: ☑ Alta ☐ Media ☐ Baja
Necesidad: ☑ Esencial ☐ Deseable ☐ Opcional

Identificador: RUF-13	
Descripción:	Tanto la refactorización automática como la semiautomática se realizarán a través del menú contextual que surgirá al pinchar con el botón derecho sobre un fichero.

Identificador: RUN-14	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☑ Esencial ☐ Deseable ☐ Opcional	
Descripción:	La opción de refactorizar solo saldrá en el menú contextual cuando se pinche con el botón derecho sobre un fichero que contenga código fuente escrito en lenguaje C.

Identificador: RUF-15	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☐ Deseable ☑ Opcional	
Descripción:	Después de insertar las directivas OpenMP en un archivo dicho fichero deberá de guardarse de forma automática.

Identificador: RUF-16	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	El complemento deberá de insertar directivas del preprocesador al inicio del programa para poder ejecutarlo de forma secuencial en caso de no compilar el código fuente con la biblioteca de OpenMP.

Identificador: RUF-17
Prioridad: ☐ Alta ☑ Media ☐ Baja
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional

Identificador: RUF-17	
Descripción:	El complemento dispondrá de una vista donde se podrán ir añadiendo los ficheros que se quieran refactorizar en el futuro.

Identificador: RUN-18	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	La vista dispondrá de las siguientes columnas: • Nombre de los ficheros. • Localización.

Identificador: RUN-19	
Prioridad: ☐ Alta ☐ Media ☑ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	El usuario solo podrá añadir a la vista de refactorización ficheros con código fuente escrito en lenguaje C.

Identificador: RUF-20	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	Los ficheros se añadirán a la vista a través del menú contextual que saldrá al pinchar con el botón derecho sobre un archivo.

Identificador: RUF-21	
Prioridad: ☐ Alta ☐ Media ☑ Baja	
Necesidad: ☐ Esencial ☐ Deseable ☑ Opcional	
Descripción:	El usuario podrá añadir a la vista de refactorización varios ficheros al mismo tiempo.

Identificador: RUN-22	
Prioridad: ☐ Alta ☐ Media ☑ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	Al intentar añadir un fichero a la vista de refactorización se comprobará que dicho fichero no ha sido añadido anteriormente. En caso de ser así, no se permitirá añadir de nuevo el fichero.

Identificador: RUF-23	
Prioridad: ☐ Alta ☐ Media ☑ Baja	
Necesidad: ☐ Esencial ☐ Deseable ☑ Opcional	
Descripción:	Los ficheros que se encuentren en la vista de refactorización se podrán colocar en orden ascendente o descendente según cualquiera de las columnas de la vista.

Identificador: RUF-24	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	El usuario podrá borrar los ficheros que desee que se encuentren en la vista de refactorización.

Identificador: RUF-25	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	El borrado de los ficheros se realizará a través del menú contextual cuando se pinche sobre un archivo que se encuentre en la vista de refactorización.

Identificador: RUF-26	
Prioridad: ☐ Alta ☐ Media ☑ Baja	
Necesidad: ☐ Esencial ☐ Deseable ☑ Opcional	
Descripción:	El borrado de los ficheros también se podrá realizar a través de un botón que se encuentre en la barra de herramientas de la vista de refactorización.

Identificador: RUF-27	
Prioridad: ☐ Alta ☐ Media ☑ Baja	
Necesidad: ☐ Esencial ☐ Deseable ☑ Opcional	
Descripción:	Existirá una opción para llevar a cabo el borrado de todos los ficheros que se encuentre en la vista.

Identificador: RUF-28	
Prioridad: ☐ Alta ☐ Media ☑ Baja	
Necesidad: ☐ Esencial ☐ Deseable ☑ Opcional	
Descripción:	El borrado de todos los ficheros se realizará tanto a través del menú contextual como a través de un botón en la barra de herramientas de la vista.

Identificador: RUF-29	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	La vista dispondrá de dos botones en su barra de herramientas que permitirán realizar la refactorización semiautomática y automática sobre el fichero seleccionado en la vista.

Identificador: RUF-30

Identificador: RUF-30	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	La refactorización de ficheros a través de la vista también se podrá realizar a través del menú contextual al pinchar con el botón derecho sobre un archivo que esté añadido a la vista.

Identificador: RUF-31	
Prioridad: ☐ Alta ☐ Media ☑ Baja	
Necesidad: ☐ Esencial ☐ Deseable ☑ Opcional	
Descripción:	El complemento permitirá realizar la refactorización tanto automática como semiautomática sobre más de un fichero que se encuentre en la vista.

Identificador: RUF-32	
Prioridad: ☐ Alta ☐ Media ☑ Baja	
Necesidad: ☐ Esencial ☐ Deseable ☑ Opcional	
Descripción:	Existirá una acción dentro del menú contextual al pinchar con el botón derecho sobre la vista que permitirá realizar la refactorización automática de todos los ficheros que se encuentren añadidos a la misma.

Identificador: RUF-33	
Prioridad: ☐ Alta ☐ Media ☑ Baja	
Necesidad: ☐ Esencial ☐ Deseable ☑ Opcional	
Descripción:	Existirá una acción dentro del menú contextual al pinchar con el botón derecho sobre la vista que permitirá realizar la refactorización semiautomática de todos los ficheros que se encuentren añadidos a la misma.

Identificador: RUF-34	
Prioridad: ☐ Alta ☐ Media ☑ Baja	
Necesidad: ☐ Esencial ☐ Deseable ☑ Opcional	
Descripción:	Existirá un botón en la barra de herramientas principal de Eclipse que permitirá abrir la vista de refactorización.

Identificador: RUF-35	
Prioridad: ☐ Alta ☐ Media ☑ Baja	
Necesidad: ☐ Esencial ☐ Deseable ☑ Opcional	
Descripción:	En caso de que la vista ya esté abierta, el botón que la abre la enfocará.

Identificador: RUF-36	
Prioridad: ☐ Alta ☑ Media ☐ Baja	
Necesidad: ☐ Esencial ☑ Deseable ☐ Opcional	
Descripción:	Los ficheros que están añadidos a la vista serán persistentes entre diferentes sesiones de Eclipse.

Identificador: RUF-37	
Prioridad: ☐ Alta ☐ Media ☑ Baja	
Necesidad: ☐ Esencial ☐ Deseable ☑ Opcional	
Descripción:	Tras llevar a cabo la refactorización de un fichero a través de la vista el archivo será borrado de la misma.

Identificador: RUN-38
Prioridad: ☑ Alta ☐ Media ☐ Baja
Necesidad: ☑ Esencial ☐ Deseable ☐ Opcional

Identificador: RUN-38	
Descripción:	El complemento estará comprimido en un fichero el cuál contendrá todas las rutas necesarias para extraer el contenido en el directorio raíz de instalación de Eclipse y que funcione correctamente.

3.3 CASOS DE USO

A continuación se muestra un diagrama con los casos de uso que se han derivado de los requisitos funcionales definidos anteriormente.

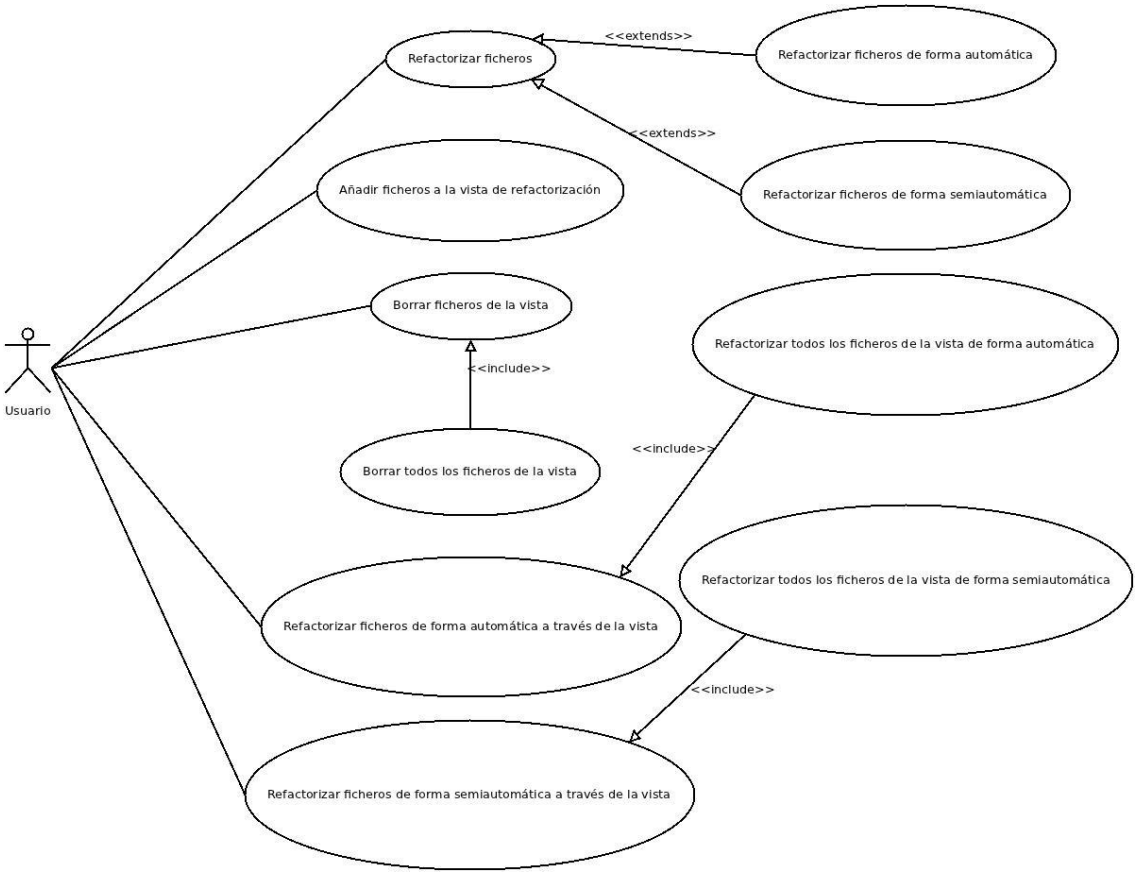


Figura 21 – Diagrama de casos de uso

En el diagrama de casos de uso solo hay un actor que será el usuario que descargue el complemento. Dado que el objetivo del complemento es llevar a cabo la refactorización de ficheros de código fuente escritos en lenguaje C todos los casos de uso están relacionados, directa o indirectamente, con esta finalidad. Los casos de uso son los siguientes:

Caso de uso	Descripción
Refactorizar ficheros de forma automática.	El usuario podrá llevar a cabo la refactorización de ficheros de forma automática a través del menú contextual.
Refactorizar ficheros de forma semiautomática.	El usuario podrá llevar a cabo la refactorización de ficheros de forma semiautomática a través del menú contextual.
Añadir ficheros a la vista de refactorización.	El usuario podrá añadir ficheros a la vista de refactorización siempre que sean ficheros de código fuente escritos en lenguaje C.
Borrar ficheros de la vista.	El usuario podrá llevar a cabo el borrado de ficheros de la vista de refactorización.
Borrar todos los ficheros de la vista.	El usuario podrá borrar todos los ficheros de la vista de refactorización de una vez, dejando la vista limpia.
Refactorizar ficheros de forma automática a través de la vista.	El usuario podrá llevar a cabo la refactorización automática de ficheros añadidos a la vista a través de diversas herramientas proporcionadas por el complemento.
Refactorizar ficheros de forma semiautomática a través de la vista.	El usuario podrá llevar a cabo la refactorización semiautomática de ficheros añadidos a la vista a través de diversas herramientas proporcionadas por el complemento.
Refactorizar todos los ficheros de la vista de forma automática.	El usuario podrá con una sola acción llevar a cabo la refactorización automática de todos los ficheros añadidos a la vista.
Refactorizar todos los ficheros de la vista de forma semiautomática.	El usuario podrá con una sola acción llevar a cabo la refactorización semiautomática de todos los ficheros añadidos a la vista.

Tabla 2 - Descripción de los casos de uso

3.4 PATRONES DE PROGRAMACIÓN PARALELA

En este apartado se van a explicar los diferentes patrones que existen en la programación que se pueden aprovechar para ejecutar código de forma paralela y conseguir un aumento notable del rendimiento.

La creación de un programa paralelo es un proceso incremental [54]. En primer lugar será necesario analizar el código y buscar posibles zonas de ejecución concurrente. Una vez encontradas las zonas a ejecutar de forma paralela habrá que diseñar un algoritmo para realizar la ejecución de las mismas. A continuación habrá que tener en cuenta los detalles de bajo nivel del procesador y optimizar el código para que se ejecute de acuerdo a su estructura. Por último, habrá que escoger un método de implementación de programación paralela. En este proyecto de fin de grado se va a centrar la atención en los dos primeros pasos. Los métodos de implementación de programación paralela ya están definidos en el apartado Modelos de programación paralela.

BÚSQUEDA DE ZONAS CONCURRENTES

Una vez que se haya decidido que se va a paralelizar un programa, el primer paso será la búsqueda de las zonas que se puedan ejecutar de forma concurrente. Dicha búsqueda deberá de seguir una serie de pasos para que se realice de forma satisfactoria. La Figura 22 ilustra el proceso:

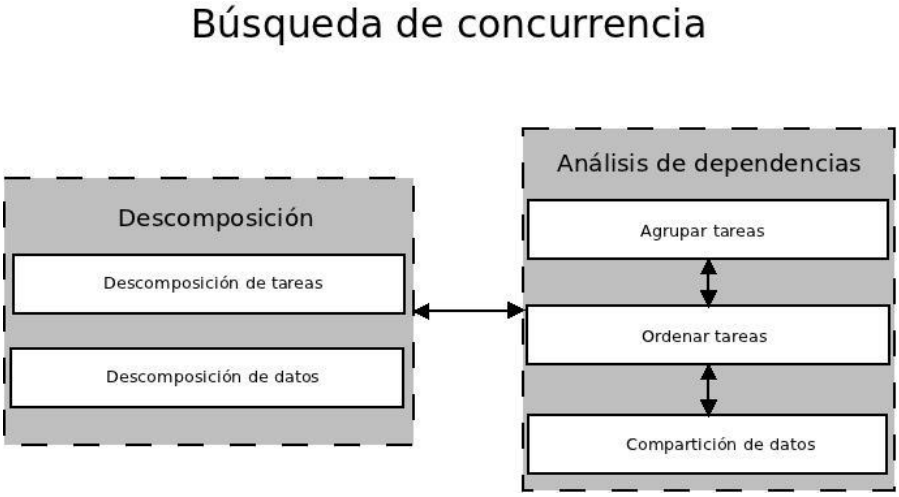


Figura 22 – Proceso de búsqueda de concurrencia [54]

Descomposición de tareas

El primer paso que habrá que realizar será dividir la lógica del programa en pequeñas tareas que se puedan ejecutar de forma concurrente. Para poder realizar esta división es necesario que el programador tenga un profundo conocimiento sobre el software desarrollado para conocer donde se realiza la mayor carga de trabajo y si es posible dividirla en pequeñas tareas.

Descomposición de los datos

El siguiente paso será identificar una manera de descomponer los datos usados en unidades que puedan ser operados de forma independiente. La descomposición de datos está marcada por la necesidad de cada una de las tareas creadas anteriormente. Sin embargo, en un inicio habrá que fijarse en la estructura general del programa y ver como se podría dividir. Unos de los casos más claros a la hora de descomponer los datos son los siguientes:

- **Cálculos con arrays**: cuando se está realizando operaciones con arrays se puede definir la concurrencia al realizar actualizaciones de diferentes partes de los mismos. En arrays multidimensionales se puede dividir por filas, columnas o bloques de diferente tamaño.
- **Estructuras de datos recursivas:** algunas estructuras de datos presentan una división muy clara. Por ejemplo, las estructuras en árbol se pueden dividir en subárboles que pueden actualizarse de forma concurrente.

Agrupación de tareas

Una vez que el programa tiene sus datos divididos en pequeños grupos independientes y su ejecución se encuentra dividida en pequeñas tareas, es necesario analizar ambas características para encontrar grupos de tareas relacionadas que se puedan agrupar para disminuir el tiempo perdido debido a las dependencias que puedan existir entre ejecuciones.

Existirá una dependencia temporal entre tareas cuando una tarea comparta datos con otra y necesite bloquearse hasta que la primera haya terminado de usarlos. En

muchos casos esta dependencia crea un tiempo de espera importante que se puede subsanar si se agrupan ambas tareas y una se ejecuta a continuación de la otra.

Ordenar la ejecución de las tareas

Ahora que las tareas se encuentran agrupadas, es necesario observar si es necesario un orden de ejecución para que el comportamiento del programa sea adecuado.

El orden debe de garantizar que se satisfacen todas las restricciones de ejecución, de manera que el diseño resultante sea correcto. Sin embargo, no debe de ser más restrictivo de lo necesario. Un orden muy restrictivo limita las opciones de diseño y puede disminuir la eficiencia del programa.

Datos compartidos

Cada grupo de tareas utiliza una serie de datos para realizar las operaciones. Si esos datos están asociados solamente con ese grupo de tareas, no habrá ningún problema en el acceso y modificación de los mismos. Sin embargo, existen datos que son comunes y no se pueden englobar dentro de ningún grupo de tareas. Será necesario identificar estas dependencias de datos y realizar un diseño que sea correcto y eficiente. Una vez identificada la dependencia de datos, se puede incluir en uno de los tres grupos siguientes:

- Solo lectura.
    - En estos casos los datos son leídos pero no escritos, por lo que no es necesario modificar el programa.
- Datos locales.
    - Es el caso de la división de un array en pequeñas partes que serán repartidas entre los diferentes hilos. En estos casos no hay que realizar ningún cambio, debido a que cada tarea accederá a una parte del array y no modificará la zona de memoria a la que accederán las demás tareas.
- Lectura y escritura.

- Es el caso más común: varias tareas acceden a un mismo dato y lo modifican. Para estos casos es necesario utilizar algún tipo de mecanismo de exclusión mutua.

Existen dos casos especiales dentro del grupo de datos de lectura y escritura:

- Datos acumulativos.
  - Se trata de datos que se utilizan para acumular un resultado. En este caso cada tarea tiene una copia local de los datos sobre los que realizan las operaciones. Una vez terminada la ejecución de las tareas, se suman los resultados en una variable global.
- Varias lecturas y una sola escritura.
  - En este caso, varias tareas leen datos y una sola los modifica. Para datos de este tipo se crearán dos copias de la variable: una que almacene el valor inicial de la misma y otra que será modificada por la tarea.

ELECCIÓN DEL ALGORITMO

Una vez definidos los grupos de tareas que deben ejecutarse en un programa será necesario marcar como se repartirán entre los procesadores de manera que se aproveche al máximo la capacidad de realizar cálculos paralelos. Se van a ver seis algoritmos básicos, divididos en tres grupos: organizados por tareas, organizados por descomposición de datos y organizados por el flujo de los datos. El proceso puede verse en la Figura 23.

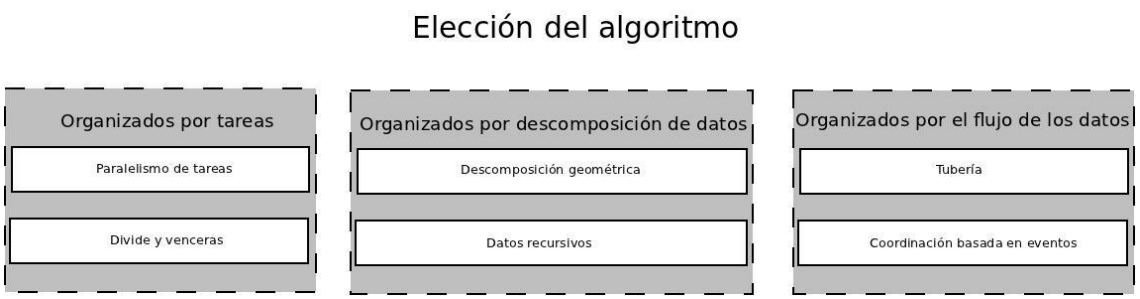


Figura 23 – Elección del algoritmo más adecuado [54]

El algoritmo escogido debe de tener un compromiso entre eficiencia, portabilidad, simplicidad y escalabilidad.

Organizados por tareas

Estos algoritmos son los más adecuados cuando el mejor principio de organización es la ejecución de los grupos de tareas en determinado orden. Los algoritmos de este tipo estarán basados en numerar los grupos de tareas, repartirlos entre los procesadores y ejecutarlos en orden. Existen dos algoritmos dentro de esta categoría: paralelismo de tareas y divide y vencerás.

Patrones de paralelismo de tareas

Es el patrón más sencillo. En este patrón la mejor forma de expresar la concurrencia de un programa es asignar a los procesadores los grupos de tareas a ejecutar de forma concurrente. Será necesario tener cuidado con las dependencias de datos existentes entre los diferentes grupos de tareas. Al usar este patrón, es necesario considerar los siguientes factores:

Tareas

Las tareas en las que está descompuesto el problema principal deben de cumplir con dos condiciones. La primera de ellas es que deben de existir tantas tareas como sea posible; como mínimo deben de existir el mismo número de tareas que de procesadores. En segundo lugar, la carga de trabajo asociada a cada tarea debe de ser suficientemente grande como para que el tiempo introducido debido al manejo de tareas y sus dependencias no sea notable.

Dependencias

Las dependencias entre grupos de tareas pueden tener un gran impacto en el rendimiento de un programa paralelo. Si las dependencias dependen del orden de ejecución de los grupos de tareas bastará con establecer un orden para solucionar el problema. En cambio si las dependencias son datos compartidos se debe de intentar modificar el código de manera que se resuelva la dependencia. En caso de que no se

pueda resolver la dependencia, se deberán de insertar de forma explícita mecanismos para garantizar el acceso a variables de forma ordenada.

Reparto de trabajo

Los procesadores deben de tener una carga de trabajo similar, de forma que la ejecución de sus grupos de tareas terminen a la vez. En caso de no ser así, un procesador estará libre antes que los demás y se estará desperdiciando su potencia de cálculo.

Patrón divide y vencerás

Este patrón se basa en dividir un problema inicial en pequeños problemas independientes y en unir la solución parcial de cada uno de ellos en última instancia. Los problemas derivados a su vez pueden ser divididos, por lo que se trata de un patrón recursivo. La concurrencia del patrón reside en que cada uno de los pequeños problemas puede ejecutarse de forma paralela, por lo que cuanto más se divida el problema inicial mayor grado de concurrencia existirá. En la Figura 24 puede verse el patrón ilustrado.

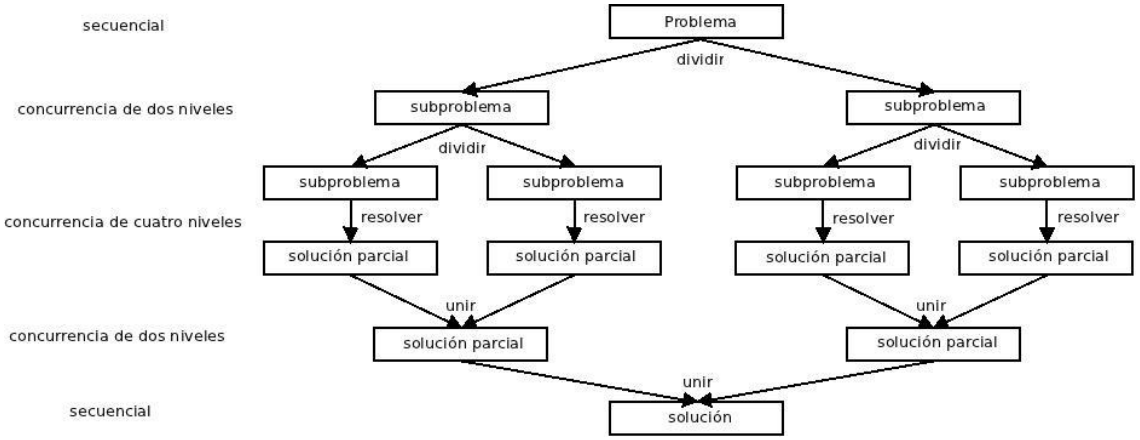


Figura 24 - Patrón divide y vencerás [54]

En este tipo de patrón el grado de concurrencia varía a lo largo de la ejecución del programa. Las operaciones de separación y unión de problemas introducen una carga de trabajo adicional que no puede ser ejecutada de forma concurrente. Si se lleva al límite puede ocurrir que los problemas finales sean tan simples que la sobrecarga introducida haga que no se obtenga ningún beneficio de la ejecución concurrente.

Al usar este patrón, hay que considerar los siguientes factores:

Distribuir tareas a los procesadores

El proceso de distribuir las tareas entre los procesadores es sencillo cuando los problemas generados a partir de la división de un problema mayor presentan una carga de trabajo similar. En este caso, se asigna cada uno de los problemas generados a un procesador.

La dificultad reside cuando un problema no es regular y la carga de trabajo asignada a cada procesador no es equitativa. En estos casos la solución es crear un conjunto de hilos y una cola de tareas. Los hilos van ejecutando tareas y cuando terminan sacan de la cola de tareas una nueva. De esta forma los procesadores estarán siempre ejecutando código.

Coste de división del problema

Como ya se ha dicho anteriormente existe un límite en el cual la división de un problema en dos más pequeños no aporta beneficio. Se debe de estudiar el algoritmo y marcar su grado de recursividad en unos límites que consigan la mayor eficiencia posible.

Organizados por descomposición de datos

Este tipo de algoritmos son los más adecuados cuando la concurrencia se realiza gracias a una buena descomposición de los datos entre distintas tareas. Existen dos patrones dentro de esta categoría: patrones de descomposición geométrica y patrones de datos recursivos.

Patrón de descomposición geométrica

Este patrón e basa en la descomposición de grandes estructuras de datos en pequeñas subestructuras contiguas consiguiendo realizar la actualización de datos de forma concurrente. Este patrón es muy adecuado cuando se usan arrays de una o de varias dimensiones.

Al utilizar este patrón, hay que tener en cuenta los siguientes factores:

Descomposición de los datos

El grado de descomposición de los datos tiene una gran influencia en la ejecución del código. Si se descomponen demasiado existirán muchos más conjuntos de datos que unidades de procesamiento y necesitarán intercambiar muchos mensajes para actualizar la estructura de datos general. Sin embargo, será fácil distribuir los datos entre los procesadores para conseguir una carga de trabajo adecuada.

Por otro lado, si los datos se descomponen en grandes bloques el reparto de trabajo entre los procesadores puede no estar balanceado. Sin embargo, los mensajes para actualizar la estructura de datos serán menores.

Distribución de los datos y reparto de trabajo entre tareas

Es importante decidir al usar este patrón como se realizará el reparto de tareas entre los distintos procesadores. La idea es realizar la división de los datos en grupos y a continuación asignar cada grupo a un procesador a través de una tarea.

El caso más simple es asignar estáticamente tareas a los diferentes procesadores y ejecutarlas de forma concurrente. Sin embargo, este proceso puede dar lugar a una carga de trabajo mal repartida debido a que se realizan distintas operaciones sobre los distintos grupos de datos.

Una de las soluciones a este problema es realizar una división muy grande de los datos; de esta manera se dispondrá de más tareas que procesadores y se irán ejecutando según se vayan solicitando.

Otra solución es utilizar un reparto de tareas dinámico para conseguir un balance de trabajo equilibrado. Esta solución introduce una sobrecarga que es mejor intentar evitar.

Patrón de datos recursivos

Este patrón es apropiado para estructuras de datos recursivos, como son una lista o un árbol. En principio estas estructuras de datos no parece que puedan ser compatibles con un procesamiento paralelo. Sin embargo, si se cambia la perspectiva del problema existe una concurrencia que se puede explotar.

La idea reside en realizar cálculos parciales en cada uno de los nodos que forman la estructura de datos y a continuación unir el resultado parcial con el sucesor directo. Una vez unido los resultados, se buscará el padre del padre de cada nodo y se combinarán de nuevo los resultados. Este proceso iterativo continuará hasta llegar a la raíz. La Figura 25 muestra el proceso de forma gráfica.

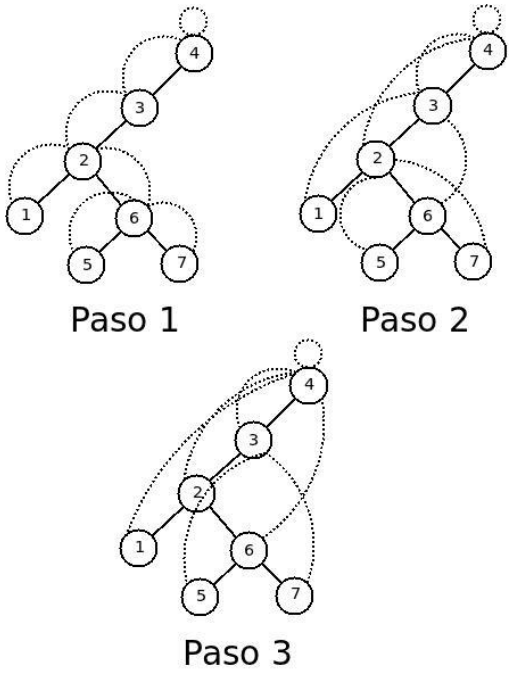


Figura 25 - Patrón recursivo aplicado a un árbol [54]

Una de las cosas que llaman más la atención de este tipo de patrón es que la carga de trabajo es mucho mayor que en su versión secuencial. Por ello solo se debe de utilizar cuando la ejecución de forma paralela de los cálculos proporcione un gran beneficio.

Al utilizar este tipo de patrón hay que tener en cuenta los siguientes factores:

Descomposición de datos

En este patrón cada estructura de datos se divide en elementos individuales y cada uno de ellos se asigna a diferentes unidades de procesamiento. En caso de que muchas de las unidades individuales se asignen a unos pocos procesadores, el paralelismo será muy pobre y no compensará la carga de trabajo mayor introducida por el proceso. Por lo tanto, será necesario disponer de un número adecuado de procesadores para que sea efectiva la ejecución de este patrón.

Sincronización

Debido a las dependencias que existen entre un elemento y su sucesor, se pueden encontrar situaciones en las que se intente actualizar un valor en el padre y aún no esté disponible. Para evitar esta situación es necesaria la utilización de barreras entre diferentes iteraciones del algoritmo, lo que introduce una sobrecarga en el mismo.

Organizados por el flujo de los datos

Este tipo de algoritmos se utilizarán cuando el principio de organización sea la necesidad de un orden de ejecución debido al flujo de los datos. Existen los siguientes algoritmos dentro de esta categoría:

Patrón de tubería

Este patrón se utilizará cuando las distintas operaciones que se realizan sobre un gran conjunto de datos se puedan ver como un flujo que pasa por distintas etapas. En cada momento se pueden estar ejecutando todas las etapas sobre distintos conjuntos de datos sin que ninguna de ellas tenga influencia sobre las demás.

La concurrencia en este tipo de algoritmos se encuentra en atribuir cada etapa a una tarea de forma que se puedan ejecutar todas en paralelo. Las dependencias serán subsanadas gracias al orden de ejecución de las distintas tareas.

La desventaja de este tipo de patrón es que al principio la concurrencia es muy limitada y muchas tareas están a la espera sin realizar operaciones. Sin embargo, cuando se llena la tubería, la ejecución paralela se ejecutará hasta que se terminen los datos. La Figura 26 muestra las etapas de ejecución de este tipo de patrones.

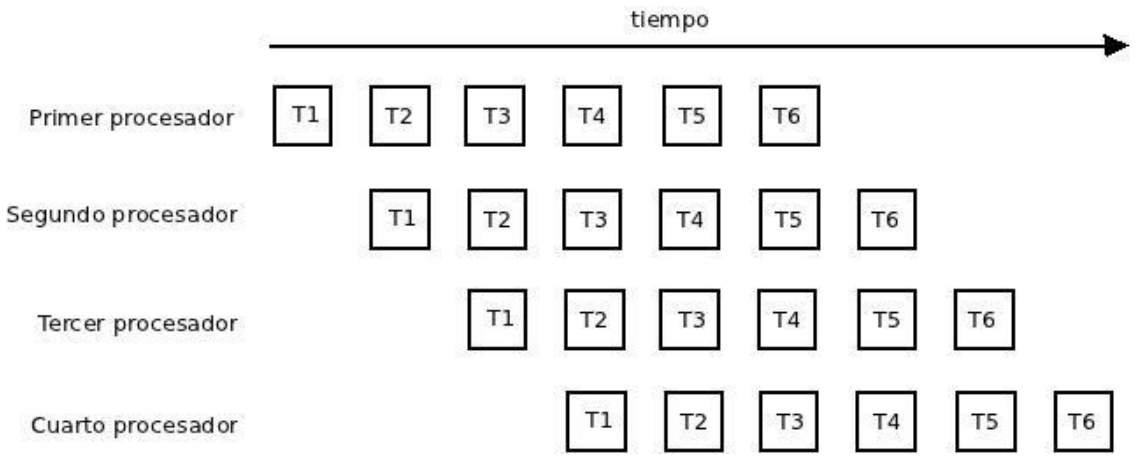


Figura 26 - Etapas de ejecución del patrón de tubería [54]

Los factores que hay que tener en cuenta al utilizar este patrón son los siguientes:

Estructura del código

Es necesario estructurar el código de manera que se puedan definir las diferentes operaciones a ejecutar de forma paralela. Para ello, se puede utilizar el identificador de cada hilo para asignarle una tarea diferente.

Manejo de errores

Debido a que las operaciones tienen que ejecutarse una detrás de otra, es necesario controlar de alguna manera los posibles errores que puedan surgir en alguna de las etapas de ejecución. Para ello, se puede disponer de un hilo dedicado solamente al manejo de errores, mientras el resto están ejecutando las operaciones.

Asignación de tareas a procesadores y reparto de trabajo

La asignación de tareas a procesadores más lógica es asignar cada etapa de procesamiento a una unidad de ejecución. Sin embargo, si se dispone de más tareas que

procesadores, será necesario asignar más de una etapa a cada procesador. Para ello, como norma general se deberán asignar etapas que no compartan recursos a un mismo procesador. Sin embargo, si se disponen de etapas con distinta carga de trabajo se deberán de asignar varias etapas cuyo trabajo sea menor a una misma unidad de procesamiento.

En caso de que se disponga de más procesadores que etapas, habrá que intentar paralelizar alguna de las etapas y asignarla a varios procesadores. En caso de que no exista un orden de ejecución marcado se podrán realizar varias ejecuciones en tubería en paralelo.

Patrón basado en coordinación por eventos

Este patrón se utilizará en casos en los que una aplicación se descomponga en pequeñas tareas que no sean totalmente independientes y necesiten interactuar de un modo irregular.

La solución para este tipo de algoritmos está basada en expresar el flujo de datos usando eventos. Cada evento será generado por una sola tarea y será consumido por otra. Como cada evento tiene que ser generado antes de ser consumido, se define un orden explícito en la ejecución de las tareas. Este tipo de patrones estarán marcados por los siguientes factores:

Definir las tareas

La estructura básica de cada tarea consistirá en recibir un evento, procesarlo y generar un nuevo evento.

Representar el flujo de eventos

Para permitir que la comunicación y la computación se ejecuten a la vez, las tareas deben ejecutar una comunicación asíncrona de los eventos en la cual el procesador envíe el evento y no tenga que esperar la respuesta para continuar su ejecución.

Forzar el orden de los eventos

En ocasiones es necesario que una tarea procese eventos en diferente orden en el que los recibe por lo que se debe permitir a una determinada tarea buscar y borrar eventos fuera de orden. Para realizar esta tarea, se pueden utilizar enfoques optimistas o pesimistas.

Los enfoques optimistas deshacen el resultado de un evento cuando ha sido ejecutado fuera de orden. Los enfoques pesimistas se aseguran de que el próximo evento a ejecutar es el correcto. Los enfoques pesimistas introducen un incremento de la latencia y del tiempo de comunicación, pero son más seguros.

Evitar los puntos muertos

Es posible que un sistema se quede esperando de forma infinita un evento que debe de ser generado por una tarea debido a que dicha tarea no podrá enviar el evento por alguna circunstancia irregular.

Asignación de tareas a procesadores y reparto de trabajo

La forma más lógica de realizar el reparto de trabajo es asignar una tarea a cada unidad de procesamiento y permitir a todas las tareas ejecutarse concurrentemente. En caso de que no existan suficientes procesadores, múltiples tareas deben de ser procesadas por cada unidad. Esto se debe de realizar de forma que el balance de trabajo esté equilibrado.

Eficiencia de la comunicación de eventos

El mecanismo utilizado para realizar la comunicación debe de ser lo más eficiente posible. En sistemas de memoria compartida debe asegurarse que el sistema de comunicación no se convierta en un cuello de botella para el procesamiento de las tareas.

3.5 DIAGRAMA DE CLASES

A continuación se va a mostrar el diagrama de clases de la aplicación desarrollada, donde podrá verse la relación entre los distintos paquetes existentes en la aplicación y sus clases.

Debido al gran tamaño que tiene el diagrama de clases se ha decidido dividirlo por paquetes; de esta manera se podrán apreciar mejor los detalles del esquema, que de otra manera sería imposible.

Paquete pfc.uc3m.plugin.moc.refactor.openmp

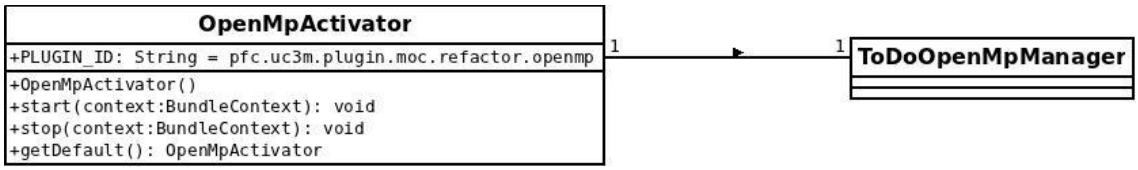


Figura 27 - pfc.uc3m.plugin.moc.refactor.openmp

Este paquete está formado solamente por la clase OpenMpActivator, que será la encargada de controlar el ciclo de vida del complemento, creando los objetos necesarios y destruyéndolos cuando sea oportuno.

Paquete pfc.uc3m.plugin.moc.refactor.openmp.fullauto.parser

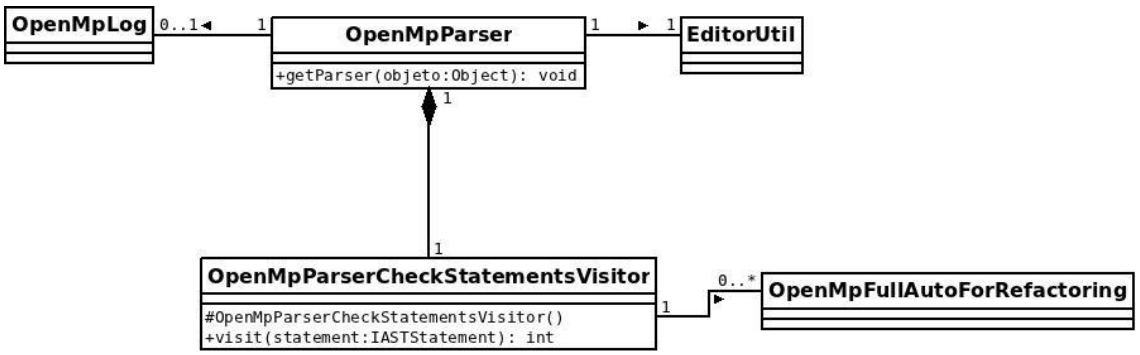


Figura 28 - pfc.uc3m.plugin.moc.refactor.openmp.fullauto.parser

En este paquete se encuentran las clases encargadas de analizar el código para poder realizar la refactorización posterior de forma automática.

Paquete pfc.uc3m.plugin.moc.refactor.openmp.handlers

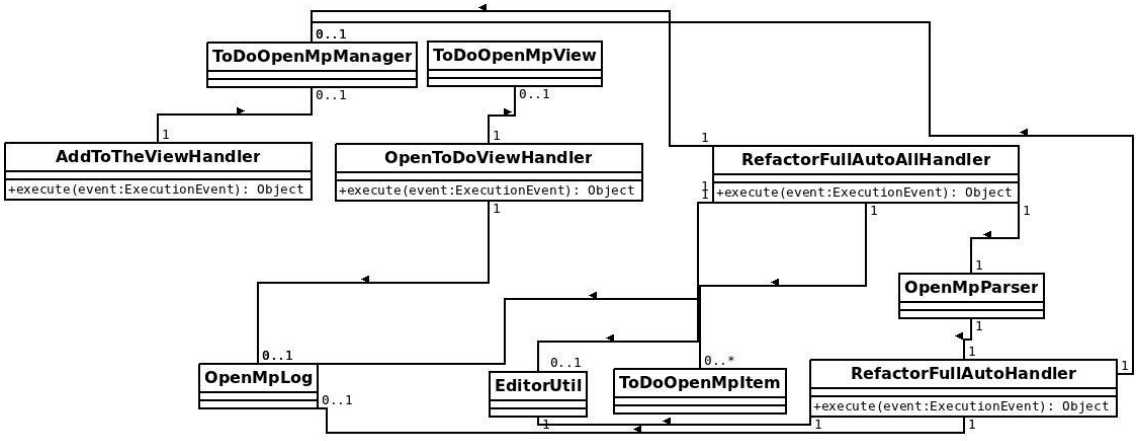


Figura 29 - pfc.uc3m.plugin.moc.refactor.openmp.handlers

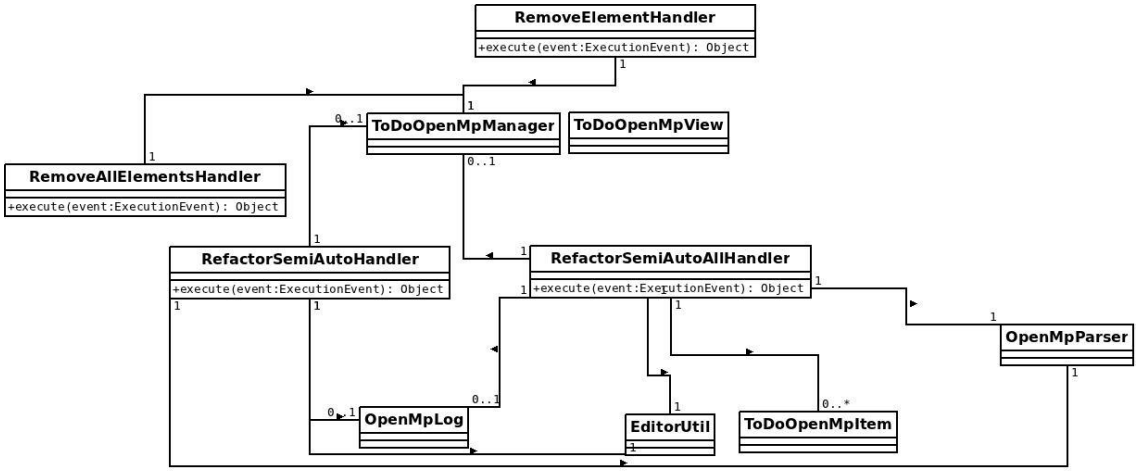


Figura 30 - pfc.uc3m.plugin.moc.refactor.openmp.handlers

Las clases contenidas en este paquete serán las encargadas de llevar a cabo toda la lógica de los comandos, comunicándose con el manejador del complemento y efectuando las operaciones necesarias.

Paquete pfc.uc3m.plugin.moc.refactor.openmp.log

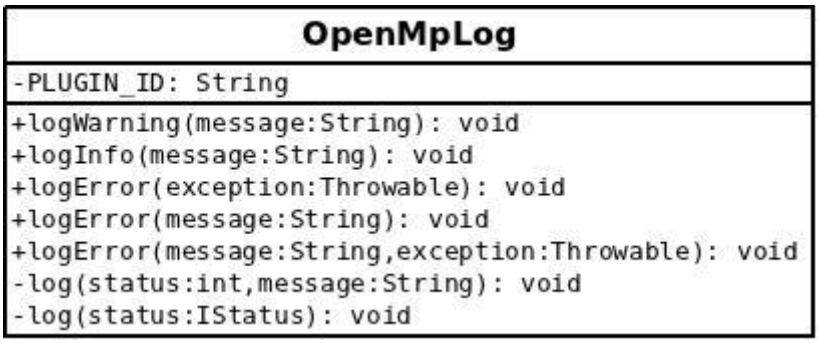


Figura 31 - pfc.uc3m.plugin.moc.refactor.openmp.log

Este paquete estará formado por una clase solamente. La clase OpenMpLog será la encargada de escribir los mensajes de error, de aviso o de información en la consola.

Paquete pfc.uc3m.plugin.moc.refactor.openmp.model

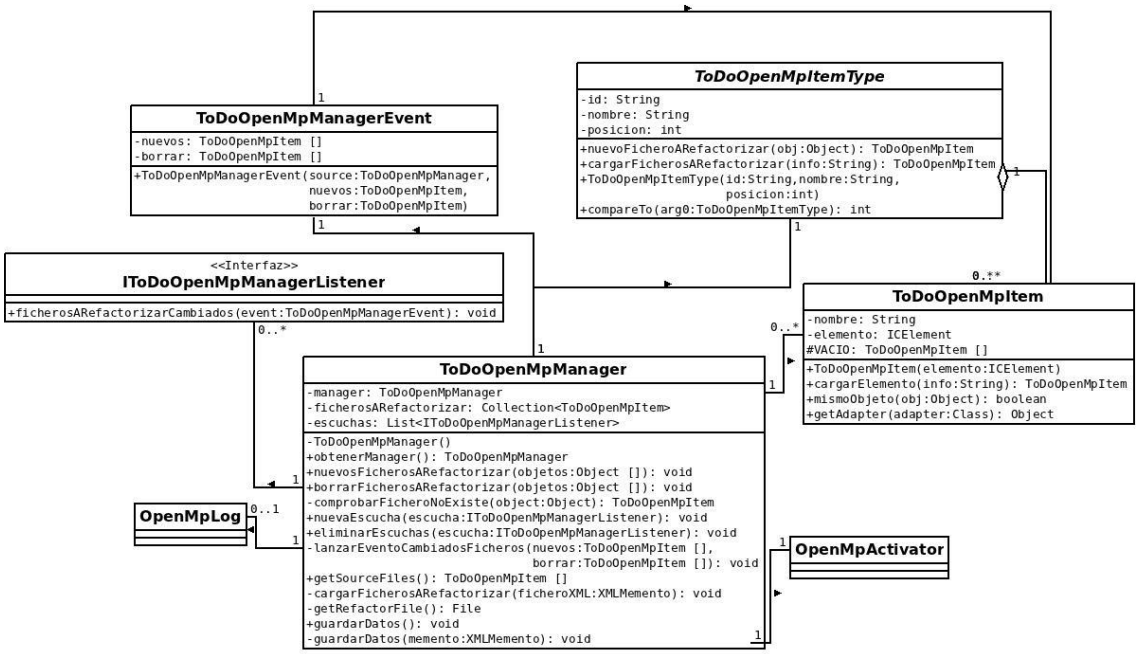


Figura 32 - pfc.uc3m.plugin.moc.refactor.openmp.model

Este paquete contendrá las clases encargadas de manejar distintos aspectos del complemento, como son lanzar eventos de actualización de la vista, envolver los objetos que se encuentran añadidos u obtener los ficheros XML de configuración del complemento.

Paquete pfc.uc3m.plugin.moc.refactor.openmp.refactor

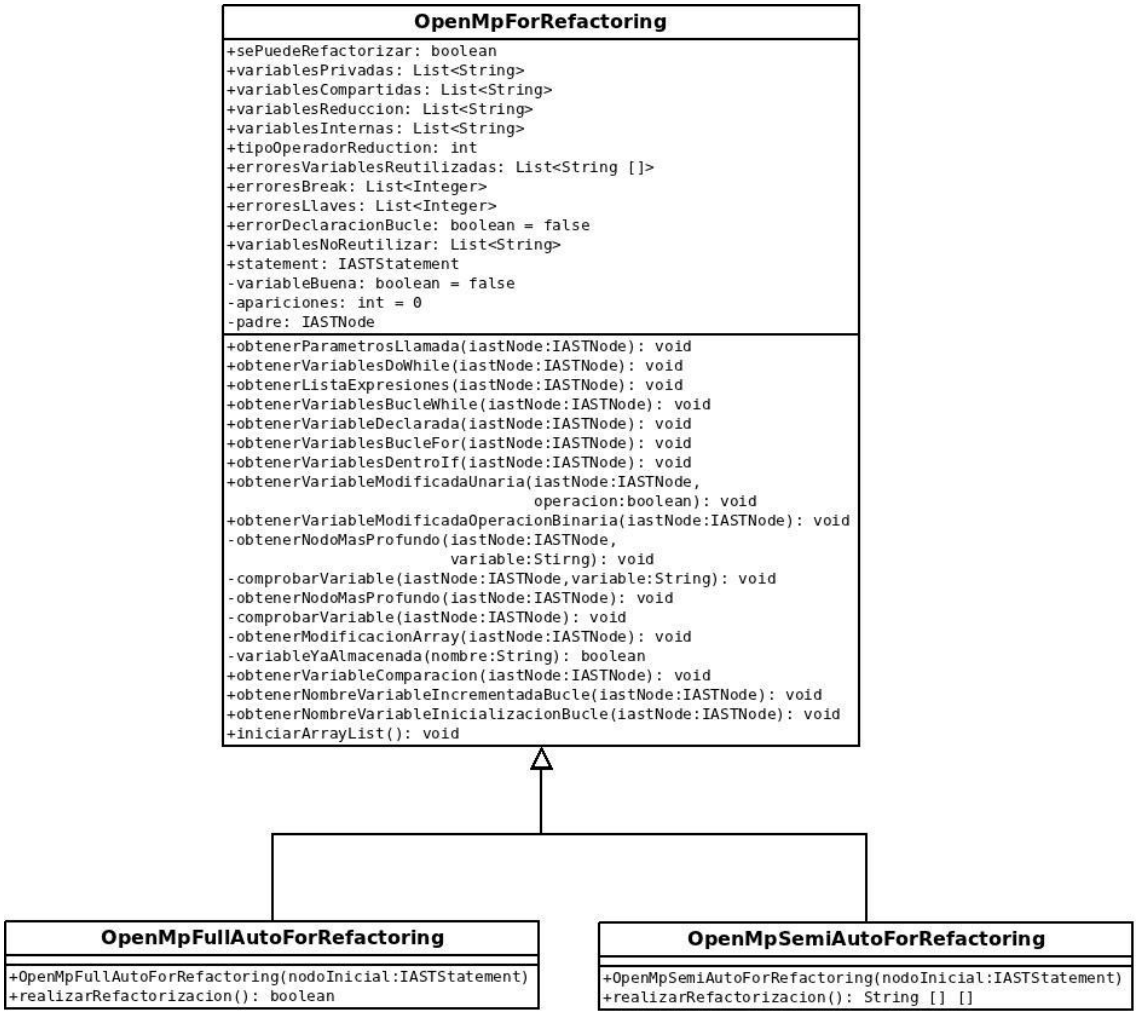


Figura 33 - pfc.uc3m.plugin.moc.refactor.openmp.refactor

Las clases contenidas en este paquete se utilizarán para buscar condiciones dentro del código que impidan llevar a cabo la refactorización del mismo. Utilizarán el AST generado e irán recorriendo uno a uno todos los nodos que estén contenidos en los bucles encontrados. La refactorización automática y la semiautomática se llevarán a cabo de forma muy parecida, por lo que ambas clases tienen una clase padre común de la que utilizarán la mayoría de los métodos.

Paquete pfc.uc3m.plugin.moc.refactor.openmp.semiauto.parser

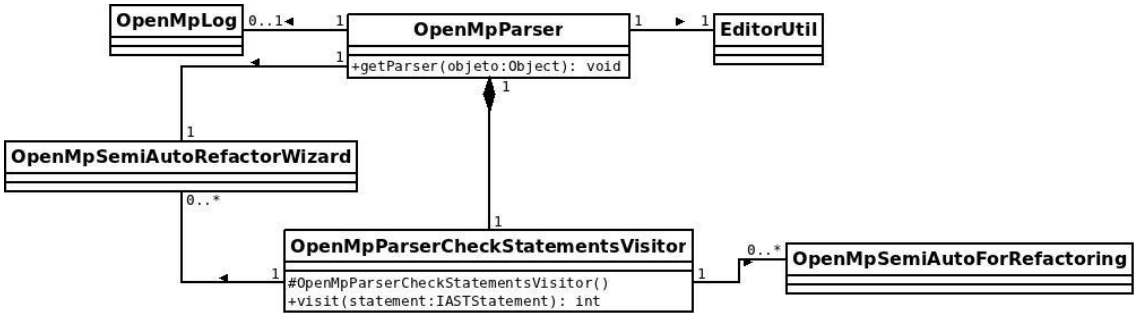


Figura 34 - pfc.uc3m.plugin.moc.refactor.openmp.semiauto.parser

Las clases de este paquete serán las equivalentes a las del paquete pfc.uc3m.plugin.moc.refactor.openmp.fullato.parser pero para la versión semiautomática de la refactorización. Se encargarán de analizar el código y de lanzar el asistente para que el usuario pueda seleccionar las opciones de las directivas a introducir en el código.

Paquete pfc.uc3m.plugin.moc.refactor.openmp.util

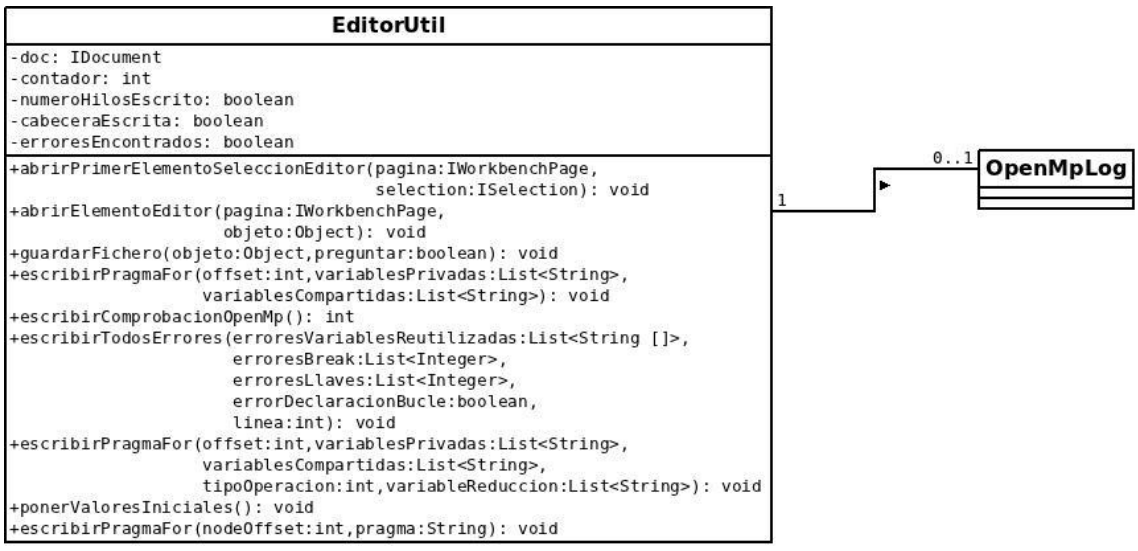


Figura 35 - pfc.uc3m.plugin.moc.refactor.openmp.util

La clase contenida en este paquete será la encargada de acceder directamente al editor y de insertar el código de las directivas OpenMP generadas durante el proceso de refactorización.

Paquete pfc.uc3m.plugin.moc.refactor.openmp.views.todo

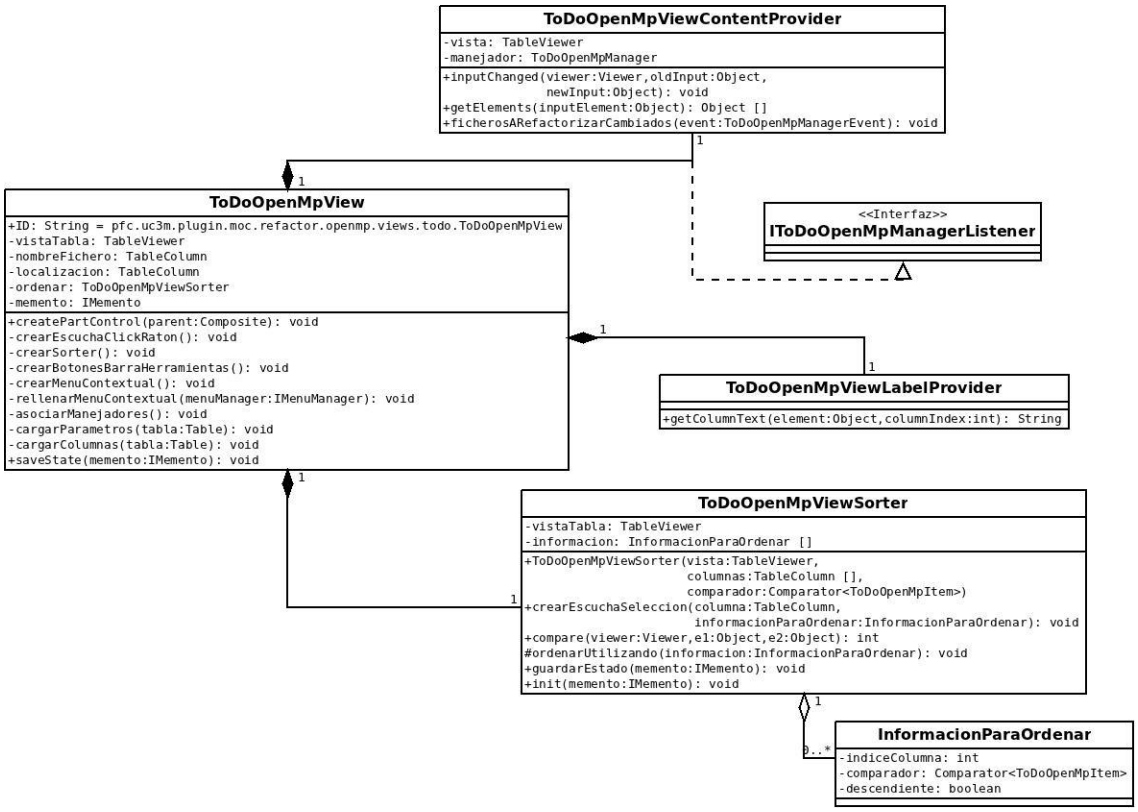


Figura 36 - pfc.uc3m.plugin.moc.refactor.openmp.views.todo

En este paquete estarán contenidas todas las clases relacionadas con la vista de refactorización. La vista estará formada por un *ContentProvider*, un *LabelProvider* y un *Sorter* que serán los encargados de mostrar los ficheros a refactorizar y de ordenarlos según el criterio escogido.

Paquete pfc.uc3m.plugin.moc.refactor.openmp.wizards.semiauto

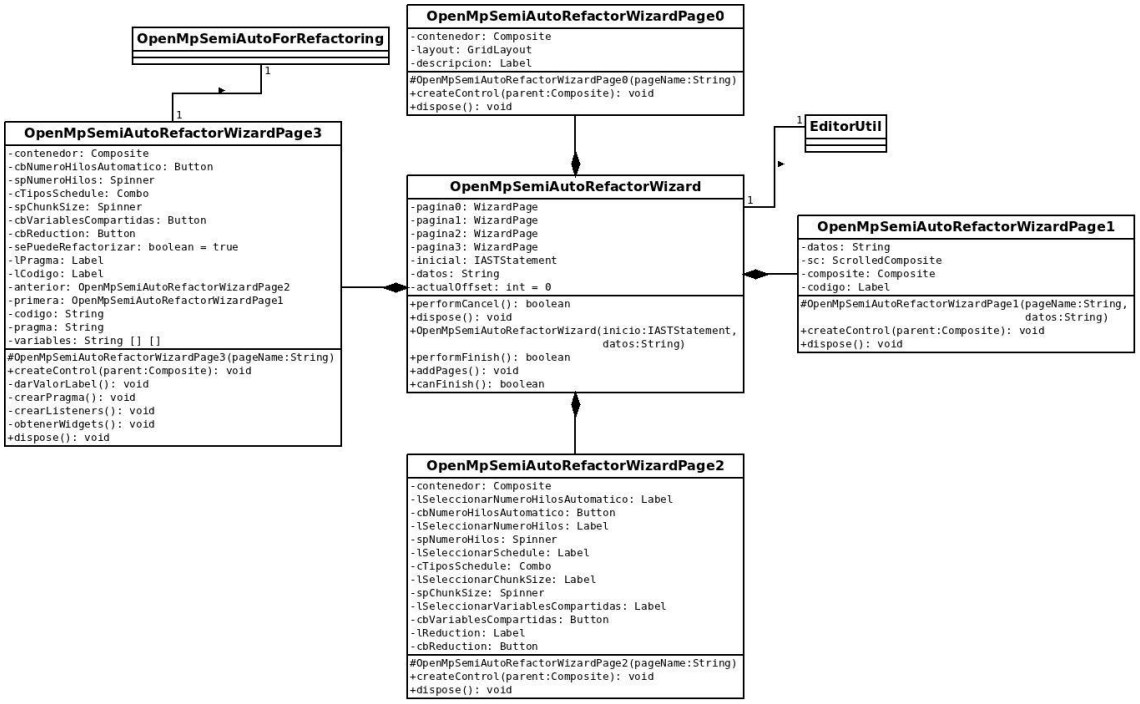


Figura 37 - pfc.uc3m.plugin.moc.refactor.openmp.wizards.semiauto

Por último, este paquete estará formado por el asistente a través del cual el usuario llevará a cabo la refactorización semiautomática del código y por todas las páginas que lo forman.

4. FASES DEL DESARROLLO

A continuación se detallarán cada una de las etapas del desarrollo del proyecto desde su inicio hasta la versión final.

4.1 PLANIFICACIÓN DEL TRABAJO

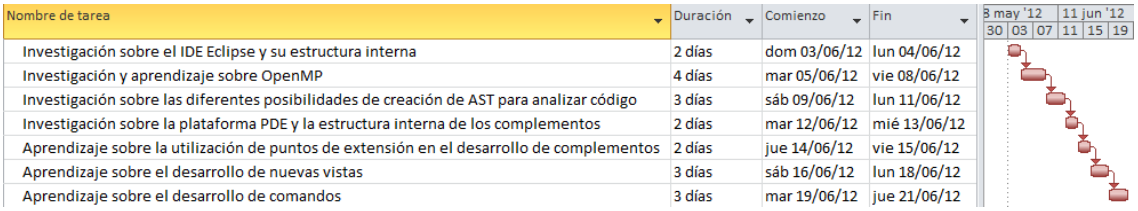
Para el correcto desarrollo del complemento se ha realizado una planificación de las tareas que son necesarias para cumplir con los requisitos de usuario descritos anteriormente. La fecha de comienzo de creación del proyecto de fin de grado se sitúa el 3 de Junio del año 2012. La fecha prevista de finalización es el día 1 de Septiembre de 2012. El desarrollo del proyecto de fin de grado se ha separado en cuatro fases diferenciadas de manera que cada una de ellas es independiente de las demás. Para cada fase se muestra las diferentes tareas por las que están formadas y un diagrama de Gantt donde se puede ver los días destinados a cada una de las tareas.

En primer lugar ha sido necesario un proceso de aprendizaje de desarrollo de complementos en Eclipse. En la Tabla 2 pueden verse las tareas por las que está formado el proceso de aprendizaje.

Tarea	Descripción
Investigación sobre el IDE Eclipse y su estructura interna	Es necesario comprender como funciona la plataforma Eclipse internamente para poder comenzar a desarrollar un complemento para este IDE.
Investigación y aprendizaje sobre OpenMP	Es necesario aprender el funcionamiento de OpenMP y de todas sus directivas disponibles.
Investigación sobre las diferentes posibilidades de creación de AST para analizar código	Debido al gran número de herramientas para la creación de un AST es necesario estudiarlas y elegir la más adecuada para el complemento.
Investigación sobre la plataforma PDE y la estructura interna de los complementos	La plataforma PDE servirá de gran ayuda a la hora de desarrollar el complemento por lo que es necesario aprender su funcionamiento.
Aprendizaje sobre la utilización de puntos de extensión en el desarrollo de complementos	El desarrollo de complementos en Eclipse estará basado en el uso de puntos de extensión de otros complementos desarrollados anteriormente por lo que habrá que aprender a utilizarlos.
Aprendizaje sobre el desarrollo de nuevas vistas	Una de las partes fundamentales del complemento desarrollado es la vista de refactorización por lo que habrá que aprender a desarrollar nuevas vistas.
Aprendizaje sobre el desarrollo de comandos	Los comandos serán los encargados de llevar a cabo la lógica del complemento por lo que habrá que aprender a crearlos y utilizarlos.

Tabla 3 – Tareas de la fase uno

El diagrama de Gantt asociado a esta fase del proyecto puede verse en la Figura 38:



Nombre de tarea	Duración	Comienzo	Fin
Investigación sobre el IDE Eclipse y su estructura interna	2 días	dom 03/06/12	lun 04/06/12
Investigación y aprendizaje sobre OpenMP	4 días	mar 05/06/12	vie 08/06/12
Investigación sobre las diferentes posibilidades de creación de AST para analizar código	3 días	sáb 09/06/12	lun 11/06/12
Investigación sobre la plataforma PDE y la estructura interna de los complementos	2 días	mar 12/06/12	mié 13/06/12
Aprendizaje sobre la utilización de puntos de extensión en el desarrollo de complementos	2 días	jue 14/06/12	vie 15/06/12
Aprendizaje sobre el desarrollo de nuevas vistas	3 días	sáb 16/06/12	lun 18/06/12
Aprendizaje sobre el desarrollo de comandos	3 días	mar 19/06/12	jue 21/06/12

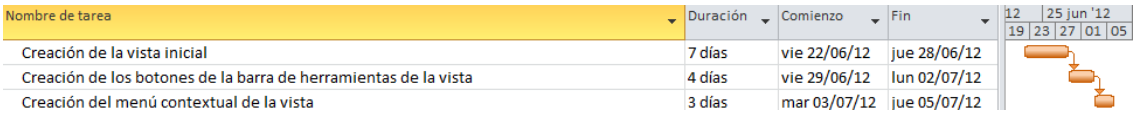
Figura 38 - Diagrama de Gantt de la fase uno

En cuanto al desarrollo de la interfaz de usuario la Tabla 3 muestra todas las tareas por las que está formada esta fase.

Tarea	Descripción
Creación de la vista inicial	Desarrollo de la vista que contendrá los ficheros a refactorizar en el futuro.
Creación de los botones de la barra de herramientas de la vista	Desarrollo de los cuatro botones situados en la barra de herramientas de la vista que permitirán realizar las distintas acciones disponibles.
Creación del menú contextual de la vista	Desarrollo del menú contextual que contendrá todas las posibles acciones disponibles.

Tabla 4 - Tareas de la fase dos

El diagrama de Gantt asociado a esta fase de desarrollo puede verse en la Figura 39:



Nombre de tarea	Duración	Comienzo	Fin
Creación de la vista inicial	7 días	vie 22/06/12	jue 28/06/12
Creación de los botones de la barra de herramientas de la vista	4 días	vie 29/06/12	lun 02/07/12
Creación del menú contextual de la vista	3 días	mar 03/07/12	jue 05/07/12

Figura 39 - Diagrama de Gantt de la fase dos

Una vez desarrollada la parte visual del complemento será necesario crear el motor léxico y gramatical del sistema, el cuál contendrá toda la lógica del complemento. Las tareas de esta fase pueden verse en la Tabla 4.

Tarea	Descripción
Desarrollo del comando y del manejador de añadir ficheros a la vista de refactorización	Creación de la lógica para añadir ficheros con código fuente escrito en C a la vista de refactorización.
Desarrollo del comando y del manejador de borrar elementos de la vista	Creación de la lógica para borrar los elementos seleccionados de la vista de refactorización.
Desarrollo del comando y del manejador de borrar todos los elementos de la vista	Desarrollo del código para permitir al usuario borrar de una vez todos los elementos que se encuentren en la vista de refactorización.
Desarrollo del comando y del manejador de refactorizar ficheros de forma automática	Desarrollo del código para realizar la refactorización de ficheros seleccionados de forma automática.
Desarrollo del comando y del manejador de refactorizar todos los ficheros de la vista de forma automática	Creación de la lógica para permitir al usuario la refactorización de todos los ficheros que se encuentren en la vista de forma automática.
Desarrollo del comando y del manejador de refactorizar ficheros de forma semiautomática	Desarrollo de la lógica para realizar la refactorización de ficheros seleccionados de forma semiautomática.
Desarrollo del comando y del manejador de refactorizar todos los ficheros de la vista de forma semiautomática	Creación del código para permitir al usuario llevar a cabo la refactorización semiautomática de todos los ficheros que se encuentren en la vista.
Creación y análisis del AST al realizar la refactorización de ficheros	Análisis del AST y búsqueda de posibles condiciones que impidan la refactorización del código.

Tabla 5 - Tareas de la fase tres

El diagrama de Gantt asociado a esta fase puede verse en la Figura 40:

Nombre de tarea	Duración	Comienzo	Fin
Desarrollo del comando y del manejador de añadir ficheros a la vista de refactorización	4 días	vie 06/07/12	lun 09/07/12
Desarrollo del comando y del manejador de borrar elementos de la vista	2 días	mar 10/07/12	mié 11/07/12
Desarrollo del comando y del manejador de borrar todos los elementos de la vista	1 día	jue 12/07/12	jue 12/07/12
Desarrollo del comando y del manejador de refactorizar ficheros de forma automática	7 días	vie 13/07/12	jue 19/07/12
Desarrollo del comando y del manejador de refactorizar todos los ficheros de la vista de forma automática	1 día	vie 20/07/12	vie 20/07/12
Desarrollo del comando y del manejador de refactorizar ficheros de forma semiautomática	4 días	sáb 21/07/12	mar 24/07/12
Desarrollo del comando y del manejador de refactorizar todos los ficheros de la vista de forma semiautomática	2 días	mié 25/07/12	jue 26/07/12
Creación y análisis del AST al realizar la refactorización de ficheros	10 días	vie 27/07/12	dom 05/08/12

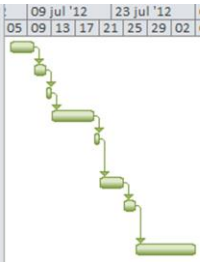


Figura 40 - Diagrama de Gantt de la fase tres

En último lugar se encuentra la parte de redacción de la memoria del trabajo de fin de grado. Esta fase del proyecto estará formada por las tareas contenidas en la Tabla 5:

Tarea	Descripción
Capítulo 1: Introducción	Redacción y revisión del capítulo de introducción.
Capítulo 2: Estado de la cuestión	Redacción y revisión del capítulo de estado de la cuestión.
Capítulo 3: Análisis y diseño	Redacción y revisión del capítulo de análisis y diseño.
Capítulo 4: Fases del desarrollo	Redacción y revisión del capítulo de fases del desarrollo.
Capítulo 5: Conclusiones	Redacción y revisión del capítulo de conclusiones.
Anexo I: Guía de instalación	Redacción y revisión del anexo de guía de instalación.
Anexo II: Guía del usuario	Redacción y revisión del anexo de guía del usuario.
Anexo III: Guía de creación de complementos en Eclipse	Redacción y revisión del anexo de guía de creación de complementos en Eclipse.
Resto del documento	Redacción del resto de apartados del documento, como la bibliografía o los índices.

Tabla 6 – Tareas de la fase dos

El diagrama de Gantt asociado a la redacción de la memoria puede verse en la Figura 41:

Nombre de tarea	Duración	Comienzo	Fin
Redacción del capítulo 1: Introducción	4 días	lun 06/08/12	jue 09/08/12
Redacción del capítulo 2: Estado de la cuestión	5 días	vie 10/08/12	mar 14/08/12
Redacción del capítulo 3: Análisis y diseño	4 días	mié 15/08/12	sáb 18/08/12
Redacción del capítulo 4: Fases del desarrollo	5 días	dom 19/08/12	jue 23/08/12
Redacción del capítulo 5: Conclusiones	3 días	vie 24/08/12	dom 26/08/12
Redacción del anexo I: Guía de instalación	1 día	lun 27/08/12	lun 27/08/12
Redacción del anexo II: Guía del usuario	2 días	mar 28/08/12	mié 29/08/12
Redacción del anexo III: Guía de creación de complementos en Eclipse	2 días	jue 30/08/12	vie 31/08/12
Redacción del resto del documento	1 día	sáb 01/09/12	sáb 01/09/12

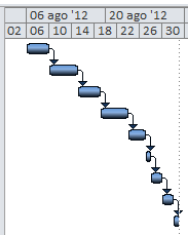


Figura 41 - Diagrama de Gantt de la fase cuatro

La Figura 42 muestra el diagrama de Gantt completo de todo el proyecto, donde se podrán apreciar las distintas tareas a realizar:

Nombre de tarea	Duración	Comienzo	Fin
Investigación sobre el IDE Eclipse y su estructura interna	2 días	dom 03/06/12	lun 04/06/12
Investigación y aprendizaje sobre OpenMP	4 días	mar 05/06/12	vie 08/06/12
Investigación sobre las diferentes posibilidades de creación de AST para analizar código	3 días	sáb 09/06/12	lun 11/06/12
Investigación sobre la plataforma PDE y la estructura interna de los complementos	2 días	mar 12/06/12	mié 13/06/12
Aprendizaje sobre la utilización de puntos de extensión en el desarrollo de complementos	2 días	jue 14/06/12	vie 15/06/12
Aprendizaje sobre el desarrollo de nuevas vistas	3 días	sáb 16/06/12	lun 18/06/12
Aprendizaje sobre el desarrollo de comandos	3 días	mar 19/06/12	jue 21/06/12
Creación de la vista inicial	7 días	vie 22/06/12	jue 28/06/12
Creación de los botones de la barra de herramientas de la vista	4 días	vie 29/06/12	lun 02/07/12
Creación del menú contextual de la vista	3 días	mar 03/07/12	jue 05/07/12
Desarrollo del comando y del manejador de añadir ficheros a la vista de refactorización	4 días	vie 06/07/12	lun 09/07/12
Desarrollo del comando y del manejador de borrar elementos de la vista	2 días	mar 10/07/12	mié 11/07/12
Desarrollo del comando y del manejador de borrar todos los elementos de la vista	1 día	jue 12/07/12	jue 12/07/12
Desarrollo del comando y del manejador de refactorizar ficheros de forma automática	7 días	vie 13/07/12	jue 19/07/12
Desarrollo del comando y del manejador de refactorizar todos los ficheros de la vista de forma automática	1 día	vie 20/07/12	vie 20/07/12
Desarrollo del comando y del manejador de refactorizar ficheros de forma semiautomática	4 días	sáb 21/07/12	mar 24/07/12
Desarrollo del comando y del manejador de refactorizar todos los ficheros de la vista de forma semiautomática	2 días	mié 25/07/12	jue 26/07/12
Creación y análisis del AST al realizar la refactorización de ficheros	10 días	vie 27/07/12	dom 05/08/12
Redacción del capítulo 1: Introducción	4 días	lun 06/08/12	jue 09/08/12
Redacción del capítulo 2: Estado de la cuestión	5 días	vie 10/08/12	mar 14/08/12
Redacción del capítulo 3: Análisis y diseño	4 días	mié 15/08/12	sáb 18/08/12
Redacción del capítulo 4: Fases del desarrollo	5 días	dom 19/08/12	jue 23/08/12
Redacción del capítulo 5: Conclusiones	3 días	vie 24/08/12	dom 26/08/12
Redacción del anexo I: Guía de instalación	1 día	lun 27/08/12	lun 27/08/12
Redacción del anexo II: Guía del usuario	2 días	mar 28/08/12	mié 29/08/12
Redacción del anexo III: Guía de creación de complementos en Eclipse	2 días	jue 30/08/12	vie 31/08/12
Redacción del resto del documento	1 día	sáb 01/09/12	sáb 01/09/12

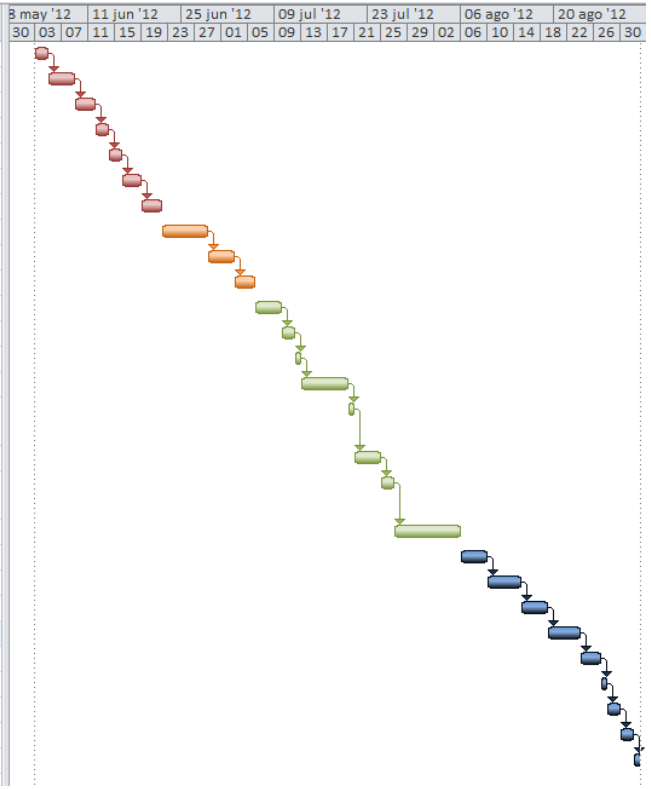


Figura 42 - Diagrama de Gantt completo

En las siguientes secciones se describe de forma más detallada el desarrollo del código que forma el complemento:

4.2 INTERFAZ DE USUARIO

El complemento desarrollado tiene una parte visual muy simple formada por una vista y varios botones que permitirán a los usuarios interactuar con los diferentes ficheros que se quieran refactorizar de C a C con OpenMP.

Para el desarrollo de la vista ha sido necesario utilizar el punto de extensión de la clase org.eclipse.ui.views, el cuál se utilizará siempre que se quiera crear una nueva vista del IDE. Una vez creada la vista dentro del punto de extensión, será necesario definir los elementos que la formarán de forma programática.

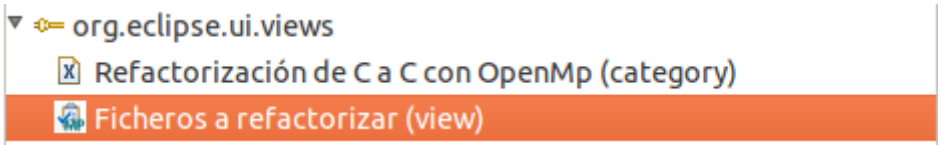


Figura 43 - Punto de extensión de la vista

La vista estará formada por una tabla de dos columnas. La primera de las columnas mostrará el nombre del fichero que está añadido a la vista y la segunda mostrará su localización absoluta.

La barra de herramientas de la vista estará formada por cuatro botones, los cuales permitirán realizar las siguientes acciones:

- Refactorizar los ficheros seleccionados de forma automática.
- Refactorizar los ficheros seleccionados de forma semiautomática.
- Borrar los ficheros seleccionados de la vista.
- Borrar todos los ficheros de la vista.

Figura 44 - Botones de la vista de refactorización

Para añadir los botones a la barra de herramientas será necesario crearla de forma programática y añadir el separador que se utilizará para unir los comandos pertenecientes a una misma categoría. Será necesario utilizar el punto de extensión org.eclipse.ui.menus, el cuál se utilizará siempre que se quiera contribuir a algún menú en el IDE.

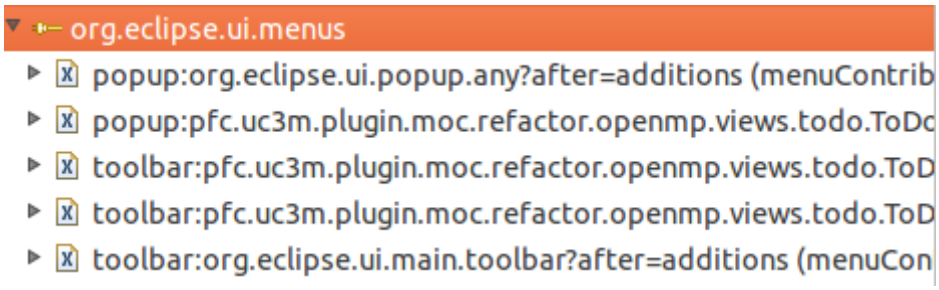


Figura 45 - Punto de extensión de menús

Por último, se creará un menú contextual donde se encontrarán las distintas acciones que se podrán llevar a cabo a través del complemento. Para ello, se creará de forma programática y luego se añadirá a través del punto de extensión org.eclipse.ui.menus, de la misma manera que los botones de la barra de herramientas.

4.3 MOTOR LÉXICO Y GRAMATICAL DEL SISTEMA

La parte más importante del complemento será la formada por la lógica que llevará a cabo el análisis del código de forma estática e insertará en los ficheros correspondientes las directivas que permitirán ejecutar los bucles de forma paralela.

Para crear los comandos que contendrán la lógica del complemento será necesario utilizar el punto de extensión org.eclipse.ui.commands, el cuál se utilizará siempre que se quieran crear nuevos comandos que lleven a cabo algún tipo de lógica.

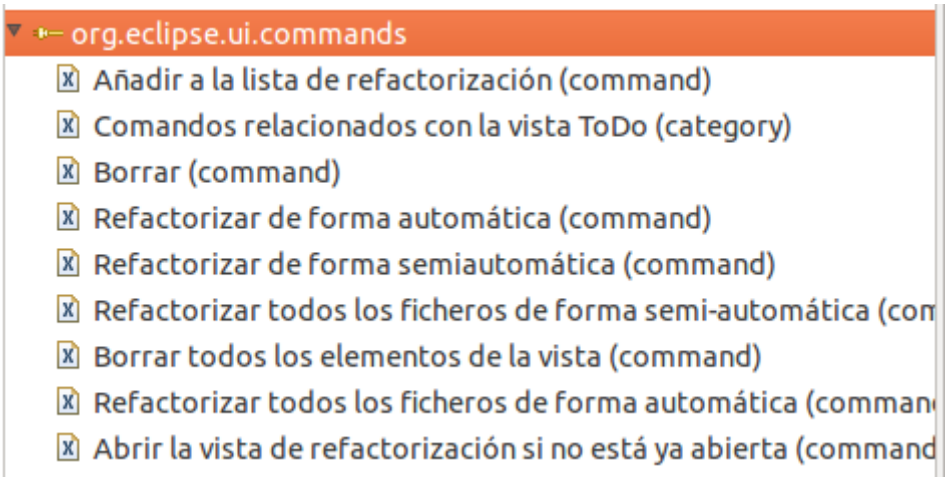


Figura 46 - Punto de extensión de todos los comandos

Para definir el comportamiento de cada uno de estos comandos será necesario crear un manejador utilizando la extensión org.eclipse.ui.handlers. Cada manejador creado estará formado por una clase java donde se encontrará toda la lógica que tendrá que ejecutar el complemento cuando se ejecute un determinado comando.

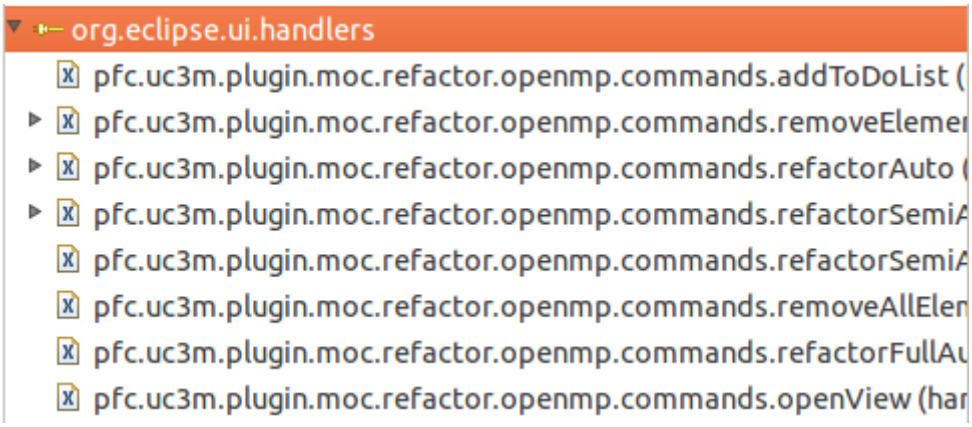


Figura 47 - Punto de extensión de los manejadores

Para comprobar si un determinado manejador debe estar activo se utilizarán condiciones dentro del propio fichero XML. En el caso de los manejadores de los comandos de refactorización de ficheros solo estarán activos cuando uno o más elementos estén seleccionados, ya sean en la vista o en el explorador de proyectos.

Para reutilizar una condición en varios manejadores se puede utilizar el punto de extensión org.eclipse.core.expressions.definitions. Dentro de esta extensión se podrán crear definiciones de condiciones que luego serán referenciadas desde los manejadores o desde otro punto de extensión, facilitando la reutilización de código.

AÑADIR FICHEROS A LA VISTA

Para poder llevar a cabo la refactorización de un fichero a través de la vista de refactorización será necesario añadirlo a la misma. Para poder añadir el archivo se ha creado un comando el cual estará disponible en un menú contextual visible cuando se pinche con el botón derecho sobre un fichero que sea un archivo de código fuente escrito en lenguaje C.

El comando en primer lugar obtendrá una instancia de la clase que controla la vista. A través de la instancia, se convertirán los elementos seleccionados en un array genérico de objetos, los cuales se irán recorriendo uno a uno y se irán comparando con los ficheros que han sido añadidos anteriormente a la vista para evitar que se añadan dos veces.

Una vez que se ha comprobado que el fichero no ha sido añadido anteriormente, se convertirá en un tipo de objeto especial que servirá de envoltorio para poder manejarlo. A continuación se añadirá a la lista de ficheros que contiene la vista y se lanzará un evento para actualizarla, de manera que el usuario pueda observar los cambios realizados.

BORRAR ELEMENTOS DE LA VISTA

El borrado de elementos de la vista de refactorización se podrá realizar tanto desde la barra de herramientas como desde el menú contextual disponible en la vista. El proceso de borrado será ejecutado por un comando el cuál solo estará disponible cuando estén seleccionados uno o más elementos de la vista de refactorización.

El comando en primer lugar obtendrá una instancia de la clase que controla la vista. A través de esta instancia, convertirá los ficheros seleccionados en un array

genérico de objetos, los cuales se irán recorriendo para comprobar que los ficheros seleccionados pertenecen a la vista.

Una vez comprobado que el archivo existe en la vista, se procederá con el borrado y se lanzará un evento de actualización de manera que el usuario pueda observar los cambios realizados.

ANÁLISIS Y REFACTORIZACIÓN DEL CÓDIGO

En cuanto al análisis del código se ha optado por utilizar librerías proporcionadas por el complemento CDT [53] para crear un árbol de sintaxis. Estas librerías permiten crear visitantes de AST, que son clases las cuales son llamadas cuando se encuentra un determinado tipo de nodo en el árbol de sintaxis.

Para el caso concreto del complemento desarrollado en este proyecto de fin de grado basta con crear un visitante de AST que se ejecute cada vez que se encuentre un nodo de tipo sentencia y que compruebe si el nodo es de tipo bucle *for*. En caso afirmativo, se procederá a crear un objeto de la clase que se encargará de profundizar dentro del nodo.

El proceso de búsqueda de dependencias dentro del bucle seguirá varios pasos, profundizando cada vez más en los nodos hijos:

Comprobación de la declaración del bucle

En primer lugar, es necesario comprobar que la declaración del bucle sigue un patrón sencillo. La API de OpenMP exige que la declaración del bucle siga el siguiente patrón para poder realizar la refactorización del bucle [14]:

*for(**expresión-inicio; var op b; incremento-var**)* donde:

- var y b son variables enteras.
- expresión-inicio es la operación de asignación de valor a la variable var.
- op es uno de los siguientes operadores:
  - >, <, <=, =>.
- incremento-var es una expresión donde se incrementa o se disminuye var con uno de los siguientes operadores:
  - ++, --, +=, -=.

En caso de que la declaración del bucle no siga este patrón la refactorización será imposible. Para comprobar cada una de las tres partes de la declaración del bucle el método obtendrá los tres primeros nodos hijos y llamará a los métodos respectivos, que realizarán las siguientes comprobaciones:

Comprobación de la expresión de inicialización de variable

En primer lugar se comprobará que el nodo hijo que se corresponde con la expresión de inicio de variable en la declaración del bucle sea un nodo de tipo IASTExpressionStatement.

En caso afirmativo se comprobará que el hijo de dicho nodo no es de tipo IASTExpressionList. Si el nodo fuese de ese tipo significaría que la expresión en la declaración del bucle no sería una expresión simple, sino que estaría formada por varias inicializaciones y por lo tanto la refactorización no sería posible.

En caso de que todo sea correcto se obtendrá el nombre de la variable de inicialización y se almacenará en la lista de variables privadas y en la lista de variables que no se pueden modificar dentro del bucle. Se procederá a comprobar la condición de ejecución del bucle.

Comprobación de la condición de ejecución del bucle

En primer lugar se comprobará que el nodo hijo que se corresponde con la condición de ejecución sea un nodo de tipo IASTBinaryExpression.

En caso afirmativo se comprobará que el hijo de dicho nodo no es de tipo IASTBinaryExpression. Si el nodo fuese de ese tipo significaría que la condición de ejecución del bucle estaría formada por varias operaciones de comparación de variables, por lo que el bucle no tendría una declaración simple y la refactorización no sería posible.

El siguiente paso sería obtener los hijos por los que estará formada la expresión. El primero de los hijos siempre será la variable que se encuentra en el lado izquierdo de la expresión. Dicha variable debe de corresponderse con la variable que se ha inicializado en la primera parte de la declaración del bucle para que se pueda llevar a cabo la refactorización.

Para comprobar que son la misma variable, se verifica si el nombre ya está almacenado en la lista de variables que no se pueden reutilizar; si no se encuentra en la lista significará que la declaración del bucle no es adecuada.

Posteriormente se comprueba el segundo de los hijos de la expresión y se observa si se trata de un nodo de tipo IASTLiteralExpression. En caso afirmativo, significará que la parte derecha de la expresión es un valor literal y no habrá que realizar ninguna operación con él. En caso contrario significará que la parte derecha de la expresión es una variable y habrá que obtener su nombre y añadirla a la lista de variables que no se pueden modificar y a la lista de variables compartidas por todos los hilos.

Se procederá a comprobar la operación de incremento de la variable.

Comprobación de la operación de incremento de la variable

En primer lugar se comprobará que el nodo hijo que se corresponde con la operación de incremento de la variable es un nodo de tipo IASTUnaryExpression o de tipo IASTBinaryExpression.

En caso de que sea un nodo de una operación unaria se obtendrá el nombre de la variable y se comprobará que se corresponda con una de las variables ya almacenadas en las demás comprobaciones. Sin embargo, si se trata de una operación binaria será necesario comprobar que la variable que se incrementa es la adecuada y si se utiliza alguna otra variable en la parte derecha de la expresión. En caso de que se utilice una variable en vez de un valor literal, se almacenará dicha variable como una variable compartida.

Se procederá a ejecutar las comprobaciones necesarias dentro del bucle principal.

Comprobación de la existencia de llaves en el bucle

Para poder llevar a cabo el análisis del fichero de forma correcta, es necesario que el bucle esté formado por un nodo hijo de tipo IASTCompoundStatement. Si un hijo directo es de este tipo significará que existen llaves que encierren todo el contenido del bucle y por lo tanto se podrá realizar el análisis del fichero.

Una vez encontrado un nodo de este tipo, se obtendrán los hijos y se procederá a buscar los diferentes tipos de sentencias que se podrán encontrar dentro de un bucle.

Comprobación de las sentencias dentro del bucle

Un bucle estará formado por un conjunto de sentencias que se irán ejecutando en orden y que realizarán operaciones sobre diferentes variables. Lo que se busca con el análisis del código es encontrar condiciones que hagan que sea imposible la refactorización. Estas condiciones son las siguientes:

- Modificación de variables que se encuentren en la declaración del bucle.
    - En caso de que se modifiquen las variables utilizadas en la declaración del bucle será imposible calcular el número de iteraciones del bucle antes de ejecutarlo y por lo tanto no se podrá llevar a cabo el reparto entre los diferentes hilos.
- Presencia de una sentencia de tipo break.
    - Un bucle paralelizado con OpenMP debe de estar formado por un solo punto de entrada y un solo punto de salida, tras el cual se llevará a cabo la sincronización de los hilos. Una sentencia break rompería el esquema y crearía varios puntos de salida del bucle por lo que la refactorización no sería posible.
- Presencia de una sentencia compuesta sin llaves.
    - Para poder crear el árbol sintáctico es necesario que todas las sentencias compuestas que se encuentren dentro del bucle estén formadas por llaves.

La forma de proceder para comprobar que las sentencias dentro del bucle no generan ninguna situación de las antes citadas será ir recorriéndolas una a una, verificar que tipo de nodo son y llamar al método correspondiente que se encargará de verificar los nodos dentro de la sentencia. Se podrán encontrar los siguientes tipos de nodos:

Expresiones

Estos nodos se corresponderán con el tipo IASTExpressionStatement y serán distintos tipos de operaciones que se realizarán sobre variables. Para comprobar a que tipo de operación hace referencia, será necesario obtener el nodo hijo y verificar su tipo. Se podrán encontrar cuatro tipos de operaciones:

Operación binaria

En caso de que el nodo sea de tipo IASTBinaryExpression, se tratará de una operación binaria formada por una parte izquierda, un operador de asignación y una parte derecha. Para comprobar que variable se está modificando en la operación será necesario obtener todos los hijos.

El primer hijo de la operación siempre se corresponderá con la parte izquierda de la expresión. Se podrán encontrar dos casos diferentes: que el hijo sea de tipo IASTArraySubscriptExpression y por lo tanto se esté modificando un array o bien que sea una variable normal. En caso de que sea un array se llamará a un nuevo método.

El método recibirá el nodo correspondiente al array y se encargará de obtener el índice que se está utilizando para acceder. En caso de que el índice sea una variable que se modifique a través de una operación unaria se comprobará si la variable utilizada es una de las usadas en la declaración del bucle.

En caso afirmativo se estará modificando una de las variables que fijan el número de iteraciones del bucle y por lo tanto no se podrá llevar a cabo la refactorización. Sin embargo, si el índice utilizado es una variable que no se modifica se almacenará en la lista de variables compartidas por todos los hilos.

Si la parte izquierda de la expresión se trata de una variable normal, se obtendrá el nombre de la variable y se comprobará que no se trate de una de las variables usadas en la declaración del bucle. Si es una de las variables que no se pueden reutilizar, no se podrá llevar a cabo la refactorización.

En cuanto a la parte derecha de la expresión, se podrán encontrar operaciones binarias, variables, operaciones unarias o llamadas a funciones.

En caso de que el nodo sea de tipo llamada a función (tipo IASTFunctionCallExpression) se invocará a un método que se encargará de obtener los parámetros usados. Si alguno de los parámetros se corresponde con una de las variables usadas en la declaración del bucle y si se realiza una operación unaria sobre la misma, se estará modificando una de las variables que marcan el número de iteraciones del bucle y por lo tanto la refactorización no podrá llevarse a cabo.

Si el nodo es una variable se llamará a un método que obtendrá el nombre de la misma y comprobará si es alguna de las usadas en la declaración del bucle. Si se trata de

una variable de este tipo se comprobará si se está realizando alguna operación unaria sobre ella. En caso de ser así, no se podrá realizar la refactorización del bucle al estar modificándose el número de iteraciones.

El método también se encargará de localizar posibles operaciones de reducción. Son operaciones de reducción aquellas operaciones binarias que tienen la siguiente forma [14]:

var operador_asignación expresión o bien ***var = var operador expresión*** o ***var = expresión operador var*** donde:

- var es una variable entera.
- operador_asignación es una operador de asignación de uno de los siguientes tipos:
  - +=, -=, *=, &=, ^=, |=.
- operador es un operador de uno de los siguientes tipos:
  - +, –, *, &, |, &&, ||.

En caso de encontrar una operación que siga alguno de esos patrones se comprobará en una lista si ya se ha encontrado anteriormente alguna otra operación de reducción. OpenMP solo permite usar un tipo de operador en la cláusula de reducción, por lo que si se ha encontrado anteriormente algún otro tipo de operación de reducción y además era del mismo tipo que la operación actual se añadirá la variable a la lista de variables de reducción. Si la lista de variables de reducción está vacía se añadirá directamente la variable actual a la lista de reducción.

Operación unaria

En caso de que el tipo del nodo sea IASTUnaryExpression se tratará de una operación unaria en la cual se está incrementando o disminuyendo una variable. El método se encargará de obtener el nombre de la variable modificada y comprobará que no sea ninguna de las utilizadas en la declaración del bucle.

En caso de ser una de las variables, no se podrá llevar a cabo la refactorización del bucle, ya que se está modificando el número de iteraciones.

Por otro lado, el método también se encargará de buscar posibles operaciones de reducción. En primer lugar se observará la lista de variables de reducción y se comprobará si ya hay alguna en la lista. En caso de existir alguna, se obtendrá el operador almacenado en la operación de reducción encontrada anteriormente y se comparará con el utilizado en la operación unaria. En caso de ser del mismo tipo se almacenará la variable modificada en la operación en la lista de variables de reducción.

Llamada a función

En caso de que el tipo del nodo sea IASTFunctionCallExpression se tratará de una llamada a una función. El método se encargará de obtener los parámetros que se utilizan en la llamada a la función y comprobar si se utiliza alguna de las variables usadas en la declaración del bucle. En caso afirmativo se observará si se lleva a cabo una operación unaria sobre las variables utilizadas como parámetros. Si se está realizando una operación significa que se está modificando una de las variables que marcan el número de iteraciones del bucle y por lo tanto no se puede llevar a cabo la refactorización.

Lista de expresiones

En caso de que el tipo del nodo sea IASTExpressionList se tratará de una lista de expresiones. El método se encargará de obtener los hijos de la lista e ir llamando a los métodos adecuados según el tipo de cada uno de ellos.

Sentencias condicionales

Estos nodos se corresponderán con el tipo IASTIfStatement y serán sentencias condicionales donde el flujo del programa variará en función de si se cumple la condición.

El método solo tendrá que buscar en el cuerpo de la sentencia si existe alguna situación comprometida para llevar a cabo la refactorización; no tendrá que mirar en la condición, ya que bajo ninguna circunstancia se realizará una modificación de variable.

En primer lugar obtendrá los hijos de la sentencia y buscará un nodo de tipo IASTCompoundStatement. En caso de que no lo encuentre, significará que el cuerpo de la sentencia no está encerrado entre llaves y por lo tanto no se podrá llevar a cabo la refactorización.

Una vez encontrado el nodo de tipo IASTCompoundStatement se obtendrán sus hijos y se procederá a comprobar el tipo de cada uno de ellos. Según los tipos que se vayan encontrando se irán llamando a los métodos correspondientes y se llevará a cabo la misma lógica que si fuesen invocados desde un hijo principal del bucle.

Sentencias de bucle for

Estos nodos se corresponderán con el tipo IASTForStatement y serán bucles for anidados dentro del bucle principal que se quiere refactorizar. Al igual que ocurría en el bucle principal, los tres primeros hijos se corresponderán con las variables utilizadas en la declaración del bucle.

En primer lugar se observará si en la declaración del bucle se modifica alguna de las variables utilizadas en la sentencia del bucle principal; en caso de ser así, la refactorización no se podrá realizar, ya que se está alterando una de las variables que marca el número de iteraciones.

A continuación se buscará un nodo de tipo IASTCompoundStatement, ya que para que la refactorización se pueda realizar correctamente es necesario que el cuerpo del bucle esté encerrado entre llaves. Una vez encontrado se obtendrán los hijos del nodo y se irán comprobando sus tipos uno a uno. Según los tipos que se vayan encontrando se irán llamando a los métodos correspondientes y se llevará a cabo la misma lógica que si fuesen invocados desde un hijo principal del bucle.

Sentencias de bucle while

Estos nodos se corresponderán con el tipo IASTWhileStatement y serán bucles *while* anidados dentro del bucle principal. Al igual que pasaba con las sentencias condicionales no será necesario buscar en la condición del bucle, ya que bajo ningún concepto se realizará una modificación de variables.

En primer lugar será necesario buscar un nodo de tipo IASTCompoundStatement, ya que para que la refactorización pueda realizarse será necesario que el cuerpo del bucle esté encerrado entre llaves.

Una vez encontrado el nodo se obtendrán sus hijos y se irán comprobando sus tipos uno a uno. Según los tipos que se vayan encontrando se irán llamando a los métodos correspondientes y se llevará a cabo la misma lógica que si fuesen invocados desde un hijo principal del bucle.

Sentencias de bucle do-while

Estos nodos se corresponderán con el tipo IASTDoStatement y serán bucles *do-while* anidados dentro del bucle principal. Al igual que en los bucles while y en las sentencias condicionales no será necesario buscar en la condición del bucle, ya que nunca se realizará una modificación de variables.

La lógica será la misma que en los bucles while: buscar un nodo de tipo IASTCompoundStatement, obtener sus hijos e ir comprobando sus tipos.

Sentencias de declaración de variables

Estos nodos se corresponderán con el tipo IASTDeclarationStatement y serán declaraciones de variables internas al bucle. Se tendrá que diferenciar entre declaraciones sin inicialización y declaraciones con inicialización.

En caso de que sea una declaración con inicialización, se obtendrá por un lado el nombre de la variable que se declara y por otro lado las variables que se encuentren en

el lado derecho de la operación. En el caso de una declaración sin inicialización, se obtiene el nombre de la variable y se termina.

Una vez que se dispone del nombre de la variable se almacena en una lista de variables internas. Dichas variables se almacenan en una lista aparte de las variables compartidas y de las variables privadas porque no hace falta declararlas en la directiva de OpenMP.

Sentencias break

Estos nodos se corresponderán con el tipo IASTBreakStatement y serán sentencias break dentro del bucle principal. Estas sentencias no pueden existir en un bucle paralelo, ya que solo puede existir un punto de entrada y un punto de salida. Por lo tanto, si se encuentra un nodo de este tipo la refactorización no será posible.

Refactorización

Una vez que se ha realizado las comprobaciones necesarias, el complemento realizará diferentes acciones según el tipo de refactorización escogida y el resultado del análisis de las sentencias.

Refactorización automática

En el caso de la refactorización automática si el resultado de las comprobaciones ha sido que se puede realizar se procederá a crear la directiva OpenMP que luego se escribirá en el fichero. Para ello, se usará la lista de variables privadas y compartidas y se irán recorriendo todos los elementos que contengan, de manera que se vayan almacenando en un objeto los nombres de las variables.

En caso de que existan variables de reducción se obtendrá el operador de reducción a utilizar y las variables que estén contenidas en la lista y se añadirán al objeto. En caso de que existan las mismas variables tanto en la lista de reducción como en la lista de variables compartidas significará que después de añadirlas a la lista de variables de reducción se ha encontrado una operación que modifica el valor de dicha variable y por lo tanto no se puede tratar como una operación de reducción. Para

controlar esta situación se recorrerá ambas listas y los elementos que estén en ambas se borrarán de las lista de reducción.

Por último se obtendrá el fichero que está abierto actualmente en el editor por defecto, que siempre será el último fichero sobre el que se ha llevado la refactorización y se escribirá la directiva OpenMP en la posición adecuada. También se comprobará si anteriormente se ha llevado a cabo la refactorización de algún bucle dentro de este fichero. En caso de que no haya sido así, se escribirá al inicio del fichero unas directivas de preprocesador en las cuales se comprueba si el código va a ejecutarse utilizando librerías de OpenMP; es decir, si se va a ejecutar de forma paralela o de forma secuencial. En caso de que se ejecute de forma secuencial, estas directivas harán que se eviten todas las llamadas a funciones de OpenMP de manera que se ejecute de forma correcta. Si la refactorización es correcta, el fichero se eliminará de la vista de elementos a refactorizar si fue añadido anteriormente.

En caso de que el resultado de las comprobaciones sea que existen errores y que no se puede llevar a cabo la refactorización, se recorrerán las listas que almacenan los errores y se irán mostrando en diálogos de alerta al usuario uno por uno, indicando el número de línea donde se encuentre el error.

Refactorización semiautomática

La refactorización semiautomática es un poco distinta a la automática ya que permite personalizar la directiva OpenMP. Para permitir que el usuario modifique algunos parámetros se utiliza un asistente formado por varias páginas que se lanzará una vez pulsado el botón de refactorización semiautomática sobre cualquier fichero.

Una vez que el usuario haya seleccionado las opciones personalizadas se llevará a cabo el análisis del fichero seleccionado y se mostrará la directiva creada antes de insertarla en el fichero fuente. Para ello, la clase encargada de realizar el análisis no escribirá directamente en el fichero fuente, sino que almacenará en un array las diferentes listas de variables compartidas, privadas y de reducción. A continuación, se

mostrará al usuario la directiva OpenMP y en caso de que acepte se insertará en el fichero. Si la refactorización es correcta, el fichero se eliminará de la vista.

En caso de que la refactorización encuentre errores el asistente no se lanzará y se recorrerán las listas que almacenan los errores y se irán mostrando en diálogos de alerta al usuario uno por uno, indicando el número de línea donde se encuentre el error.

4.4 EJEMPLOS DE USO

A continuación se van a mostrar algunos ejemplos de uso del complemento en algoritmos implementados en el lenguaje C.

CÁLCULO DEL NÚMERO Π

El cálculo de los decimales del número π es una operación bastante costosa, ya que aunque existen numerosos algoritmos que realizan dicho cálculo, todos ellos se basan en la ejecución de un gran número de iteraciones de un bucle.

El código inicial del algoritmo, obtenido de [12] puede verse en la Figura 48:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
	double local, w, pi;
	long i, N
	N = 100000;
	w = 1.0 / N;
	pi = 0.0;
	long j, k;

	for(i = 0; i < N; i++) {
		local = (i + 0.5) * w;
		pi += 4.0 / (1.0 + local * local);
	}
	pi *= w;
	printf("Número pi: %.32lf", pi);
	return 0;
}
```

Figura 48 - Código secuencial de cálculo del número PI

El número de iteraciones variará en función de la precisión que se quiere obtener y por lo tanto el número de decimales correctos calculados. Si se realiza el cálculo con 100.000 iteraciones, el resultado es el siguiente número: 3.14159265359816153306837804848328 el cual contiene los diez primeros decimales correctos pero luego comienza a cometer errores.

Para conseguir un mejor resultado, será necesario aumentar la precisión. Si se selecciona una precisión de 100.000.000 de iteraciones, el resultado no varía mucho respecto al anterior. Por ello, para conseguir un resultado muy exacto, será necesario utilizar un mínimo de 10.000.000.000 iteraciones, que es un número grande y será costoso de ejecutar en un ordenador personal.

Utilizando el complemento desarrollado en el proyecto se puede llevar a cabo la refactorización del código para introducir directivas OpenMP. En este caso al ser un bucle simple se utiliza la refactorización automática, dejando en manos del complemento decidir los parámetros más adecuados.

Las directivas introducidas por el complemento pueden verse en la Figura 49:

```
#ifdef OPENMP
	#include <omp.h>
#else
	#define omp_set_num_threads(x) 0
#endif

…

int main(int argc, char *argv[]) {
	…

	omp_set_num_threads(omp_get_max_threads());
	#pragma omp parallel for default(none) \
	schedule(static) \
	shared(N,w) private(i,local) \
	reduction(+:pi)

	for(i = 0; i < N; i++) {
	 …
	 }
}
```

Figura 49 - Código PI paralelo

Una vez que se dispone del código refactorizado, se podrá compilar y ejecutar y se apreciará la diferencia de tiempo de ejecución del modelo secuencial al paralelo.

DINÁMICA DE MOLÉCULAS

Los programas que simulan el comportamiento de las moléculas en el espacio generan una carga computacional enorme ya que los algoritmos que utilizan se basan en cálculos diferenciales independientes para calcular el movimiento de cada una de las partículas existentes.

Este tipo de programas tienen una posible ejecución paralela enorme, ya que están formadas por numerosos bucles que realizan cálculos independientes cuyas iteraciones pueden repartirse sin ningún problema entre diferentes hilos para acelerar el cálculo final.

El código inicial del algoritmo, obtenido de [12] y siendo ligeramente modificado puede verse en la Figura 50:

```
# include <stdlib.h>
# include <stdio.h>
# include <time.h>
# include <math.h>

int main ( int argc, char *argv[] )
{
      …
}

void compute ( int np, int nd, double pos[], double vel[],
             double mass, double f[], double *pot, double *kin )

{
      …

      for ( k = 0; k < np; k++ )
      {
            …
      }

}

double cpu_time ( void )

{
      …
}

double dist ( int nd, double r1[], double r2[], double dr[] )

{
      …
}

void initialize ( int np, int nd, double box[], int *seed, double pos[],
             double vel[], double acc[] )

{
      …
}

double r8_uniform_01 ( int *seed )

{
      …
}

void timestamp ( void )

{
…}

void update ( int np, int nd, double pos[], double vel[], double f[],
             double acc[], double mass, double dt )

{
```

```
        for ( j = 0; j < np; j++ )
        {
              …
        }

}
```

Figura 50 - Cálculo de dinámica de moléculas

Como se puede ver es un código bastante complejo que está formado por numerosos bucles, algunos de los cuales se pueden ejecutar de forma paralela. Al contrario que en el caso de cálculo del número π, con este programa la refactorización debe realizarse de forma semiautomática, ya que no todos los bucles deben ejecutarse de forma paralela y con este modo de ejecución tendremos un mayor control sobre las zonas a refactorizar.

Se puede observar que los bucles más susceptibles de ser paralelizados son el que se encuentra dentro del método compute, que es el que realiza los cálculos de todas las moléculas y el bucle del método update, el cual actualiza la posición, la velocidad y la aceleración de cada una de las partículas. Por lo tanto, pinchamos en realizar la refactorización semiautomática del fichero con el código fuente y cancelamos aquellos bucles que no se quieren refactorizar. El resultado final es puede verse en la Figura 51:

```
#ifdef OPENMP
#include <omp.h>
#else
#define omp_set_num_threads(x) 0
#endif

…

int main ( int argc, char *argv[] )
{
   …
}

void compute ( int np, int nd, double pos[], double vel[],
              double mass, double f[], double *pot, double *kin )

{
       …

       omp_set_num_threads(omp_get_max_threads());
        #pragma omp parallel for \
              schedule(static) \
        private(k,i,j,d,d2) shared(np,nd,f,PI2,vel) \
        reduction(+:pe,ke)

       for ( k = 0; k < np; k++ )
       {
              …
       }

        …
       return;
}

double cpu_time ( void )

{
      …
}

double dist ( int nd, double r1[], double r2[], double dr[] )

{
       …
}

void initialize ( int np, int nd, double box[], int *seed, double pos[],
              double vel[], double acc[] )

{
      …
}

double r8_uniform_01 ( int *seed )
```

```
{
     …
}

void timestamp ( void )

{
     …
}

void update ( int np, int nd, double pos[], double vel[], double f[],
              double acc[], double mass, double dt )

{
       …

       omp_set_num_threads(omp_get_max_threads());
       #pragma omp parallel for \
       schedule(static) \
       private(j,i) shared(np,nd,pos,vel,dt,acc,f,rmass)

       for ( j = 0; j < np; j++ )
       {
              …
       }

       …
}
```

Figura 51 - Código de dinámica de moléculas refactorizado

Al igual que en el caso anterior, ahora se podrá compilar y ejecutar y se verá un gran aumento del rendimiento del programa.

5. CONCLUSIONES

A continuación, se describirán las conclusiones a las que se ha llegado una vez terminado el proyecto de fin de grado, junto con las mejoras que se incluirán en futuras versiones del complemento y el presupuesto.

5.1 FUTURAS MEJORAS

El complemento podría mejorarse en futuras versiones para añadir nuevas funcionalidades que facilitasen aún más el trabajo de los desarrolladores de software a la hora de crear programas paralelos. Las posibles mejoras que se podrían incluir en versiones posteriores son las siguientes:

En primer lugar, se podría ampliar la refactorización a otras sentencias que no sean bucles. Para ello habría que buscar patrones de programación paralela en el código a refactorizar. Una vez encontrado alguno de los patrones existentes, se realizaría un análisis del AST y se insertarían las directivas adecuadas de OpenMP.

En segundo lugar, se podría ampliar los modelos de programación paralela utilizados en el complemento, ya que actualmente solo inserta directivas de tipo OpenMP. No sería difícil añadir soporte para otros tipos de directivas parecidas a OpenMP que podrían ser más eficientes en algunos casos de ejecución de código paralelo.

En tercer lugar se podría añadir un mecanismo que realizase una refactorización provisional del código y comprobase el incremento de rendimiento que se obtendría en la versión paralela respecto a la versión secuencial. De esta manera, el usuario podría observar si merece la pena realizar una versión paralela del código antes de realizar ninguna acción.

En cuarto lugar se podrían ampliar las opciones de personalización de la directiva a insertar en el modo de refactorización semiautomático. De esta manera, el

usuario podría aprovechar otras cláusulas que dispone OpenMP para crear un programa óptimo.

En quinto lugar se podría mejorar el modo de refactorización automática de forma que seleccionase el mejor reparto de trabajo entre los hilos y no fuese siempre de forma estática. De esta forma la ejecución del código paralelo sería más óptima.

5.2 PRESUPUESTO

A continuación se detalla el presupuesto calculado para el desarrollo de este trabajo de fin de grado. Para calcular el costo de recursos humanos se ha utilizado la planificación del punto 4.1 con una jornada de 5 horas diarias a lo largo de 91 días. En cuanto al software, se ha intentado utilizar en la medida de lo posible software libre. Por otro lado el coste del hardware y del software que se ha utilizado para desarrollar el proyecto se ha calculado en función de la amortización del mismo y del periodo de uso que se le ha dado.

El personal necesario para desarrollar el proyecto estaría formado por un analista y diseñador, encargado de obtener los requisitos, realizar el análisis funcional, diseñar la aplicación, realizar las pruebas y documentar todo el proceso y por un programador que implementará el código del complemento [74].

En cuanto al software los sistemas operativos que se han usado han sido Ubuntu 11.10 para desarrollar el complemento y para realizar pruebas y Windows 7 Profesional para crear la documentación. Se ha utilizado el IDE Eclipse versión 4.2 (Juno) para desarrollar el complemento, Microsoft Office 2010 Hogar y Estudiantes como suite ofimática y la herramienta DIA para crear los esquemas.

En cuanto al hardware se ha utilizado un ordenador portátil Acer Aspire 5750G tanto para desarrollar el complemento como para realizar la documentación. El ordenador portátil presenta las siguientes características:

- Procesador Intel Core i7 2630QM.
- Tarjeta gráfica NVIDIA GeForce GT 540M.
- 4 GB de memoria RAM.
- 500 GB de disco duro.

PRESUPUESTO DE PROYECTO

1.- Autor:	Miguel Olmedo Camacho
2.- Departamento:	Informática
3.- Descripción del Proyecto:	Creación de un complemento que permita refactorizar código C a código C con OpenMP
- Titulo	Diseño y evaluación de un complemento para refactorización paralela de código C usando OpenMP
- Duración (meses)	**3**
Tasa de costes Indirectos:	**20%**

4.- Presupuesto total del Proyecto (valores en Euros):

Euros Diez mil cincuenta y nueve

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F.	Categoría	Dedicación (hombres mes)	Coste hombre mes	Coste (Euro)
Olmedo Camacho, Miguel		Analista y diseñador	2,05	3.000,00	6.150,00
Olmedo Camacho, Miguel		Programador	1,4	1.416,00	1.982,40
					0,00
					0,00
					0,00
		Hombres mes	**3,45**	**Total**	8.132,40

EQUIPOS Y SOFTWARE

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable
Acer Aspire 5750G	699,00	100	3	60	34,95
Licencia Windows 7	119,99	100	3	36	10,00
Licencia Microsoft Office 2010	118,00	100	3	36	9,83
Eclipse IDE	0,00	100	3	36	0,00
Ubuntu 11.10	0,00	100	3	36	0,00
DIA	0,00	100	3	36	0,00
				Total	54,78

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
	Total	0,00

OTROS COSTES DIRECTOS DEL PROYECTO

Descripción	Empresa	Costes imputable
Internet	Telefónica	75
Electricidad	Endesa	120
	Total	195,00

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	8.132
Amortización	55
Subcontratación de tareas	0
Costes de funcionamiento	195
Costes Indirectos	1.637
Total	10.059

5.3 CONCLUSIONES PERSONALES

Como conclusión personal he de decir que el proyecto ha resultado muy útil e interesante para aprender tanto el funcionamiento de OpenMP como de la plataforma PDE para desarrollo de complementos en Eclipse.

Tras el trabajo realizado es fácil ver porque Eclipse cuenta con una comunidad tan grande que lo apoya y lo amplia continuamente en búsqueda de nuevas funcionalidades. Las facilidades que aporta la propia plataforma para desarrollar complementos hacen que los usuarios puedan generar de forma sencilla funcionalidades específicas para los problemas que van surgiendo en el proceso de creación de software. De esta manera se va facilitando el proceso de desarrollo y otros usuarios pueden aprovecharse de estas nuevas características creadas.

Ha sido interesante estudiar los diferentes modelos de programación paralela y ver como día a día va cambiando la orientación del desarrollo de software hacia una nueva perspectiva, donde uno de los objetivos principales es aprovechar tanto el hardware actual como el venidero, creando programas escalables que aprovechan todos los recursos disponibles.

En cuanto a la API de OpenMP me han sorprendido las facilidades que aporta al usuario para conseguir programas muy eficientes insertando no más de dos líneas de código. El aprendizaje de todas las directivas con sus respectivas cláusulas no requiere un gran esfuerzo por parte del usuario por lo que se puede aprender a utilizar rápidamente. Sin duda OpenMP irá evolucionando en futuras versiones y se irá adaptando al nuevo hardware, consiguiendo un rendimiento aún más óptimo de los programas.

Lo más complejo ha sido el análisis del árbol de sintaxis abstracta debido a la multitud de tipos de sentencias que existen en un programa escrito en lenguaje C. No podía quedar ninguna posibilidad sin cubrir por lo que había que tratar todos los caminos que podía tomar el programa.

El proyecto de fin de grado ha sido una experiencia que me ha proporcionado una nueva forma de pensar a la hora de desarrollar software, intentando aprovechar las características del hardware presente y futuro.

6. ANEXO I: GUÍA DE INSTALACIÓN

A continuación, se mostrará una guía de instalación de todos los elementos necesarios para hacer funcionar el complemento desarrollado en este proyecto de fin de grado.

INSTALACIÓN DEL IDE ECLIPSE

Para poder utilizar el complemento es necesario que el usuario disponga del IDE Eclipse instalado en su equipo. Se recomienda la última versión para asegurar la compatibilidad entre el complemento y el IDE. En caso de que el usuario no disponga del IDE descargado, será necesario acceder a la web de eclipse y descargarlo. Dado que el complemento desarrollado necesita que estén instaladas las herramientas de CDT [53] para funcionar, se recomienda descargar la versión que integra dicho complemento, es decir, *Eclipse IDE for C/C++ Developers.*

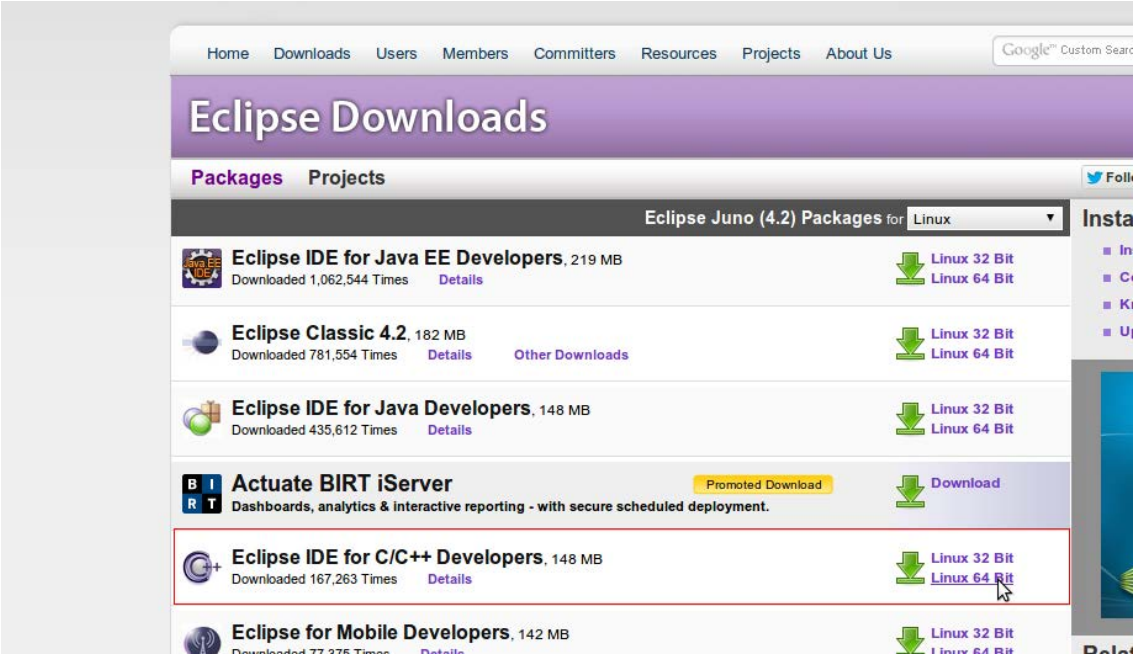


Figura 52 - Descarga de Eclipse

Una vez descargado, bastará con extraerlo para poder utilizarlo.

INSTALACIÓN DEL COMPLEMENTO CDT

En caso de que el usuario descargue una versión de eclipse que no integre el complemento CDT, será necesario instalarlo. Para llevar a cabo la instalación, el usuario deberá de pulsar sobre el menú *Help* y a continuación sobre *Install New Software*….

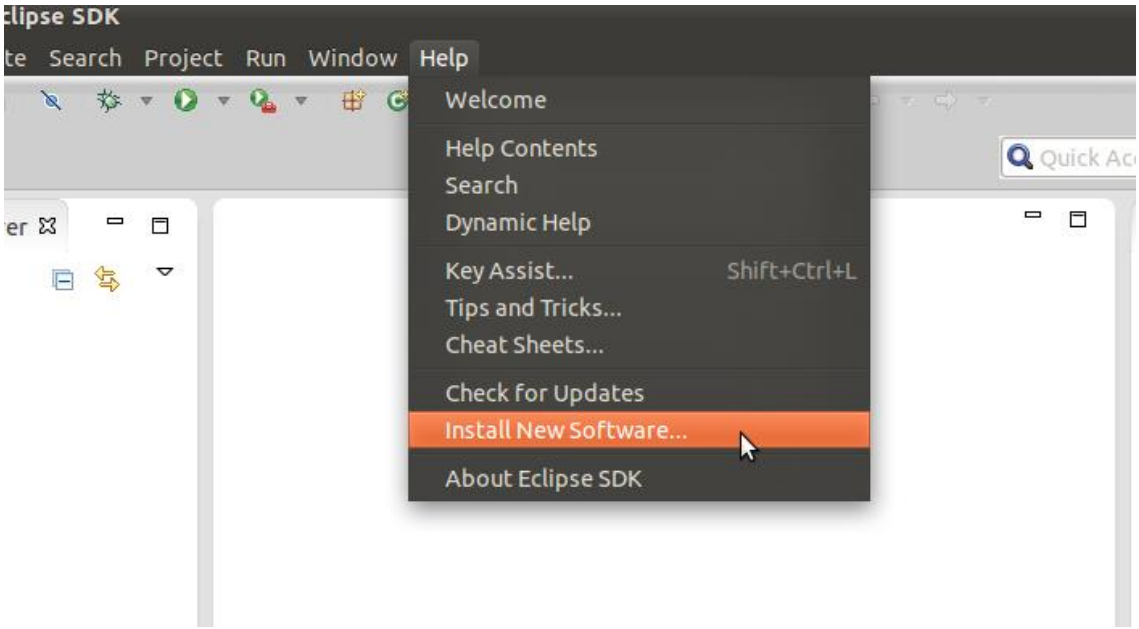


Figura 53 - Instalar nuevo software

A continuación, el usuario deberá de seleccionar en la lista desplegable de *Work With* la opción Juno-http://download.eclipse.org/releases/juno para el caso de la última versión de eclipse. Si se utilizase una versión anterior, en vez de Juno aparecería la versión instalada.

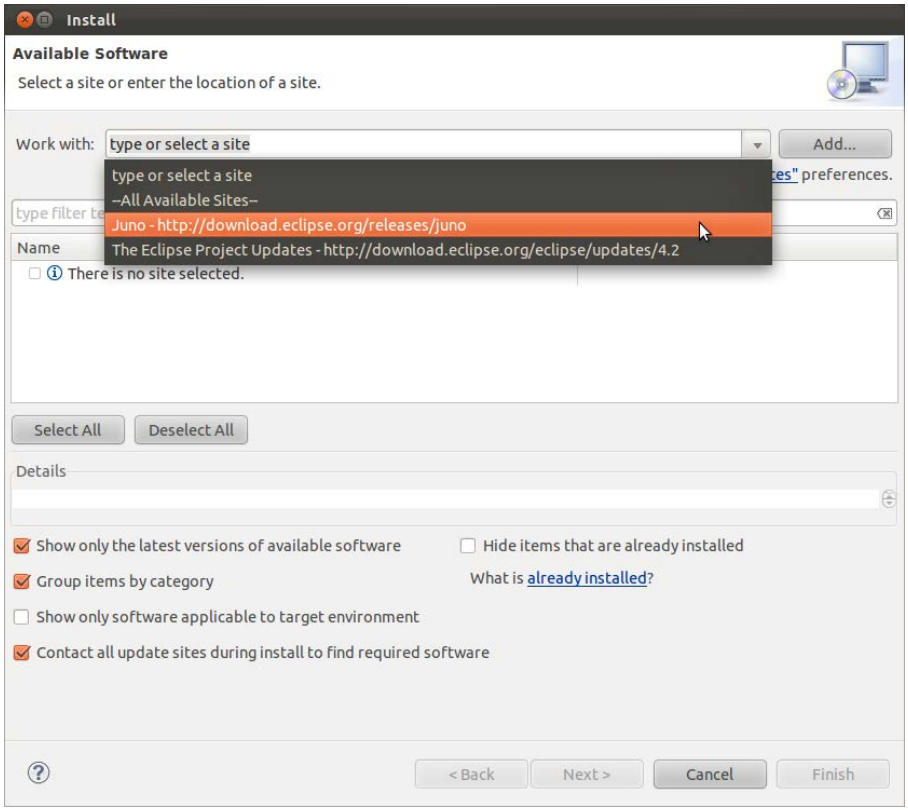


Figura 54 - Seleccionar sitio web desde donde descargar nuevo software

Una vez seleccionado el sitio, aparecerá en la pantalla el nuevo software que se podrá instalar en el IDE. En nuestro caso, habrá que acceder a *Programming Languages* y seleccionar *C/C++ Development Tools.*

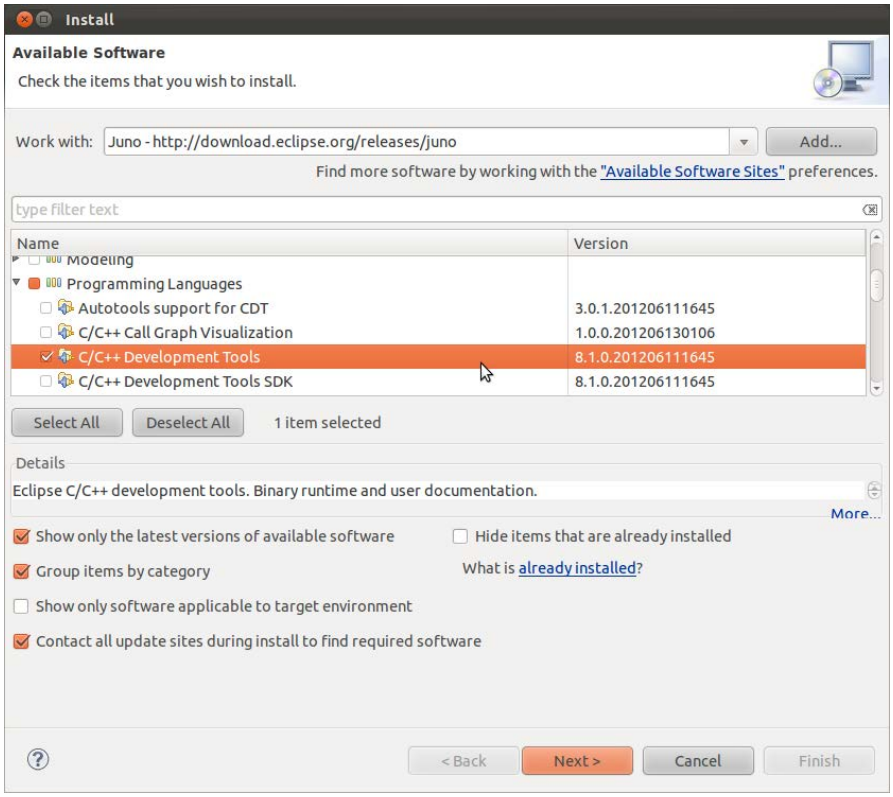


Figura 55 - Selección del complemento CDT

Una vez que se descargue el complemento, se instalará automáticamente y se pedirá al usuario que reinicie el IDE.

INSTALACIÓN DEL COMPLEMENTO DESARROLLADO

Para instalar el complemento en el IDE Eclipse, habrá que acceder al directorio *Aplicación* dentro del CD. A continuación, bastará con extraer el contenido del fichero denominado Aplicacion.zip en la carpeta raíz del eclipse. Dicho fichero ya contiene las rutas necesarias para guardar el complemento, almacenado en un jar, en la carpeta adecuada.

7. ANEXO II: GUÍA DEL USUARIO

A continuación se encuentra una guía para orientar al usuario en el uso del complemento desarrollado en este proyecto de fin de grado.

PRERREQUISITOS

Antes de poder utilizar el complemento el usuario debe haber leído el apartado Anexo I: Guía de instalación y tener instalados todos los componentes necesarios.

La refactorización de ficheros se puede llevar a cabo usando la vista destinada para ello o directamente a través del menú contextual al seleccionar archivos con código fuente. Se recomienda añadir en primer lugar los ficheros a la vista para llevar un mejor control sobre los archivos a refactorizar.

Para que aparezca la vista de refactorización por la que está formado el complemento el usuario tendrá dos opciones:

En primer lugar, el usuario tendrá que acceder al menú *Window*, luego deberá pinchar sobre *Show View* y por último sobre *Other…* o presionar Shift+Alt+Q y a continuación Q.

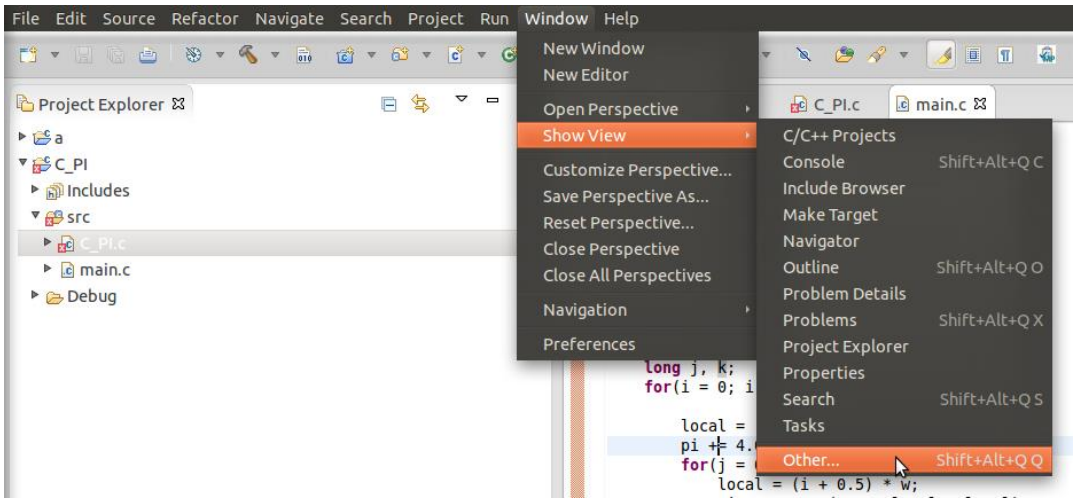


Figura 56 - Menú para abrir la vista de refactorización

En el nuevo menú que se abrirá existirá una carpeta denominada *Refactorización de C a C con OpenMP* y dentro de la carpeta se encontrará la vista *Ficheros a refactorizar*. Si se pulsa sobre dicha vista, se abrirá en la parte inferior la vista a través de la cual se podrá realizar la refactorización de ficheros.

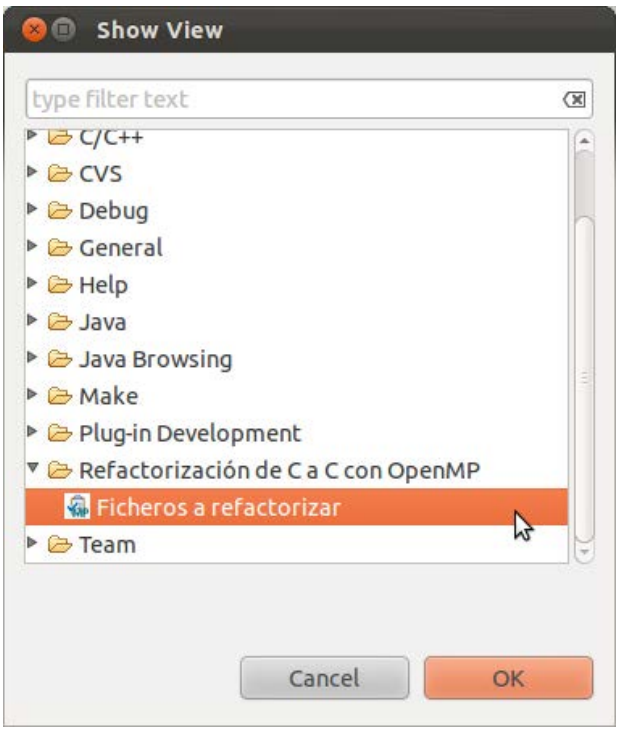


Figura 57 - Abrir la vista de refactorización

Como segunda opción, el usuario podrá pulsar un botón que se encuentra en la barra de herramientas del banco de trabajo original de Eclipse. Una vez pulsado el botón, se abrirá la vista de refactorización.

Figura 58 - Abrir vista de refactorización

Para poder llevar a cabo la refactorización deberá de existir en el espacio de trabajo un proyecto de lenguaje C que contenga ficheros de código fuente.

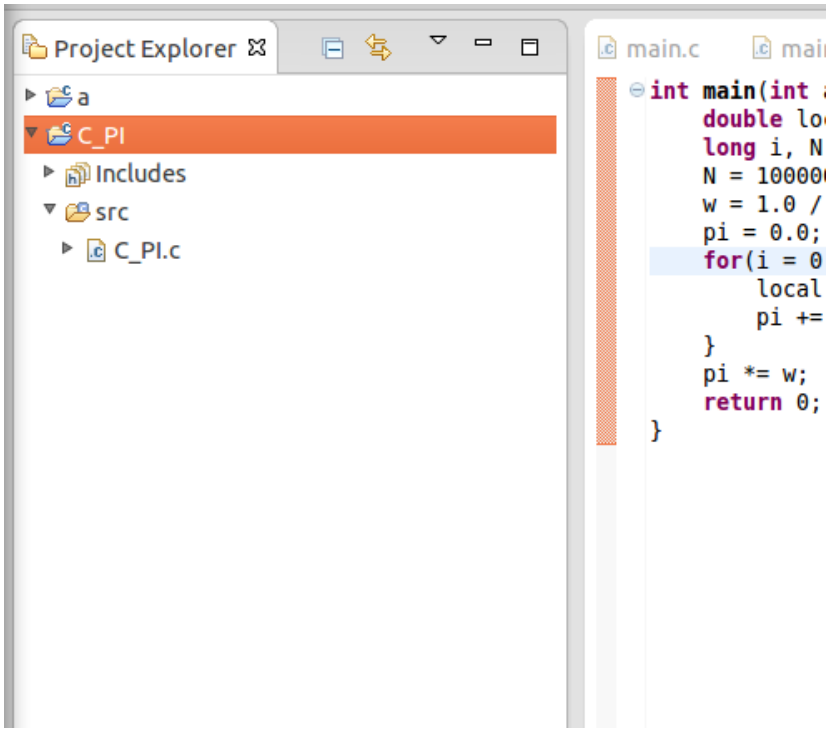


Figura 59 - Proyecto con ficheros de código fuente C

AÑADIR FICHEROS A LA VISTA DE REFACTORIZACIÓN

Para añadir los ficheros a refactorizar a la vista de refactorización se seleccionaran los ficheros con código fuente, se pinchará con el botón derecho sobre ellos, se seleccionará el menú *OpenMP* y a continuación se seleccionará la opción de *Añadir a la vista de refactorización*.

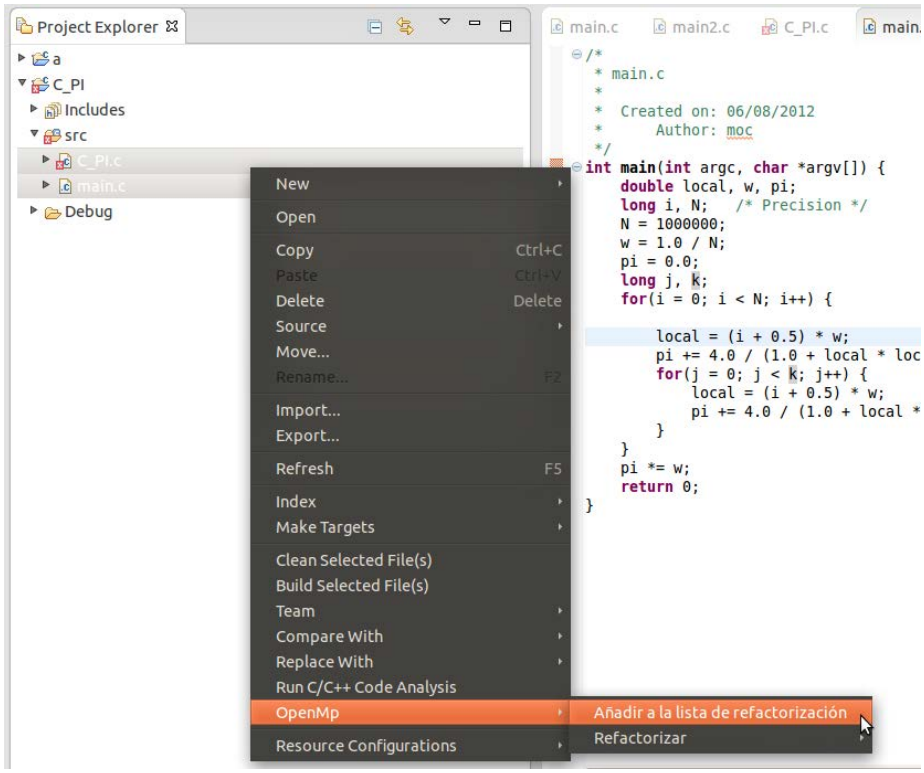


Figura 60 - Añadir ficheros a la vista de refactorización

BORRAR FICHEROS DE LA VISTA DE REFACTORIZACIÓN

Para borrar ficheros de la vista de refactorización el usuario dispondrá de dos opciones:

A través del botón de la barra de herramientas

El usuario deberá seleccionar los ficheros que quiera borrar y a continuación pulsar el botón que se encuentra en la barra de herramientas.

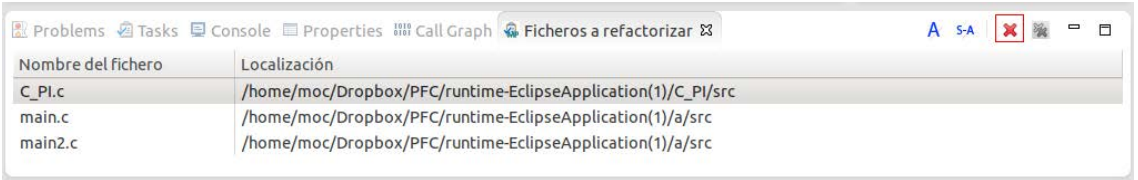


Figura 61 - Eliminar ficheros

A través del menú contextual

El usuario deberá seleccionar los ficheros que quiera borrar y pulsar el botón derecho para abrir el menú contextual. Una vez abierto el menú, el usuario tendrá que seleccionar la opción *Borrar*.

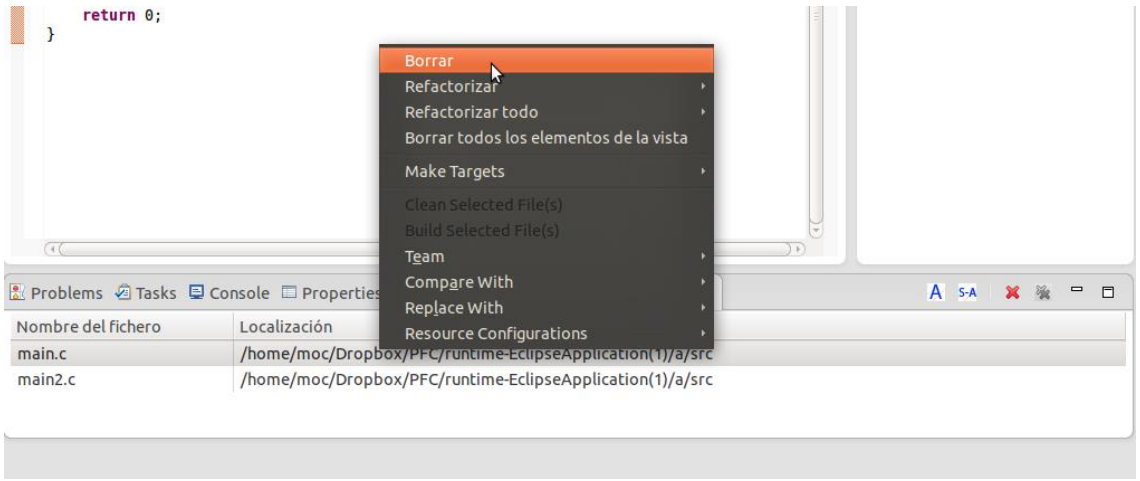


Figura 62 - Eliminar ficheros

BORRAR TODOS LOS FICHEROS DE LA VISTA

Para borrar todos los ficheros que se encuentren actualmente en la vista de refactorización, el usuario dispondrá de dos opciones:

A través del botón de la barra de herramientas

El usuario deberá pulsar el botón que se encuentra en la barra de herramientas.

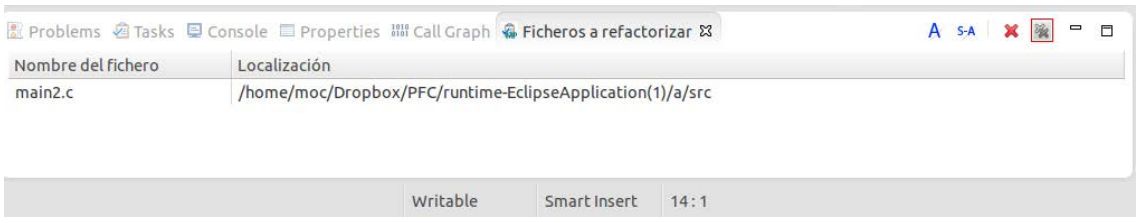


Figura 63 - Eliminar todos los ficheros de la vista

REFACTORIZAR FICHEROS DE FORMA AUTOMÁTICA

El usuario podrá refactorizar ficheros de forma automática a través de la vista o directamente desde el menú contextual del explorador de proyectos.

El usuario dispondrá de dos opciones para refactorizar los ficheros a través de la vista de refactorización: a través de la barra de herramientas de la vista o a través del menú contextual.

Si el usuario quiere utilizar el botón de la barra de herramientas deberá seleccionar los ficheros a refactorizar y a continuación pulsar el botón en la barra de herramientas.

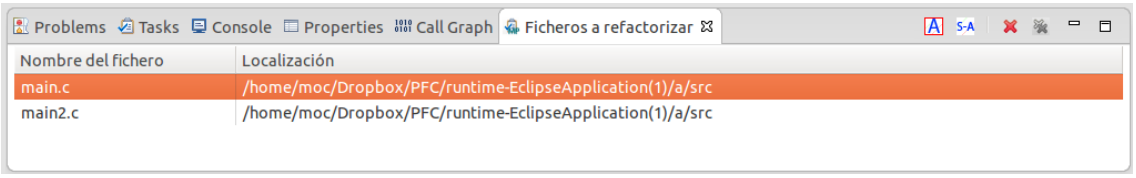


Figura 64 - Refactorizar ficheros de forma automática

Si el usuario quiere utilizar el menú contextual de la vista deberá seleccionar aquellos ficheros que quiere refactorizar y a continuación pulsar el botón derecho. Una vez que se abra el menú contextual, el usuario deberá meterse en el menú de *Refactorizar* y pulsar la opción de *Refactorización automática*.

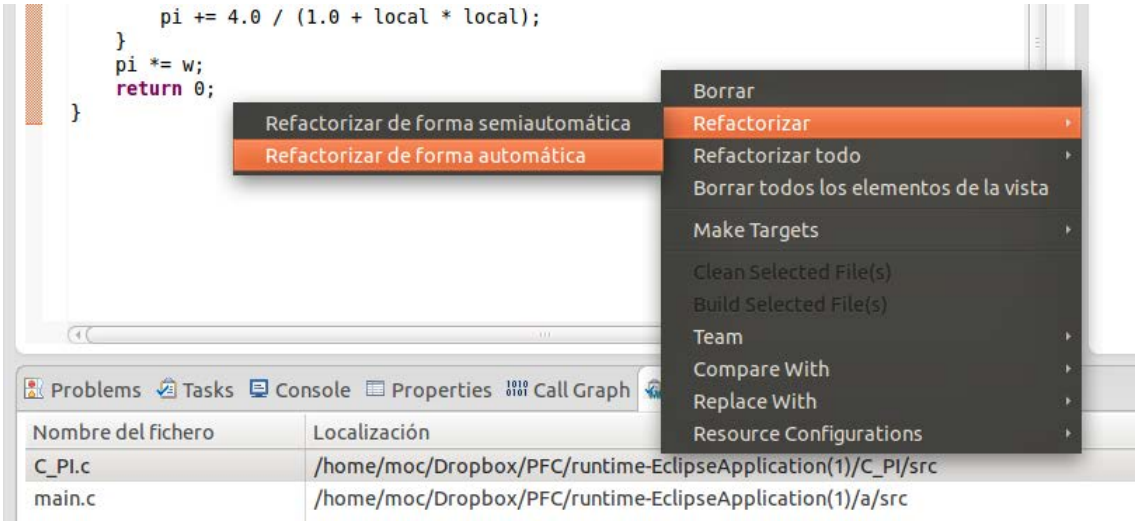


Figura 65 - Refactorizar ficheros de forma automática

Para refactorizar un fichero a través del menú contextual del explorador de proyectos bastará con que el usuario seleccione los ficheros a refactorizar y pulse el botón derecho. En el menú contextual deberá dirigirse al menú *OpenMP* y dentro de éste a *Refactorizar*. Dentro del menú Refactorizar será necesario pinchar sobre la opción *Refactorizar de forma automática*.

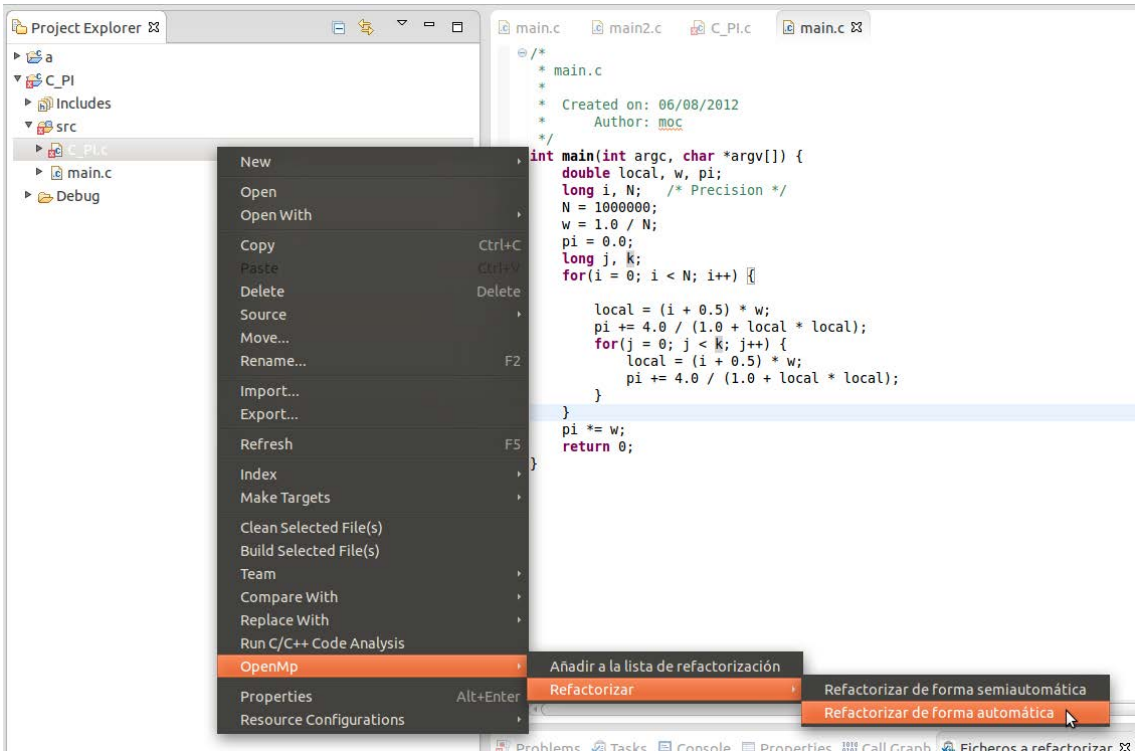


Figura 66 - Refactorizar de forma automática directamente

Una vez realizada la acción de refactorización de una de las tres maneras posibles, se abrirá una nueva ventana que mostrará un diálogo al usuario. Para proceder con la refactorización, el usuario deberá aceptar.

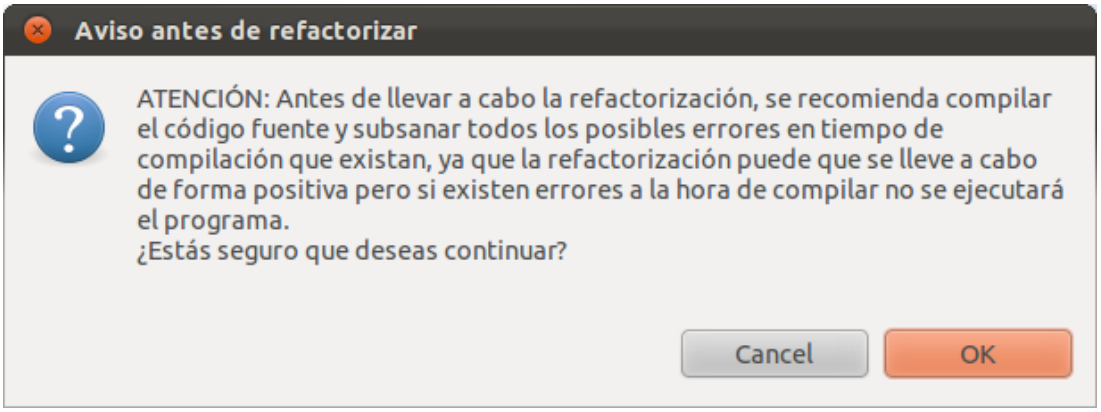


Figura 67 - Aviso antes de refactorizar

En el momento en el que el usuario acepte el aviso se procederá a realizar la refactorización de forma automática de los ficheros seleccionados. En este momento, el programa podrá tomar dos caminos diferentes:

Refactorización correcta

En caso de que la refactorización se haya realizado de forma satisfactoria y no se haya encontrado ningún error, se insertarán de forma automática las sentencias de OpenMP para que los bucles que se encuentren dentro de los ficheros seleccionados se puedan ejecutar de forma paralela.

El fichero con el código fuente modificado se guardará de forma automática y el usuario deberá compilarlo como se indica en la sección Compilación y ejecución del código.

Errores en la refactorización

En caso de que se encuentre algún error en el bucle y no se pueda llevar a cabo la refactorización, se avisará al usuario a través de un diálogo y se mostrará uno de los errores que se encuentran en la sección Errores al refactorizar

REFACTORIZAR FICHEROS DE FORMA SEMIAUTOMÁTICA

El usuario podrá refactorizar ficheros de forma semiautomática a través de la vista o directamente desde el menú contextual del explorador de proyectos.

El usuario dispondrá de dos opciones para refactorizar los ficheros a través de la vista de refactorización: a través de la barra de herramientas o a través del menú contextual.

Si el usuario quiere utilizar el botón de la barra de herramientas, deberá seleccionar los ficheros a refactorizar y a continuación pulsar el botón en la barra de herramientas.

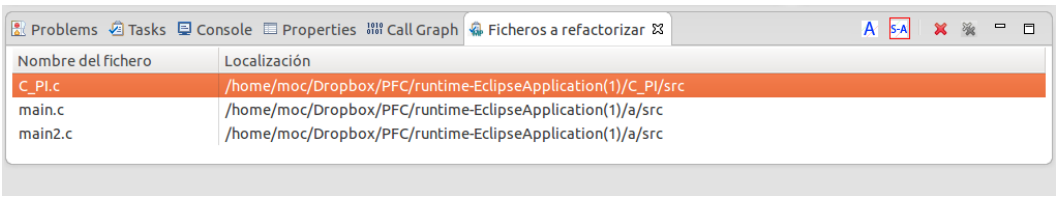


Figura 68 - Refactorizar ficheros de forma semiautomática

Si el usuario quiere utilizar el menú contextual de la vista, deberá seleccionar aquellos ficheros que quiere refactorizar y a continuación pulsar el botón derecho. Una vez que se abra el menú contextual, el usuario deberá meterse en el menú de *Refactorizar* y pulsar la opción de *Refactorización semiautomática*.

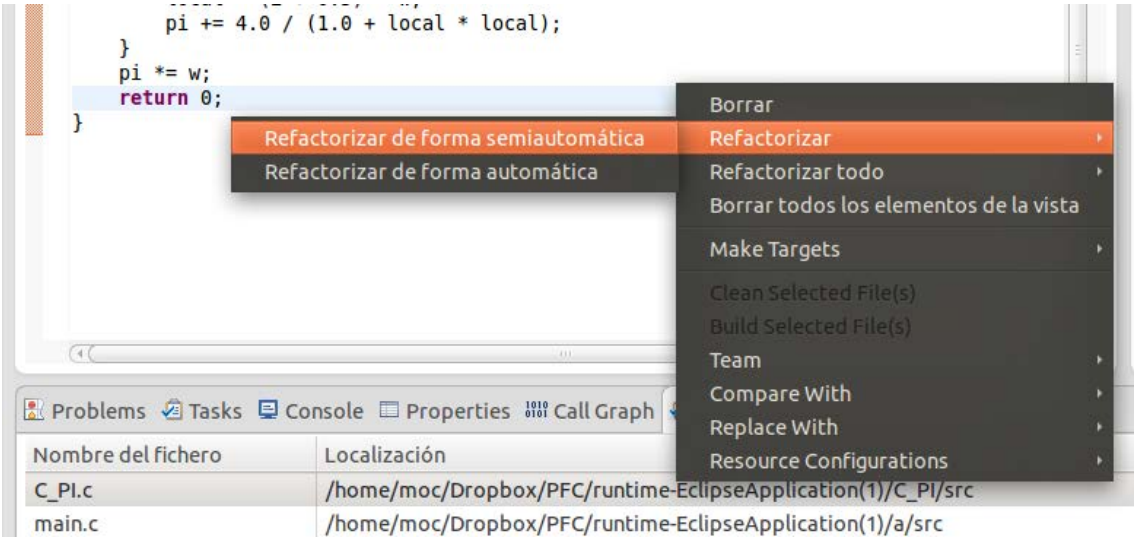


Figura 69 - Refactorizar ficheros de forma semiautomática

Para refactorizar un fichero a través del menú contextual del explorador de proyectos bastará con que el usuario seleccione los ficheros a refactorizar y pulse el botón derecho. En el menú contextual deberá dirigirse al menú *OpenMP* y dentro de éste a *Refactorizar*. Dentro del menú Refactorizar será necesario pinchar sobre la opción *Refactorizar de forma semiautomática*.

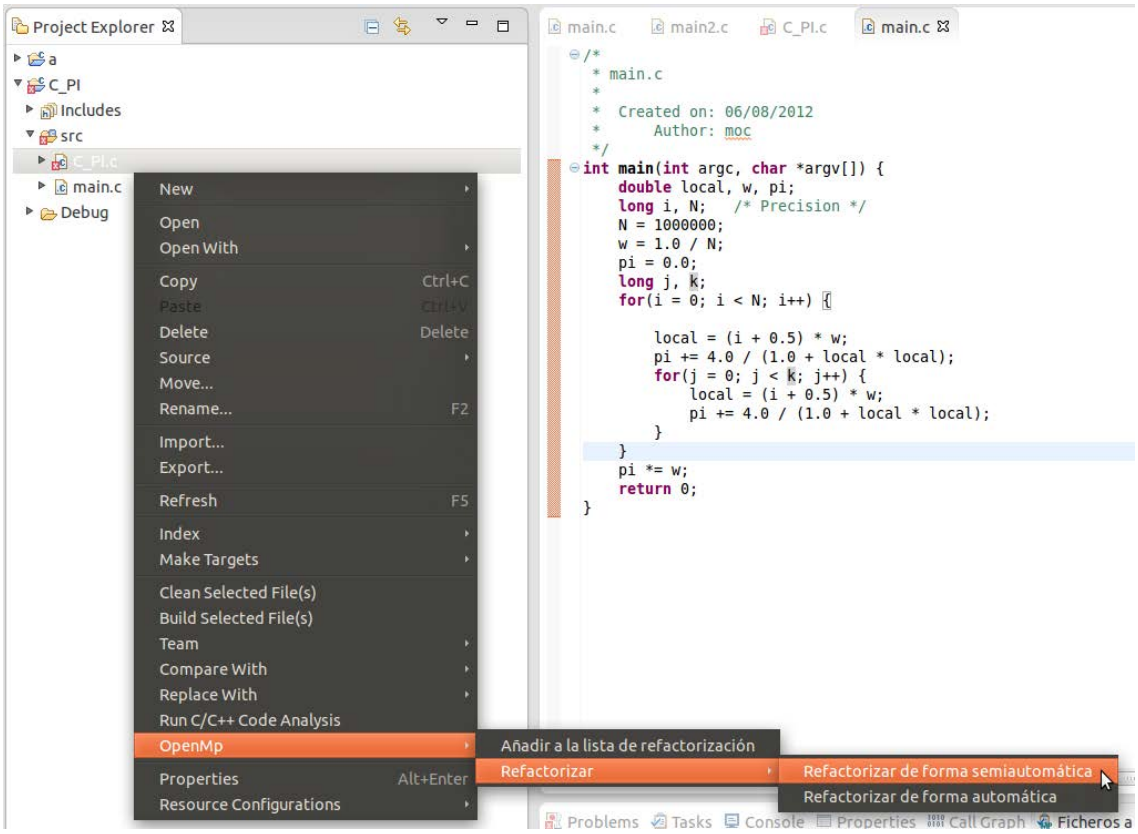


Figura 70 - Refactorizar de forma semiautomática directamente

Una vez realizada una de las tres acciones posibles, se abrirá una nueva ventana que mostrará un diálogo al usuario. Para proceder con la refactorización, el usuario deberá aceptar.

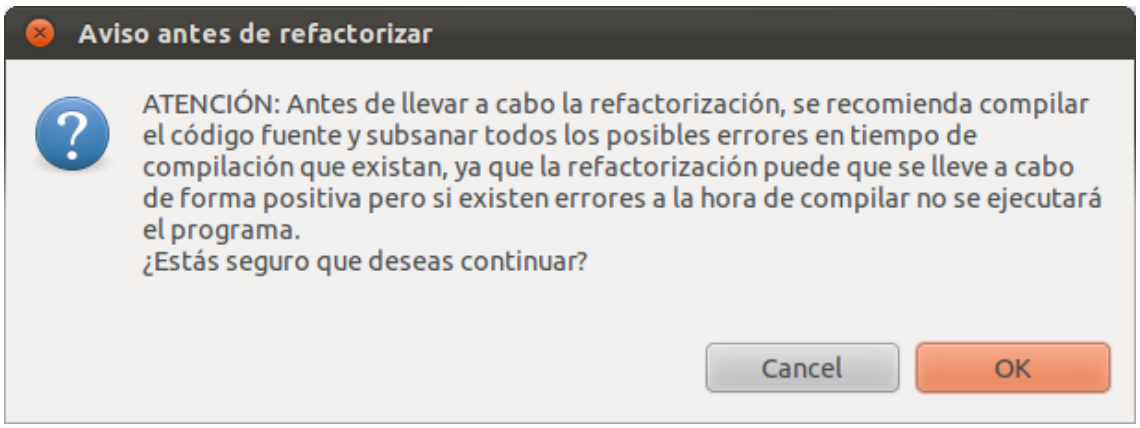


Figura 71 - Aviso antes de refactorizar

Cuando el usuario acepte el aviso se mostrará un asistente para llevar a cabo la refactorización de forma semiautomática.

La primera página del asistente mostrará un aviso al usuario de que para llevar a cabo una refactorización semiautomática es necesario cierto conocimiento de OpenMP y que si no se ha utilizado con anterioridad es recomendable utilizar la refactorización automática.

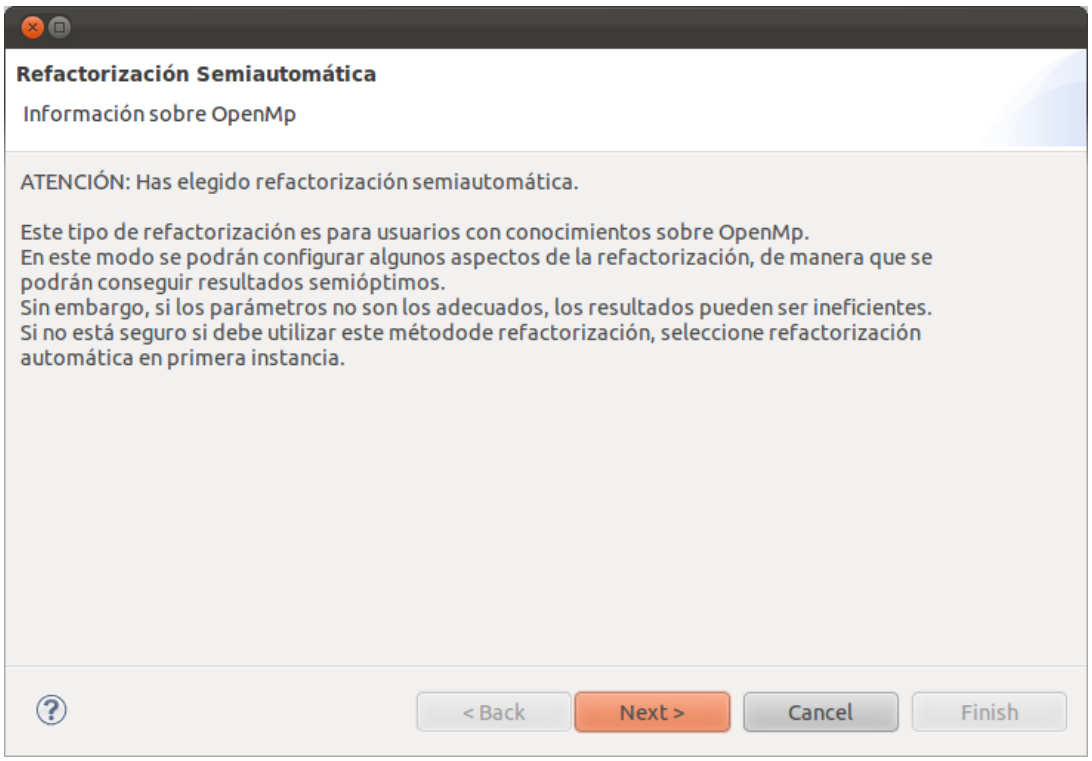


Figura 72 - Primera página del asistente

Una vez que el usuario pulse en siguiente, la segunda página del asistente mostrará el código sobre el cual se va a realizar la refactorización semiautomática.

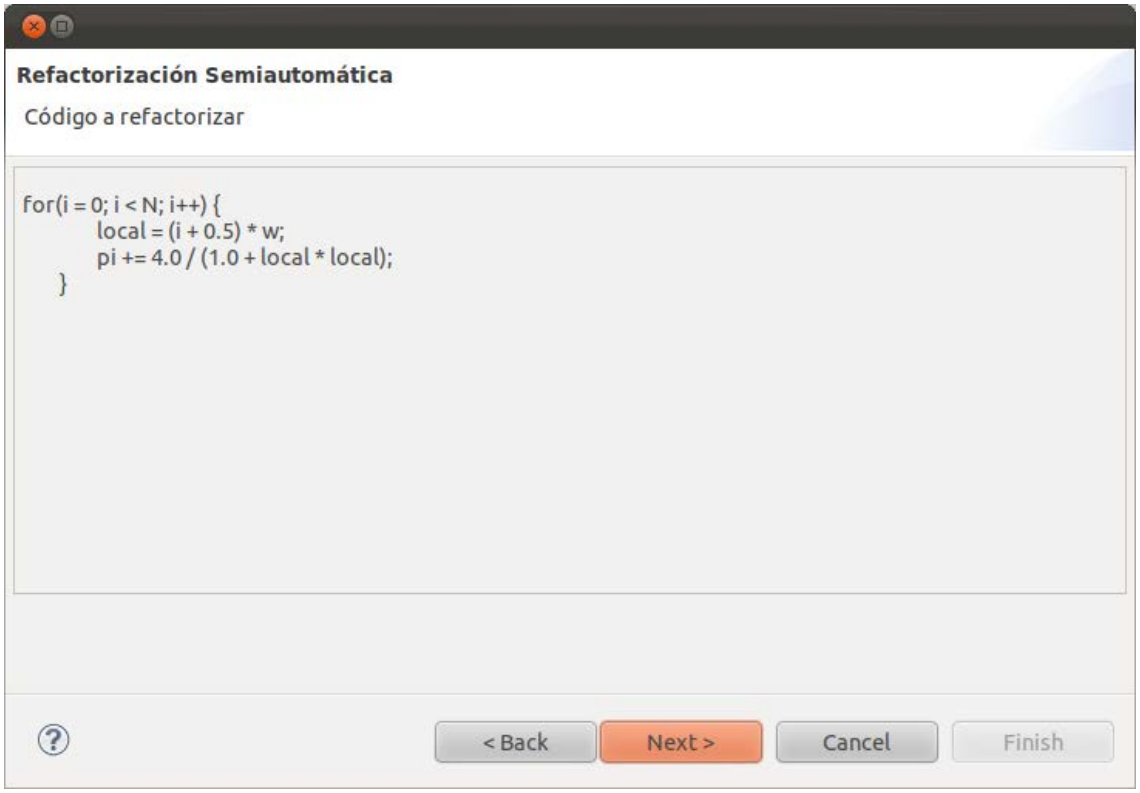


Figura 73 - Segunda página del asistente

Cuando el usuario pulse en siguiente se mostrarán las distintas opciones que puede escoger para llevar a cabo la refactorización del código de forma personalizada.

Figura 74 - Tercera página del asistente

Las opciones que puede personalizar el usuario son las siguientes:

Elegir automáticamente el número de hilos según el número de procesadores

En caso de que esta opción esté seleccionada el complemento usará llamadas a la API de OpenMP para establecer el número de hilos que ejecutarán la zona paralela.

En caso de que la opción no esté seleccionada, la opción *selecciona el número de hilos que se crearán* estará activa.

Seleccionar el número de hilos que se crearán

Con esta opción el usuario podrá escoger el número de hilos que ejecutarán la zona paralela. El mínimo número de hilos que permitirá seleccionar la opción será 2, ya que si se seleccionase un solo hilo no se ejecutaría en paralelo, sino de forma secuencial.

Seleccionar como se realizará el reparto de trabajo entre los hilos

Esta opción permitirá al usuario seleccionar el tipo de reparto de trabajo entre los hilos. Entre las opciones disponibles, estarán las siguientes:

- Reparto estático:
  - Al inicio de la zona paralela, se comprueba el número de iteraciones y se reparten de forma equitativa entre todos los hilos disponibles. En caso de que la división no sea exacta, el último hilo dispondrá de menos iteraciones que el resto.
- Reparto dinámico:
  - Según van llegando los hilos a la zona paralela, se van asignando iteraciones de forma dinámica. Por defecto, cada hilo ejecuta una sola iteración.
- Reparto guiado:
  - Al igual que en el reparto dinámico, las iteraciones se van asignando a los hilos según van llegando a la zona paralela. Se diferencia del anterior en que en este caso no se reparte una a una las iteraciones, sino que se reparten según la siguiente fórmula: número de iteraciones restantes/número de hilos menos 1.
  - Este reparto es útil en bucles en los que la carga de trabajo entre iteraciones no es equitativo.
- Reparto en tiempo de ejecución:
  - La política de reparto de trabajo se decide en tiempo de ejecución. Para decidir una u otra, se utilizará la variable de entorno OMP_SCHEDULE.

Seleccionar el tamaño de los fragmentos

Mediante esta opción, el usuario podrá seleccionar el tamaño de los bloques de iteraciones en los que se irá dividiendo el bucle según la política escogida anteriormente. Si se deja en uno, el tamaño de los fragmentos será escogido en tiempo de ejecución.

Todas las variables compartidas

Si se marca esta opción el complemento no buscará posibles variables privadas, sino que asignará una política por defecto de todas las variables compartidas. Esta opción es poco recomendable, ya que el acceso a variables compartidas es menos óptimo que a variables privadas.

Intentar realizar optimizaciones de reducción

Si se marca esta opción el complemento intentará buscar operaciones de reducción en la zona paralela. Esta opción es muy recomendable, ya que esta cláusula proporciona un acceso óptimo a variables de reducción.

En el momento en el que el usuario pulse en siguiente, se mostrará una ventana donde estará la directiva de OpenMP que se insertará en el documento junto con el código que se ejecutará en paralelo.

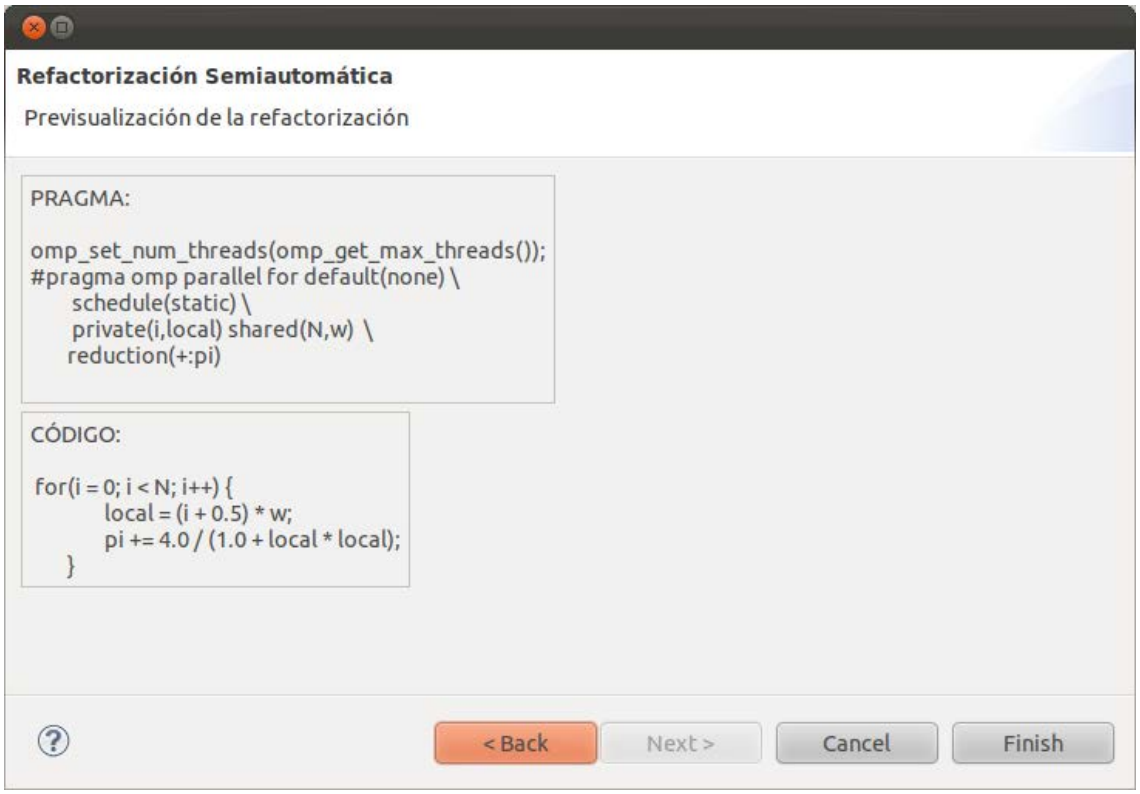


Figura 75 - Cuarta página del asistente

Por último, si el usuario pulsa sobre finalizar, el complemento escribirá la directiva creada en el fichero. Si existe otro bucle que se seleccionó para refactorizar, se volverá a abrir el asistente con el nuevo código.

REFACTORIZAR TODOS LOS FICHEROS DE LA VISTA DE FORMA AUTOMÁTICA

Para llevar a cabo la refactorización de todos los ficheros de la vista de forma automática será necesario que el usuario abra el menú contextual sobre la vista pulsando el botón derecho. Una vez abierto el menú contextual, el usuario deberá seleccionar la opción *Refactorizar todos los ficheros de forma automática* dentro del menú *Refactorizar todo*.

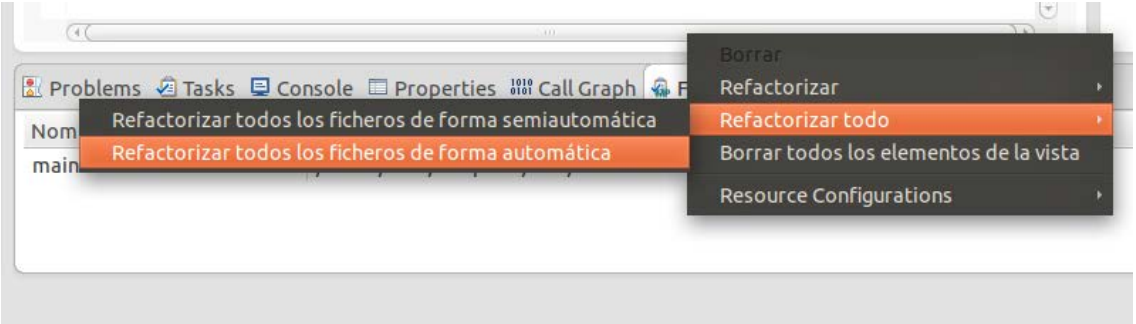


Figura 76 - Refactorizar todos los ficheros de la vista de forma automática

REFACTORIZAR TODOS LOS FICHEROS DE LA VISTA DE FORMA SEMIAUTOMÁTICA

Para llevar a cabo la refactorización de todos los ficheros de la vista de forma semiautomática será necesario que el usuario abra el menú contextual sobre la vista pulsando el botón derecho. Una vez abierto el menú contextual, el usuario deberá seleccionar la opción *Refactorizar todos los ficheros de forma semiautomática* dentro del menú *Refactorizar todo*.

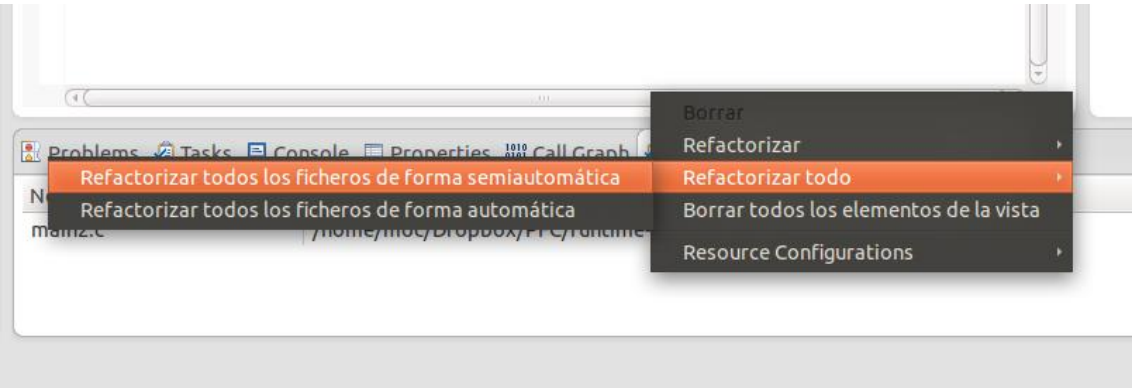


Figura 77 - Refactorizar todos los ficheros de la vista de forma semiautomática

ERRORES AL REFACTORIZAR

En caso de que la refactorización no se pueda realizar en un determinado bucle, se mostrará un aviso al usuario indicándole el motivo del error y la línea donde se ha encontrado. Si se encuentra algún error, la refactorización de ese bucle en concreto no se realizará.

Los avisos que se podrán mostrar al usuario son los siguientes:

Variable modificada

Se mostrará cuando dentro de un bucle se modifique algunas de las variables que marcan el número de iteraciones a realizar. En OpenMP el número de iteraciones debe de ser fijo en tiempo de ejecución para repartirlas entre los diferentes hilos y si no se cumple esa condición será imposible realizar la refactorización.

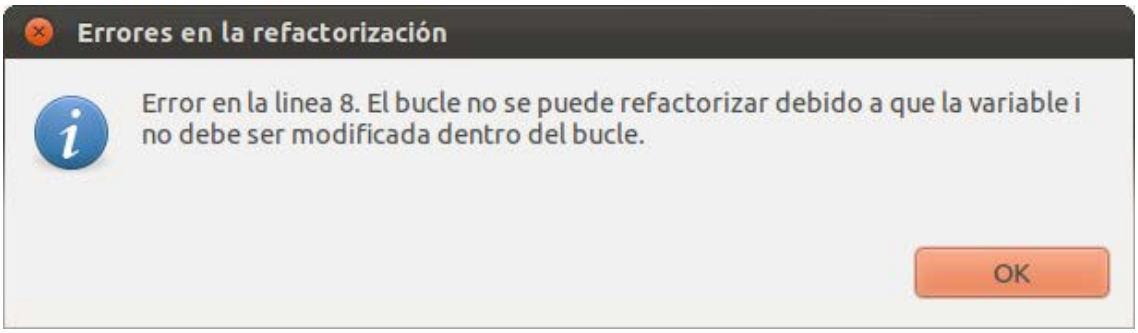


Figura 78 - Error de modificación de variables

Declaración de bucle incorrecta

Para que se pueda realizar la refactorización de un bucle for, su declaración tiene que seguir el siguiente patrón [14].

*for(**expresión-inicio; var op b; incremento-var**)* donde:

- var y b son variables enteras.
- expresión-inicio es una operación de asignación de la variable var.
- op es uno de los siguientes operadores:
  - >, <, <=, =>.
- incremento-var es una expresión donde se incrementa o se disminuye var con uno de los siguientes operadores:
  - ++, --, +=, -=.

En caso de que la declaración del bucle no cumpla estas condiciones se mostrará al usuario un aviso de que el bucle no está correctamente declarado.

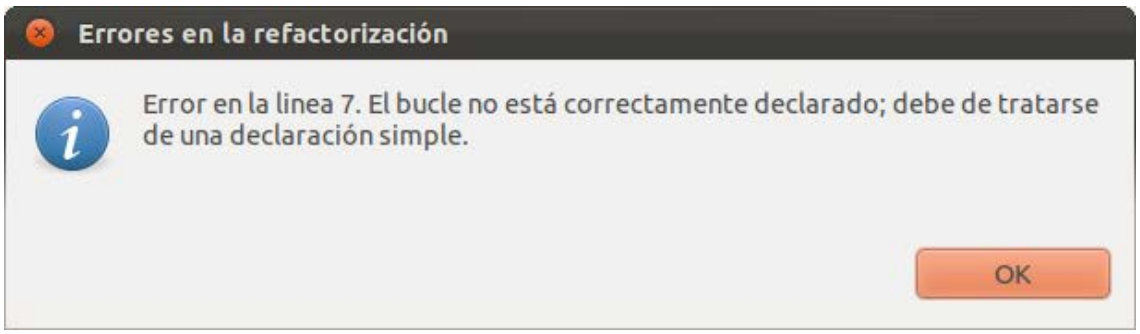


Figura 79 - Error de declaración de bucle

Falta de llaves

Para que se pueda llevar a cabo la refactorización de un bucle, es necesario que el contenido del bucle esté encerrado entre llaves. Además, los siguientes elementos internos al bucle deben de comenzar con una llave ({) y finalizar con otra (}):

- Bucles for, while y do-while.
- Sentencias if.

En caso de que alguna de estas sentencias no estén encerradas entre llaves, se mostrará el siguiente aviso:

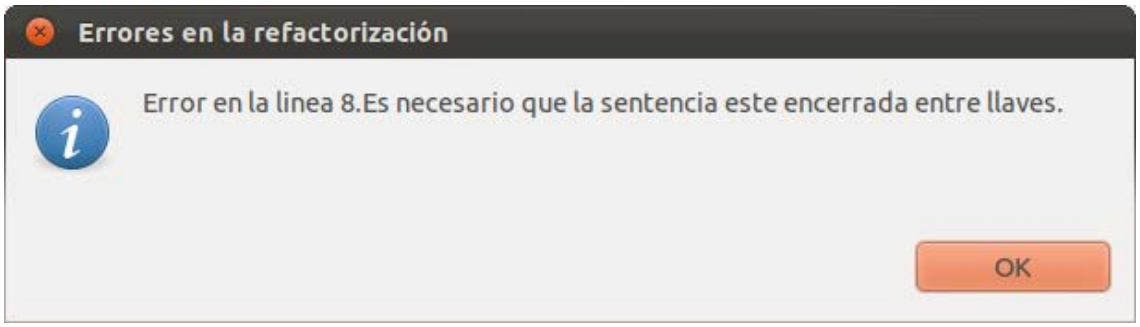


Figura 80 - Error por falta de llaves

COMPILACIÓN Y EJECUCIÓN DEL CÓDIGO

Para realizar la compilación del código refactorizado es necesario añadir una opción para que el compilador añada las librerías de OpenMP y se puedan ejecutar las zonas paralelas. A continuación se muestra una lista de compiladores compatibles con OpenMP y la opción asociada para compilar el código fuente con las librerías de OpenMP [32]:

- GNU.
  - –fopenmp.
- Oracle.
  - –xopenmp.
- Intel versión 10.1.
  - Windows: -Qopenmp.
  - Linux/Mac OSX: -openmp.
- Portland Group Compilers and tools.
  - –mp.
- Lahey/Fujitsu Fortran 95.
  - –openmp, --threadstack o –threadheap [28].
- PathScale.

- –mp [34].

- HP.
  - +Oopenmp [24].
- Cray.
  - Activado por defecto [41].
- NAG.
  - –openmp.
- OpenUH.
  - Activado por defecto.

8. ANEXO III: GUÍA DE CREACIÓN DE COMPLEMENTOS EN ECLIPSE

A continuación se describirá de forma breve como llevar a cabo el desarrollo de complementos en el IDE eclipse.

Eclipse ha facilitado desde el inicio el desarrollo de complementos por parte de los usuarios para ampliar la funcionalidad del IDE. Pensando en la comunidad, se integró desde la primera versión de Eclipse un entorno de desarrollo de complementos, denominado PDE, que facilitó enormemente la tarea de ampliar las funcionalidades de Eclipse.

PRERREQUISITOS

Para poder comenzar el desarrollo de un complemento, es necesario disponer del IDE Eclipse en el equipo. Existen múltiples versiones y variantes del IDE pero el entorno de desarrollo de complementos está integrado en todas ellas, por lo si no se dispone del IDE bastará con dirigirse a la página de Eclipse y descargar cualquiera de las versiones disponibles.

CREACIÓN DEL PROYECTO PDE

El desarrollo del complemento estará englobado dentro de un proyecto de tipo *plugin project*. Dentro del proyecto se encontrarán todos los elementos necesarios para poder empaquetar el complemento en un fichero y poder ejecutarlo en cualquier otro Eclipse en el que se instale.

Para crear el proyecto estando en el banco de trabajo principal de Eclipse, habrá que pulsar sobre el menú *File*, luego sobre *New* y por último sobre *Other*... o pulsar Control+N.

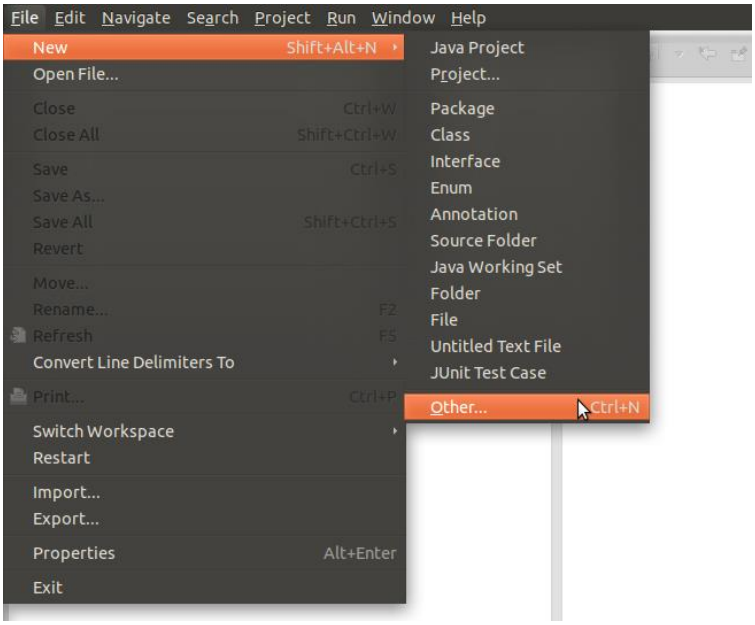


Figura 81 - Abrir el menú de nuevo elemento

En el nuevo menú que se abrirá, se seleccionará Plug-in Project para iniciar la creación del proyecto.

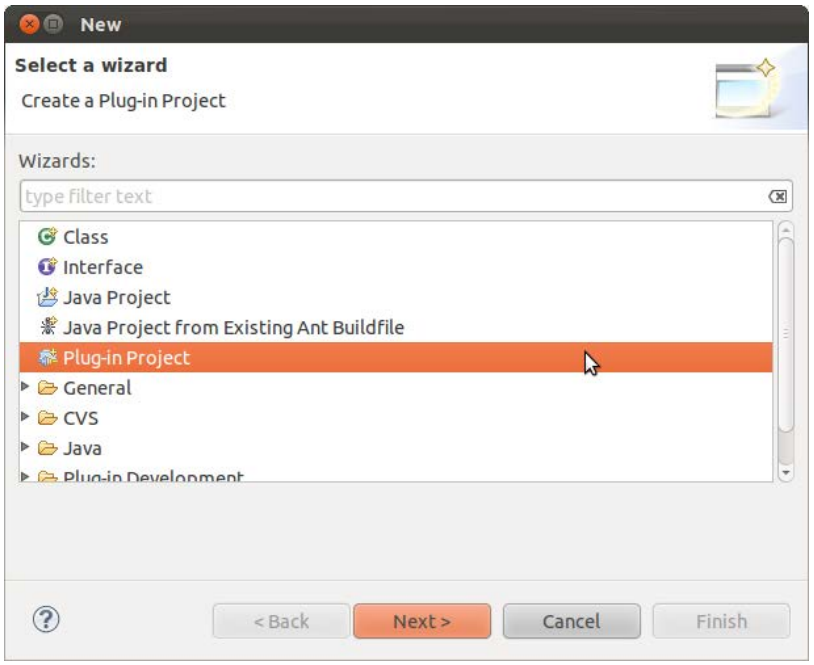


Figura 82 - Nuevo proyecto de desarrollo de complemento

Una vez que se pulse en siguiente se abrirá la pantalla principal de configuración del proyecto. En ella se deberá especificar el nombre del proyecto, su localización, las

carpetas donde se encuentran los ficheros fuente, donde se guardarán los ficheros compilados, la versión de Eclipse a partir de la cual se podrá ejecutar el complemento y si se quiere añadir el proyecto a un conjunto de trabajo. Para la guía, se dejará todos los valores por defecto y se le dará como nombre al proyecto pfg.desarrollo.complemento. La nomenclatura de los nombres de los complementos sigue una estructura jerárquica, al igual que los paquetes. De esta manera se pueden agrupar elementos con la misma funcionalidad. El nombre también servirá para crear la estructura de paquetes interna del proyecto.

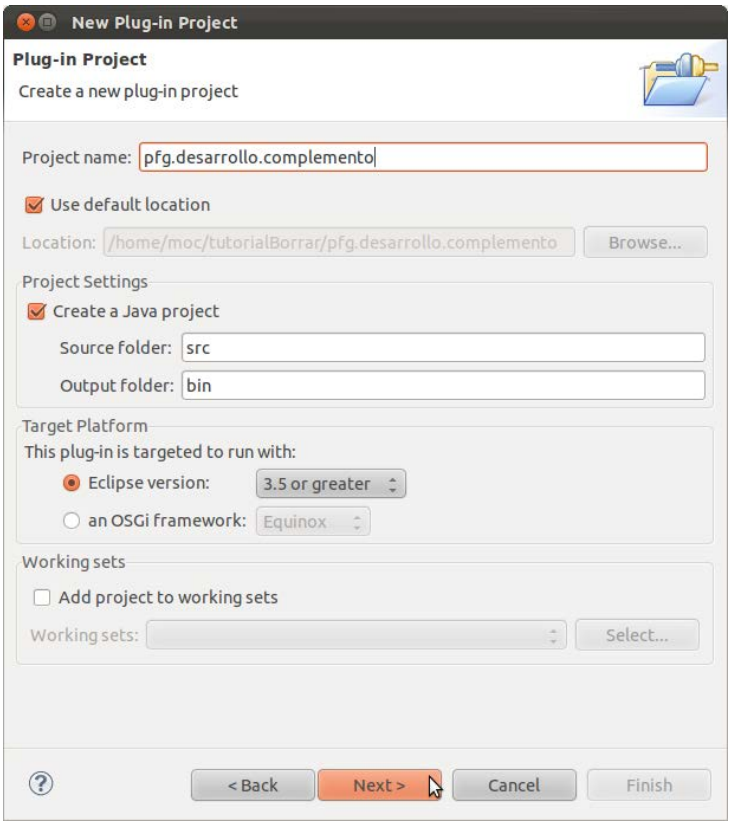


Figura 83 - Primera ventana de creación del complemento

En la siguiente pantalla se podrán modificar algunas propiedades del proyecto. El identificador será la manera de hacer referencia a nuestro complemento. Normalmente será el mismo que el nombre del proyecto, pero se le puede dar el nombre que se desee.

La versión tiene que seguir el siguiente patrón:

X.Y.Z.String donde:

- X es un valor entero que indica que se ha realizado un cambio mayor en el complemento, creándose incompatibilidades con antiguas versiones del IDE o con otros complementos.
- Y es un valor entero que indica que se han realizado cambios perceptibles por los usuarios en el complemento.
- Z es un valor entero que indica que se han realizado arreglos de diferentes tipos de errores en el complemento.
- String es una palabra alfanumérica, es opcional y hace referencia a una construcción del complemento específica.

El nombre servirá para referirse al complemento de una forma más simple. El apartado de vendedor servirá para escribir la empresa que está llevando a cabo el desarrollo del complemento.

El asistente creará de forma automática la clase activadora del complemento, la cual servirá para llevar un control sobre el ciclo de vida del mismo cuando se esté ejecutando.

En esta pantalla se dejará todo tal y como está.

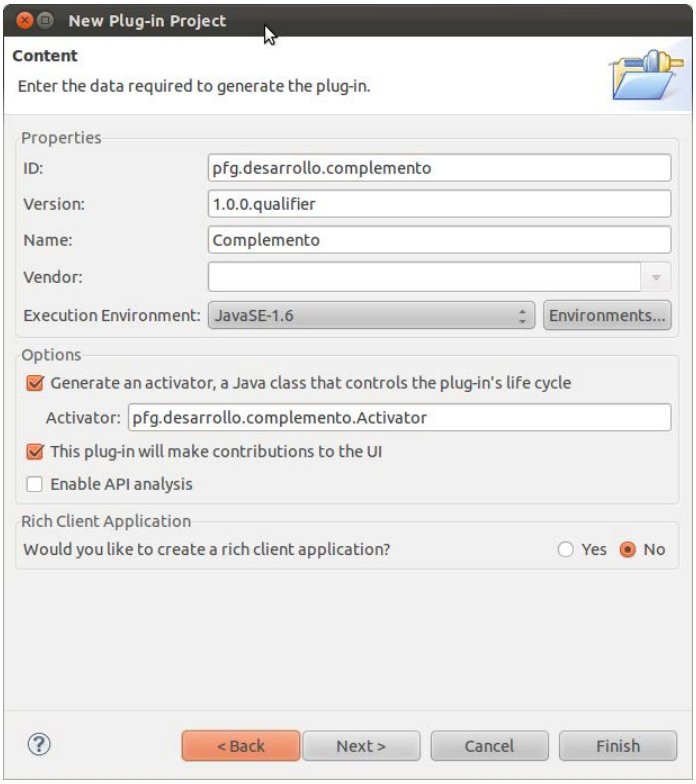


Figura 84 - Segunda ventana de creación del complemento

La siguiente ventana servirá si queremos usar alguna plantilla en el complemento. Como no es el caso, se pulsará sobre finalizar, terminando el asistente de creación y apareciendo el nuevo proyecto sobre el banco de trabajo.

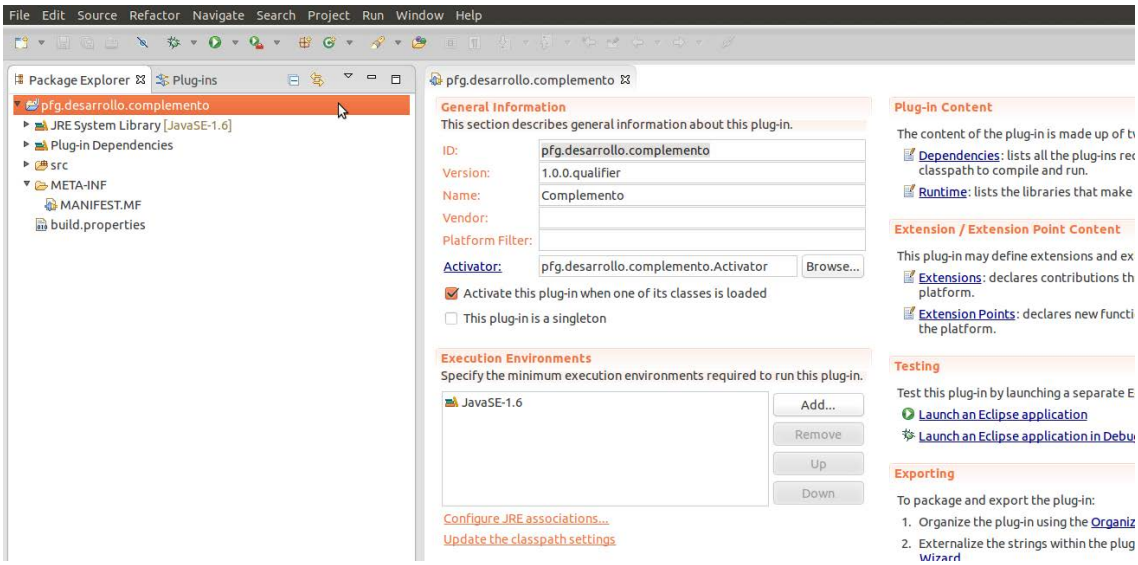


Figura 85 - Proyecto creado

Se abrirá automáticamente una ventana en el editor que será la que nos permitirá definir las dependencias de nuestro complemento, utilizar puntos de extensión, definir como se llevará a cabo la compilación del complemento, etc. Dicha ventana servirá para facilitar la edición de los ficheros MANIFEST.MF y build.properties, que no son más que ficheros XML que definirán en gran medida el comportamiento del complemento.

CREACIÓN DE UNA VISTA

Para crear una vista será necesario acceder al fichero MANIFEST.MF a través del Plug-in Manifest Editor. Normalmente bastará con pinchar dos veces sobre el fichero, pero en caso de que se haya modificado la forma de abrir dicho fichero bastará con pinchar con el botón derecho sobre él, ir al menú *Open With* y seleccionar la opción *Plug-in Manifest Editor.*

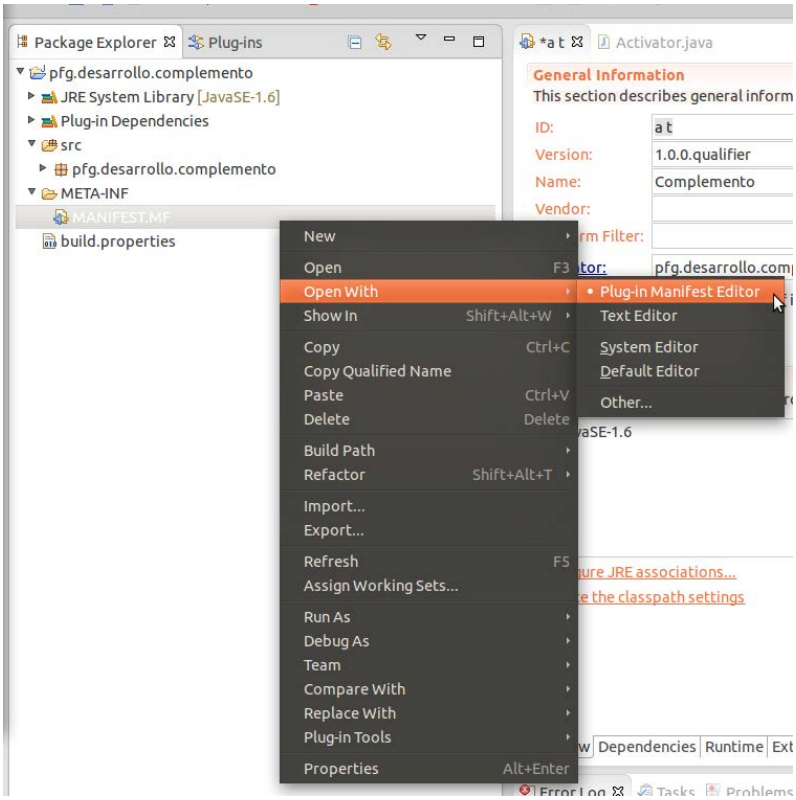


Figura 86 - Seleccionar abrir el fichero con el Plug-in Manifest Editor

Ahora será necesario dirigirse a la pestaña *Extensions* y añadir una nueva extensión de tipo org.eclipse.ui.views. Esta extensión nos permitirá crear nuevas vistas que añadan nuevas funcionalidades al IDE.

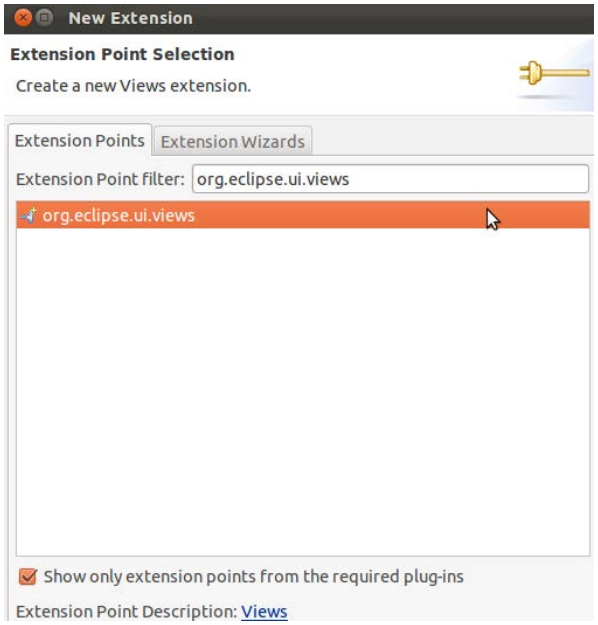


Figura 87 - Punto de extensión org.eclipse.ui.views

Lo primero que hay que hacer después de crear el punto de extensión es crear una categoría donde se encontrará la vista. Para ello, se pinchará con el botón derecho sobre el punto de extensión y se seleccionará la opción *New* y a continuación *category*.

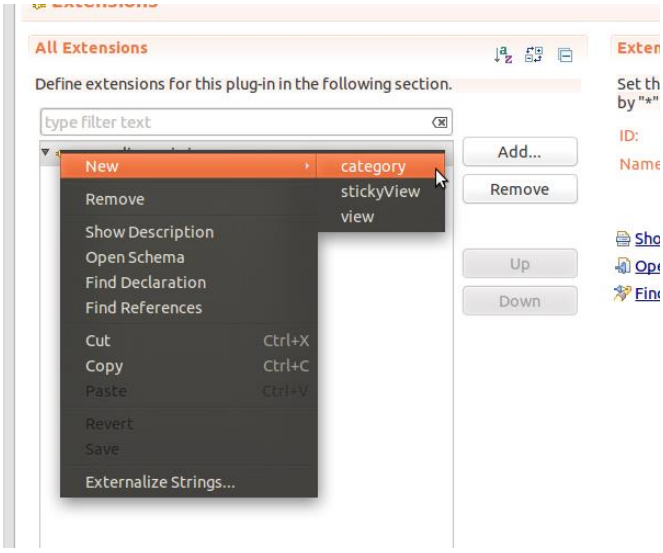


Figura 88 - Creación de una nueva categoría de vistas

Los atributos de la categoría que habrá que modificar son los siguientes:

- **id:** identificador de la vista para ser referenciada desde otros puntos del complemento.
    - Le daremos el valor "pfg.desarrollo.complemento.categoriaPrueba".
- **name:** nombre de la categoría que será visible en los menús.
    - Le daremos el valor "Categoría de Prueba".

Ahora ya se podrá añadir la vista. Para ello se pinchará con el botón derecho sobre el punto de extensión y se seleccionará la opción *New* y a continuación *view.*

Los atributos de la vista que habrá que modificar son los siguientes:

- **id:** identificador de la vista para referenciarla desde otros puntos del complemento.
    - Para nuestro caso "pfg.desarrollo.complemento.vistas.vista1".
- **name:** nombre que hará referencia a la vista en los diferentes menús. Debe de ser un nombre simple que haga referencia al contenido de la vista.
    - Para nuestro caso "Vista1".
- **class:** clase que implementará la vista.
    - Para nuestro caso "pfg.desarrollo.complemento.vistas.Vista1".
- **category:** categoría a la que pertenece la vista.
    - En este caso habrá que introducir la categoría creada con anterioridad "pfg.desarrollo.complemento.categoriaPrueba".

Para crear la clase que implementará la vista se pinchará sobre el atributo *class* una vez introducido el nombre.

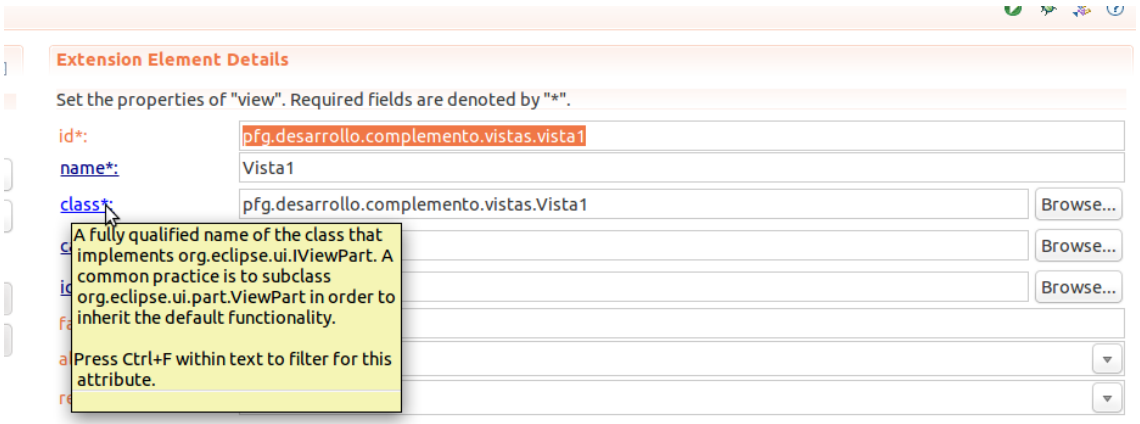


Figura 89 - Creación de la clase que implementará la vista

En la nueva pantalla que se abrirá se dejará todos los valores por defecto y se pulsará en *Finish*. A continuación se abrirá en el editor la clase que implementará la vista.

En primer lugar habrá que decidir qué tipo de estructura formará la vista. En este caso, se va a utilizar una tabla con una sola columna y tres elementos de ejemplo.

Para crear la tabla de la vista habrá que crear un objeto de la clase TableViewer en el método createPartControl, que será el método encargado de darle forma a la vista. También habrá que crear un objeto que sirva como proveedor de contenido. Normalmente dicho objeto se crea en una clase aparte y se instancia desde la vista; sin embargo, en este ejemplo sencillo, se va a crear una clase interna en la vista que sirva como proveedor.

El proveedor de contenido tendrá que implementar la interfaz IStructuredContentProvider, por lo que tendrá que definir los tres métodos que dispone dicha interfaz: dispose(), inputChanged() y getElements(). El único que nos interesa actualmente es getElements(), que nos proporcionará los elementos a mostrar en la vista.

Por último, habrá que definir que el proveedor de contenido de la vista es la clase interna creada y que los elementos serán proporcionados por la propia vista. El código de la clase queda así:

```
public class Vista1 extends ViewPart {

	private TableViewer tabla;

	class ViewProveedorContenido implements IStructuredContentProvider {

		@Override
		public Object[] getElements(Object inputElement) {
			return new String [] {"Primero", "Segundo", "Tercero" };
		}
	}

	@Override
	public void createPartControl(Composite parent) {
		tabla = new TableViewer(parent, SWT.MULTI | SWT.H_SCROLL |
SWT.V_SCROLL);
		tabla.setContentProvider(new ViewProveedorContenido());
		tabla.setInput(getViewSite());
	}
```

Figura 90 - Código de la vista

Para probar que la vista funciona seleccionaremos ejecutar en la barra de herramientas principal de Eclipse o pincharemos Control+F11.

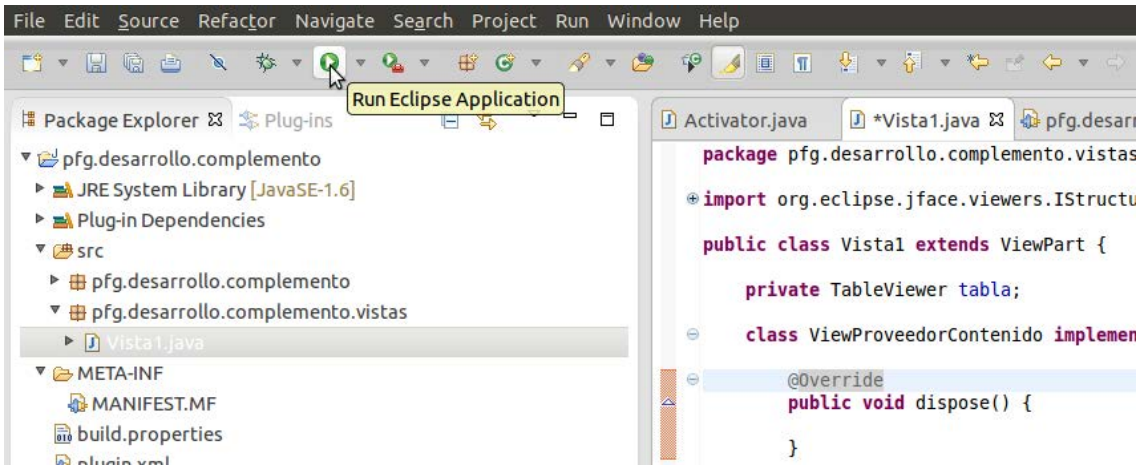


Figura 91 - Lanzar una prueba del complemento

Se abrirá un nuevo Eclipse. Para abrir la vista creada, habrá que pinchar en el menú *Window*, luego sobre *Show View* y por último sobre *Other*…. En el nuevo menú

que se abrirá habrá que ir a la carpeta llamada “Categoría de Prueba” y dentro pinchar sobre “Vista1”.

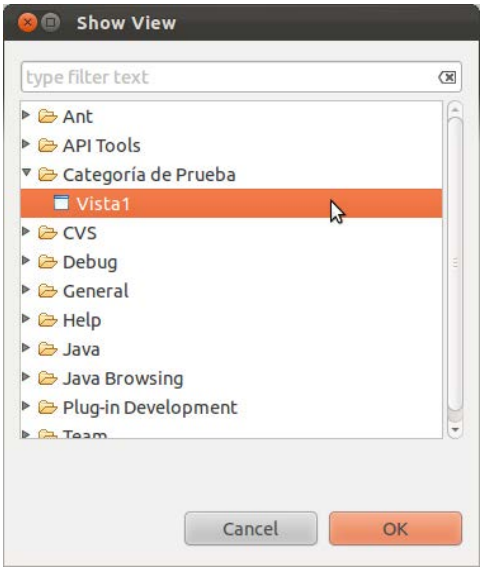


Figura 92 - Abrir la vista creada

Se abrirá la vista creada en la parte inferior del banco de trabajo de Eclipse y contendrá las tres filas creadas en el código.

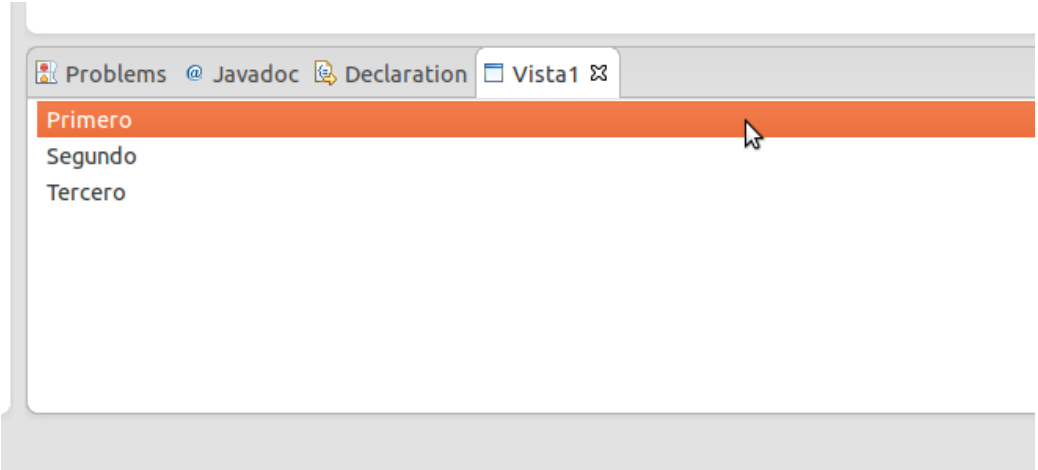


Figura 93 - La vista en el banco de trabajo de Eclipse

CREACIÓN DE UNA PERSPECTIVA

Al igual que en el caso anterior lo primero que habrá que hacer para crear una nueva perspectiva será abrir el fichero MANIFEST.MF. Una vez abierto será necesario ir a la pestaña *Extensions* y añadir una nueva extensión. Para el caso de una perspectiva, habrá que añadir el punto de extensión org.eclipse.ui.perspectives. Esta extensión permitirá definir los elementos por los que estará formado la perspectiva y el comportamiento de la misma.

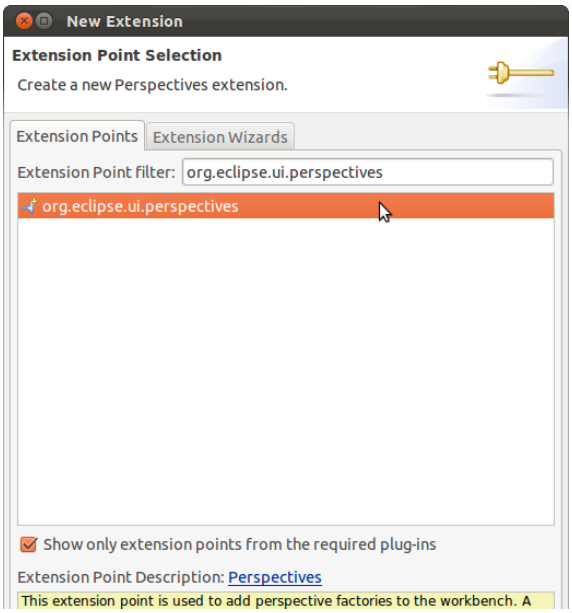


Figura 94 - Punto de extensión org.eclipse.ui.perspectives

En la nueva perspectiva creada hay que modificar los siguientes atributos:

- **id:** identificador de la perspectiva. Se usará para hacer referencia a la misma desde otras partes del complemento.
    - Le damos el valor pfg.desarrollo.complemento.perspectiva.
- **name:** nombre que hará referencia a la perspectiva en los diferentes menús. Debe de ser un nombre simple que haga referencia al contenido de la perspectiva.
    - En este caso será Perspectiva Ejemplo.
- **class:** clase que implementará dicha perspectiva.

- pfg.desarrollo.complemento.perspectivas.Perspectiva1, por ejemplo.

Para crear la clase que definirá el comportamiento de la perspectiva se pinchará sobre el atributo *class* una vez introducido el nombre completo.

En la nueva pantalla que se abrirá se dejará todo por defecto y se pulsará sobre *Finish*, abriéndose la nueva clase creada en el editor. Ahora será necesario definir de forma programática los elementos por los que estará formada la perspectiva.

En la perspectiva se va a añadir algunos accesos directos en los menús de creación de nuevos elementos y en los de mostrar vistas. Además, se añadirá por defecto la vista creada anteriormente a la perspectiva, junto con un explorador de proyectos y la vista de errores.

Para añadir dichos elementos habrá que modificar el método createInitialLayout(). En primer lugar, para añadir los accesos directos a los menús, habrá que utilizar el método addNewWizardShortcut() para el menú de crear nuevos elementos y addShowViewShortcut() para el menú de abrir vista.

A ambos métodos hay que pasarlos un atributo que indicará que elemento queremos añadir. La mayoría de ellos ya están definidos anteriormente y solo habrá que buscar en las clases de Eclipse. Para añadir un acceso directo a la vista creada, bastará con pasarle al método addShowViewShortcut el identificador de la misma.

El código es el siguiente:

```
layout.addNewWizardShortcut("org.eclipse.ui.wizards.new.folder");
layout.addNewWizardShortcut("org.eclipse.ui.wizards.new.file");

layout.addShowViewShortcut(IPageLayout.ID_BOOKMARKS);
layout.addShowViewShortcut(IPageLayout.ID_OUTLINE);
layout.addShowViewShortcut(IPageLayout.ID_PROP_SHEET);
layout.addShowViewShortcut(IPageLayout.ID_TASK_LIST);
    layout.addShowViewShortcut("pfg.desarrollo.complemento.vistas.vista1");
```

Figura 95 - Código para añadir elementos a la perspectiva

Para crear las vistas abiertas por defecto en la perspectiva habrá que crear un contenedor donde colocarlas. Para ello, se utilizará la clase IFolderLayout. Se creará uno que estará localizado a la izquierda para contener el explorador de proyectos y otro que estará en la parte inferior con la vista creada y la de errores.

Para situar los contenedores será necesario tener un elemento de referencia. Para ello, se obtendrá el editor, que siempre estará localizado en el centro y se colocarán a partir de él. El código es el siguiente:

```
String editorArea = layout.getEditorArea();

IFolderLayout left = layout.createFolder("left", IPageLayout.LEFT, (float)
0.18, editorArea);

left.addView(IPageLayout.ID_PROJECT_EXPLORER);

IFolderLayout down = layout.createFolder("down", IPageLayout.BOTTOM, (float)
0.80, editorArea);

down.addView(IPageLayout.ID_PROBLEM_VIEW);
down.addView("pfg.desarrollo.complemento.vistas.vista1");
```

Figura 96 - Contenedores y vistas abiertas por defecto

Ahora ya se podrá lanzar el complemento para comprobar que funciona. Una vez ejecutado pinchando en *Run* o pulsando Control+F11, habrá que dirigirse al menú *Window,* luego *Open Perspective* y por último *Other.* En el nuevo menú que se abrirá, habrá que seleccionar “Perspectiva Ejemplo” para abrir la perspectiva creada.

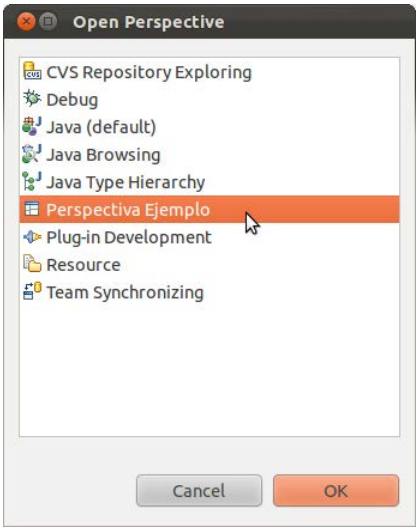


Figura 97 - Abrir la nueva perspectiva

El resultado es el siguiente:

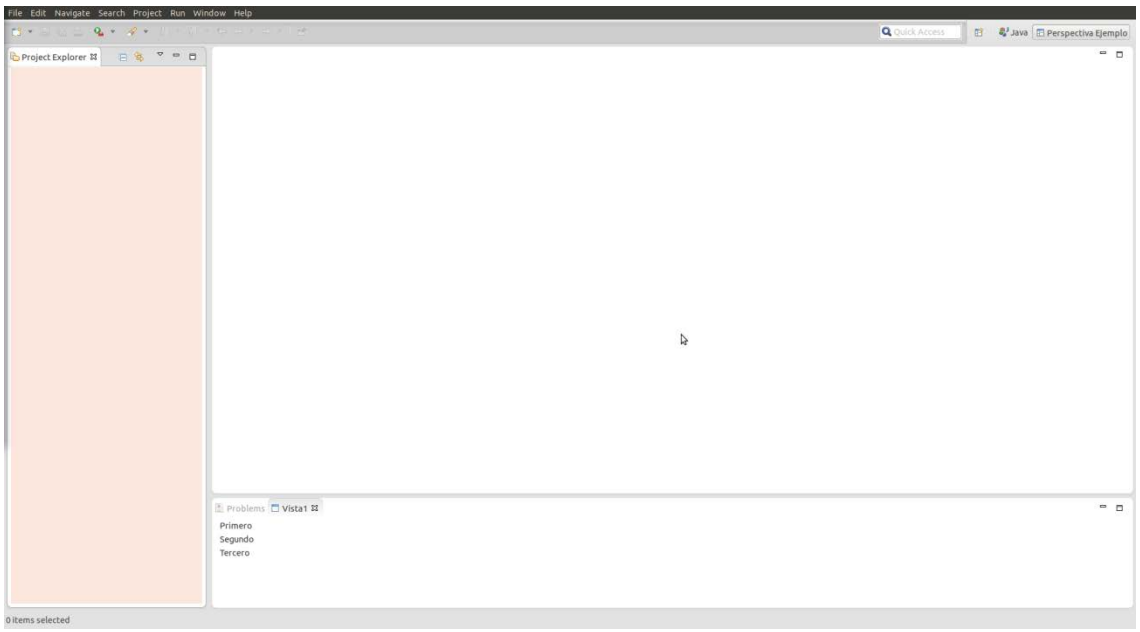


Figura 98 - La perspectiva una vez abierta

CREACIÓN DE UN COMANDO

Los comandos son las formas que tiene el usuario de interactuar con el complemento para realizar ciertas acciones. Se pueden utilizar tanto en las barras de herramientas disponibles en el banco de trabajo como en los menús contextuales. La

acción a ejecutar por cada comando estará controlada por una clase especial, denominada manejador.

Para crear un comando, lo primero será abrir el fichero MANIFEST.MF e ir a la pestaña de *Extensions*. Una vez allí, habrá que añadir una extensión de tipo org.eclipse.ui.commands.

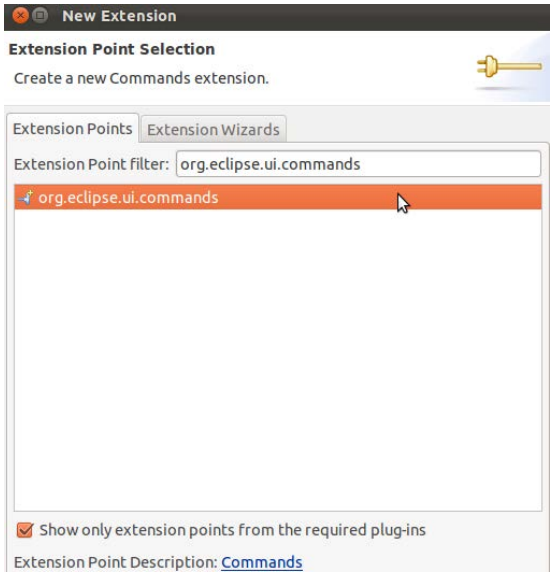


Figura 99 - Punto de extensión org.eclipse.ui.commands

Una vez añadido el punto de extensión, habrá que pulsar con el botón derecho sobre él y seleccionar *New* y a continuación *command.* Los atributos que habrá que modificar del nuevo comando serán los siguientes:

- **id:** el identificador del comando para ser referenciado desde otros puntos del complemento.
    - Le damos el valor "pfg.desarrollo.complemento.comandos.abrirDialogo".
- **name:** el nombre que se mostrará en la interfaz. Es necesario que sea un nombre sencillo y explicativo.
    - Le damos el nombre "Abrir Diálogo".

De momento, este comando no es accesible, ya que no está incluido en ningún menú. El siguiente paso será incluir el comando en el menú contextual general.

Para ello, es necesario utilizar el punto de extensión org.eclipse.ui.menus. Añadimos un punto de extensión de este tipo y dentro de él un menuContribution, pinchando con el botón derecho y seleccionando *New* y luego *menuContribution.*

Para definir el lugar donde aparecerá un determinado comando, hay que definir el atributo locationURI de los menuContribution. Dicho atributo siempre seguirá el siguiente patrón:

tipo:lugar?posicion=referencia donde:

- tipo índica el tipo de menú al que se quiere contribuir. Puede tomar los siguientes valores:
    - popup, toolbar o menu.
- lugar indica en que vista o en que perspectiva se quiere incluir la contribución a menú.
- posicion indica, dentro del menú completo, en que posición se colocará la contribución. Puede tomar los siguientes valores:
    - before, after o endof.
- referencia índica el elemento que se tomará como referencia para colocar la contribución en el menú. Se puede usar la palabra additions para indicar una referencia genérica.

Para este caso, como se quiere definir un comando en un menú contextual, habrá que usar el siguiente locationURI: "popup:org.eclipse.ui.popup.any?after=additions" para que aparezca el comando en cualquier menú contextual que sea abierto.

Ahora habrá que definir que comandos queremos que sean visibles en las contribuciones a menús que hemos creado. Para ello, se seleccionará uno de las contribuciones a menús, se pinchará con el botón derecho y se seleccionará *New* y luego *command.* Habrá que añadir en el atributo commandId el identificador del comando de abrir diálogo, que es "pfg.desarrollo.complemento.commandos.abrirDialogo". El resto de atributos se dejarán en blanco.

Los puntos de extensión definidos tendrán el siguiente aspecto:

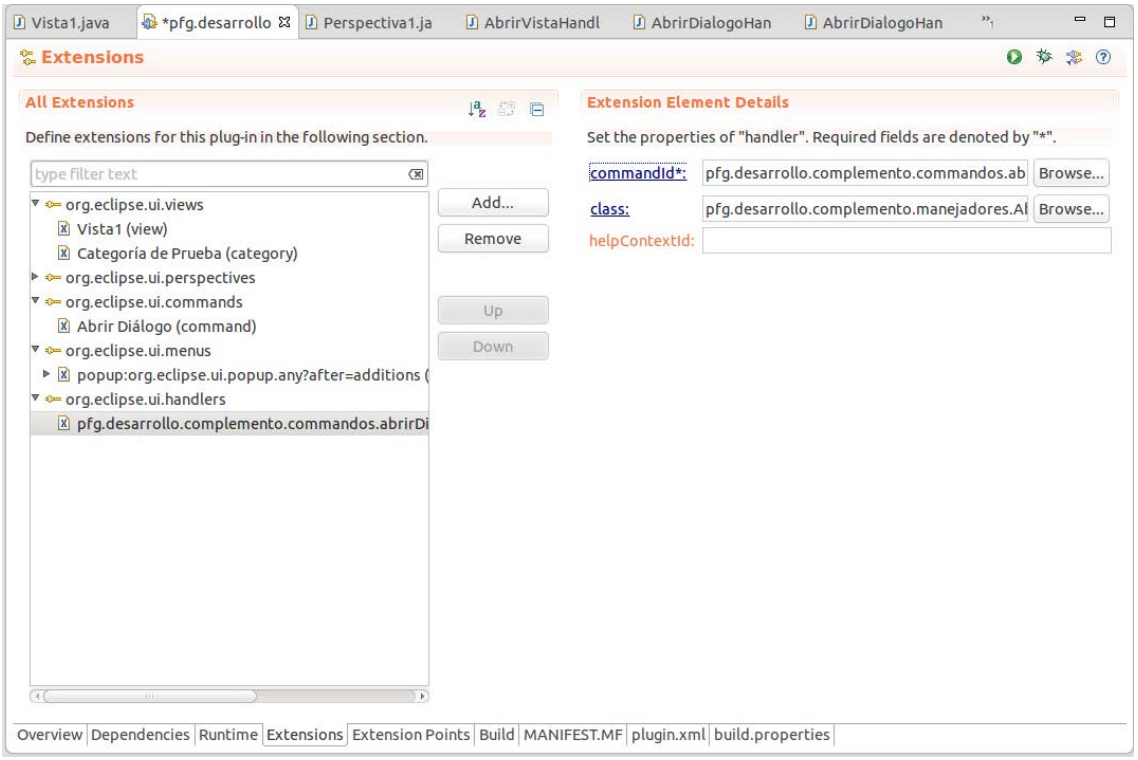


Figura 100 - Estado actual de los puntos de extensión

En último lugar habrá que crear el manejador del comando. Para crearlo, bastará con utilizar el punto de extensión org.eclipse.ui.handlers. Por lo tanto, estando en la pestaña *Extensions* habrá que pinchar en Add y añadir un punto de extensión de este tipo.

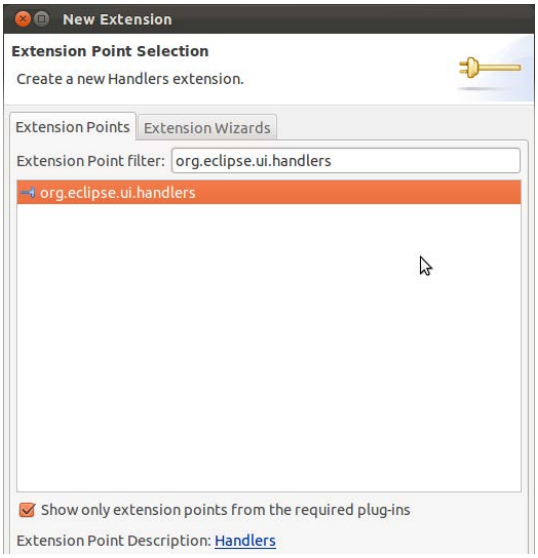


Figura 101 - Punto de extensión org.eclipse.ui.handlers

Para crear el manejador se pinchará con el botón derecho sobre el punto de extensión recién añadido y seleccionar *New* y luego handler. Los dos atributos que habrá que rellenar en el manejador creado son muy simples: commandId hace referencia al comando que va a usar este manejador y class a la clase que implementará el manejador. Introducimos el identificador del comando y como clase "pfg.desarrollo.complemento.manejadores.AbrirDialogoHandler".

A continuación habrá que pinchar sobre el atributo class y en la ventana que se abrirá se tendrá que seleccionar en el apartado de Interfaces la interfaz org.eclipse.core.commands.IHandler y se tendrá que eliminar, pinchando sobre el botón *Remove*. En el apartado de Superclass se pinchará en Browse y se buscará la clase org.eclipse.commands.AbstracHandler. Una vez encontrada, se confirmará, quedando de la siguiente manera la pantalla de creación de una nueva clase.

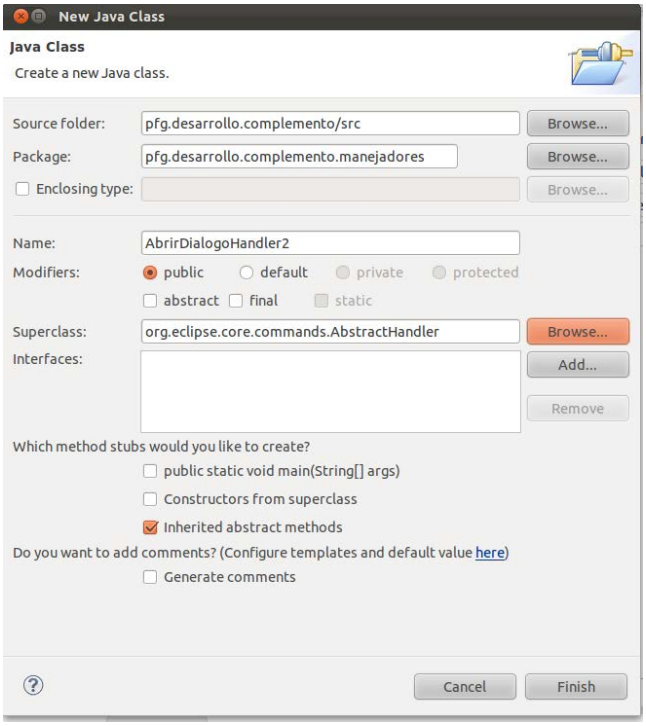


Figura 102 - Creación de los manejadores

En las clases creadas será necesario modificar el método execute(). El comando de "Abrir Diálogo" mostrará un mensaje de información al usuario. Para crear la lógica de este comando, habrá que utilizar la clase MessageDialog que servirá para crear todos los diálogos de alerta. Para que funcione, bastará con introducir el siguiente código en el método execute():

```
IWorkbenchWindow window =
HandlerUtil.getActiveWorkbenchWindowChecked(event);
            MessageDialog.openInformation(window.getShell(),
                        "Diálogo de información",
"¡El comando funciona!");
```

Figura 103 - Lógica del comando creado

Falta comprobar que el comando funciona. Para ello, lanzamos el complemento pinchando en *Run* o pulsando Control+F11. Una vez que se abra el banco de trabajo inicial de Eclipse, se pinchará con el botón derecho en cualquiera de las vistas disponibles y se pinchará sobre *Abrir Diálogo.* El resultado será que se abrirá un cuadro de diálogo de información con la información introducida anteriormente.

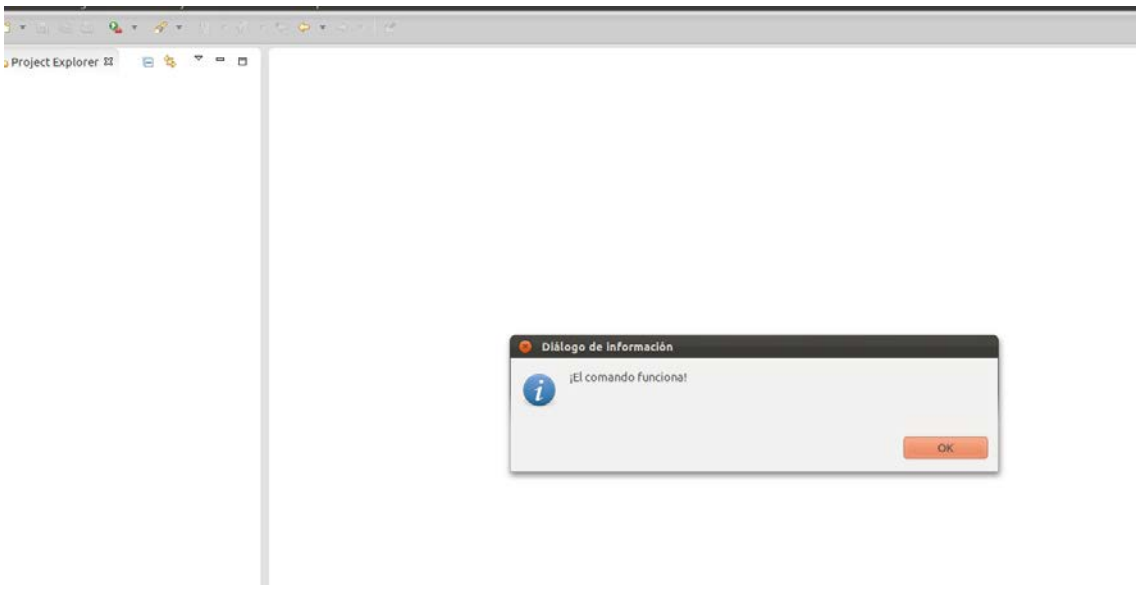


Figura 104 - Ventana de diálogo abierta

EXPORTACIÓN DEL COMPLEMENTO

Una vez que se ha desarrollado el complemento será necesario exportarlo de manera que pueda instalarse en cualquier IDE Eclipse que cumpla con la versión mínima marcada.

Para exportarlo habrá que dirigirse al menú *File* y pinchar sobre *Export.* En la nueva pantalla se tendrá que abrir la carpeta *Plug-in Development* y seleccionar *Deployable plug-ins and fragments.*

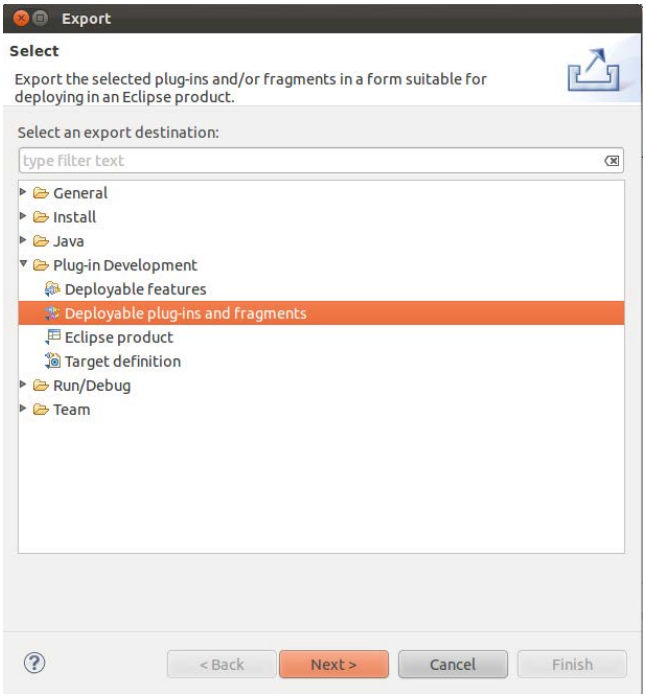


Figura 105 - Seleccionar tipo de exportación

Se pulsará en *Next* para proceder con la exportación. En la pantalla de exportación habrá que seleccionar el complemento desarrollado. En la pantalla inferior habrá que seleccionar como queremos exportar el complemento: si como un fichero comprimido o en un directorio. En nuestro caso, seleccionamos en un directorio y seleccionamos la ruta que queramos. Pulsamos sobre *Finish* y se creará un nuevo directorio denominado plugins en la ruta especificada. Para instalar el complemento en otro Eclipse bastará con copiar dicha carpeta a la carpeta raíz del IDE.

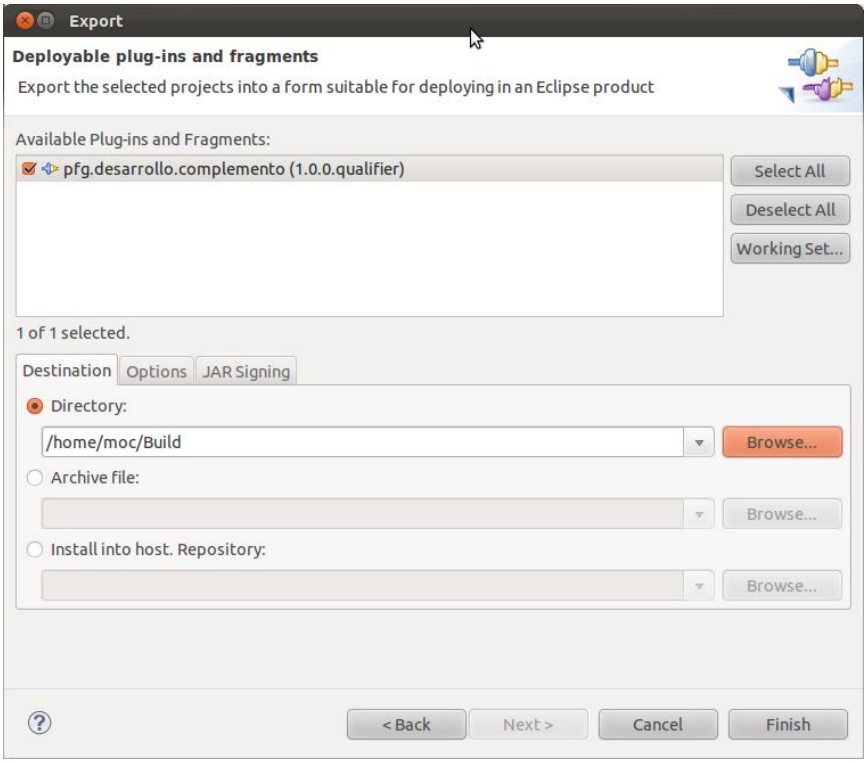


Figura 106 - Realizar la exportación del complemento

Bibliografía

[1] Amsden, J. & Irvine, A. 2003, 28.01.2003-última actualización, *Your First Plug-in*. Disponible: http://www.eclipse.org/articles/Article-Your%20First%20Plug-in/YourFirstPlugin.html [2012]

[2] OpenMP Architecture Review Board. 2012, 20.07.2012-última actualización, *OpenMP.org - The OpenMP API Specification for parallel programming*. Disponible: http://openmp.org/wp/ [2012]

[3] Intel Corporation, *General Questions about TBB*. Disponible: http://threadingbuildingblocks.org/wiki/index.php?title=General_Questions_about_TBB [2012]

[4] Oracle, *How to Write Doc Comments for the Javadoc Tool*. Disponible: http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html [2012]

[5] Intel Corporation, *Intel Cilk Plus - Intel Software Network*. Disponible: http://software.intel.com/en-us/articles/intel-cilk-plus/ [2012]

[6] RAE, *Real Academia Española*. Disponible: http://rae.es/rae.html [2012]

[7] Oracle, *Signing And Verifying JAR Files*. Disponible: http://docs.oracle.com/javase/tutorial/deployment/jar/signindex.html [2012]

[8] Eclipse Foundation, *SWT Widgets*. Disponible: http://www.eclipse.org/swt/widgets/ [2012]

[9] Barney, B. & Livermore, L. 2012, 12.07.2012-última actualización, *POSIX Threads Programming*. Disponible: https://computing.llnl.gov/tutorials/pthreads/ [2012]

[10] Bendersky, E. 2012, 16.06.2012-última actualización, *Pycparser - C Parser and AST generator written in Python*. Disponible: http://code.google.com/p/pycparser/ [2012]

[11] Braun, M. 2011, 23.12.2011-última actualización, *Cparser*. Disponible: http://sourceforge.net/projects/cparser/ [2012]

[12] Burkardt, J. 2012, 28.05.2012-última actualización, *C Source Codes*. Disponible: http://people.sc.fsu.edu/~jburkardt/c_src/c_src.html [2012]

[13] Carretero, J., García, F., de Miguel, P. & Pérez, F. 2007, *Sistemas Operativos: una visión aplicada.* 2ª edn, McGraw-Hill.

[14] Chapman, B., Jost, G. & van der Pas, R. 2008, *Using OpenMP: Portable Shared Memory Programming,* 1ª edn, MIT Press, Massachussets, Estados Unidos.

[15] Clayberg, E. & Rubel, D. 2008, *Eclipse Plug-ins,* 3ª edn, Addison-Wesley Professional, Estados Unidos.

[16] de Sandre, F. & Reyes, R. 2009, 17.07.2009-última actualización*, OmpSCR: OpenMP Source Code Repository*. Disponible: http://sourceforge.net/projects/ompscr/ [2012]

[17] Domig, M. & Webster, P. 2010, 15.08.2010-última actualización*, Eclipsepedia - Command Core Expressions*. Disponible: http://wiki.eclipse.org/Command_Core_Expressions [2012]

[18] Drake, J.M. 2008. *Programación concurrente. Sincronización basada en memoria compartida: semáforos.* Disponible: http://www.ctr.unican.es/asignaturas/procodis_3_II/Doc/Procodis_2_03.pdf [2012]

[19] Flatt, A. & Maison, M. 2011, 16.08.2011-última actualización*, Best practices for developing Eclipse plugins*. Disponible: http://www.ibm.com/developerworks/opensource/tutorials/os-eclipse-plugin-guide/ [2012]

[20] Franco, A. 2000, 01.01.2000-última actualización*, La Máquina Virtual Java*. Disponible: http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/introduccion/virtual.htm [2012]

[21] Gerber, R. 2008, 20.10.2008-última actualización*, Getting Started with OpenMP**. Disponible: http://software.intel.com/en-us/articles/getting-started-with-openmp/ [2012]

[22] Gonzalez, F.J. *Concurrencia: Exclusión mutua y sincronización*. Disponible: http://www.infor.uva.es/~fjgonzalez/apuntes/Tema6.pdf [2012]

[23] González, R. 2004, 10.04.2004-última actualización*, Una no tan breve historia de Java*. Disponible: http://mundogeek.net/archivos/2004/10/04/una-no-tan-breve-historia-de-java/ [2012]

[24] Hewlet-Packard Development Company 2010*, HP aC++ Online Programmer's Guide*. Disponible: http://h21007.www2.hp.com/portal/download/files/unprot/aCxx/Online_Help/options.htm#opt+Onoopenmp [2012]

[25] Anónimo 2010. *[C]Calcular pi*, http://foro.elhacker.net/programacion_cc/ [2012]

[26] Kabanov, J. 2010, 09.11.2010-última actualización, *The 2011 Turnaround, Container and Tools Pre-Report Snapshot*. Disponible: http://zeroturnaround.com/labs/the-2011-turnaround-container-and-tools-pre-report-snapshot/ [2012]

[27] Konigsberg, R. 2008, 23.06.2008-última actualización, *Using Property Testers in the Eclipse Command Framework*. Disponible: http://konigsberg.blogspot.com.es/2008/06/screencast-using-property-testers-in.html [2012]

[28] Lahey Computer Systems 2003. *Lahey/Fujitsu Fortran 95 User's Guide Linux Edition*. Disponible: http://www.compunity.org/resources/compilers/fujitsu/LaheyCompiler.pdf [2012]

[29] Maturana, J. 2011, 14.11.2011-última actualización, *OpenACC, nuevo estándar para computación paralela*. Disponible: http://www.muycomputer.com/2011/11/14/openacc-nuevo-estandar-para-computacion-paralela [2012]

[30] Melhem, W. & Glozic, D. 2003, 08.09.2003-última actualización, *PDE Does Plug-ins*. Disponible: http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html [2012]

[31] NVIDIA Corporation. *¿Qué es CUDA?*. Disponible: http://www.nvidia.es/object/what_is_cuda_new_es.html [2012]

[32] OpenMP Architecture Review Board 2012, 17.05.2012-última actualización, *OpenMP Compilers*. Disponible: http://openmp.org/wp/openmp-compilers/ [2012]

[33] Paez, F.E. 2009, 22.05.2009-última actualización, *Diseño UML: Diagrama de clases*. Disponible: http://egdamar877.blogspot.com.es/2009/05/expocicion.html [2012]

[34] PathScale LLC 2008. *PathScale Compiler Suite User Guide*. Disponible: http://www.pathscale.com/docs/UserGuide.pdf [2012]

[35] Pita, M.Á. 2007, 08.11.2007-última actualización, *Sincronización entre procesos*. Disponible: http://www.monografias.com/trabajos51/sincro-comunicacion/sincro-comunicacion.shtml [2012]

[36] Reinders, J. 2010, 04.05.2010-última actualización, *TBB 3.0: New (today) Version of Intel Threading Building Blocks*. Disponible: http://software.intel.com/en-us/blogs/2010/05/04/tbb-30-new-today-version-of-intel-threading-building-blocks/ [2012]

[37] Scarpino, M. 2006, 12.09-última actualización, *Building a CDT-based editor*. Disponible: http://www.ibm.com/developerworks/opensource/library/os-ecl-cdt1/index.html?S_TACT=105AGX44&S_CMP=EDU [2012]

[38] Springgay, D. 2001, 27.08.2001-última actualización, *Using Perspectives in the Eclipse UI*. Disponible: http://www.eclipse.org/articles/using-perspectives/PerspectiveArticle.html [2012]

[39] Stokes, J. 2000, 22.03.2000-última actualización, *SIMD Architectures*. Disponible: http://arstechnica.com/features/2000/03/simd/3/ [2012]

[40] Storkel, S. 2002, 12.11.2002-última actualización, *An Introduction to the Eclipse IDE*. Disponible: http://onjava.com/pub/a/onjava/2002/12/11/eclipse.html [2012]

[41] Swiss National Supercomputing Centre 2012. *Cray Compiler*. Disponible: http://user.cscs.ch/software_and_programming_environment/compilers_and_programming/rosa_cray_xt5/cray_compiler/index.html [2012]

[42] The Eclipse Foundation 2012. *About the Eclipse Foundation*. Disponible: http://www.eclipse.org/org/ [2012]

[43] Universidad Carlos III de Madrid 2012, 07.03.2012-última actualización, *Citas y referencias bibliográficas*. Disponible: http://portal.uc3m.es/portal/page/portal/biblioteca/aprende_usar/autoformacion/citas [2012]

[44] Universidad Carlos III de Madrid 2012, 16.07.2012-última actualización, *Mantenerse al día y compartir información*. Disponible: http://www.uc3m.es/portal/page/portal/biblioteca/aprende_usar/compartir [2012]

[45] Vogel, L. 2012, 28.01.2012-última actualización, *Creating Eclipse Wizards - Tutorial*. Disponible: http://www.vogella.com/articles/EclipseWizards/article.html [2012]

[46] Vogel, L. 2012, 22.02.2012-última actualización, *Eclipse Commands Tutorial*. Disponible: http://www.vogella.com/articles/EclipseCommands/article.html [2012]

[47] Vogel, L. 2012, 01.08.2012-última actualización, *Eclipse Source Code - Tutorial*. Disponible: http://www.vogella.com/articles/EclipseCodeAccess/article.html [2012]

[48] Vogel, L. 2012, 20.06.2012-última actualización, *Extending Eclipse - Plug-in Development Tutorial*. Disponible: http://www.vogella.com/articles/EclipsePlugIn/article.html [2012]

[49] Vogel, L. 2012, 01.07.2012-última actualización, *Python Development with PyDev and Eclipse - Tutorial*. Disponible: http://www.vogella.com/articles/Python/article.html [2012]

[50] Vogel, L., Ralfebert, R. & Roberts, D. 2012, 18.07.2012-última actualización, *Rich Client Platform - Eclipsepedia*. Disponible: http://wiki.eclipse.org/Rich_Client_Platform [2012]

[51] Voss, M., Kukanov, A., Robison, A., Poulsen, D., Jha, S. & Gromova, V. 2009, 27.07.2009-última actualización, *Intel Threading Building Blocks, OpenMP or native threads?*. Disponible: http://software.intel.com/en-us/articles/intel-threading-building-blocks-openmp-or-native-threads/ [2012]

[52] Williams, D., Keller, M., Arthorne, J., Suen, R. & Eicher, T. 2012, 15.03.2012-última actualización, *Eclipsepedia - Version Numbering*. Disponible: http://wiki.eclipse.org/Version_Numbering [2012]

[53] Eclipse Foundation. 2012. *Eclipse CDT.* Disponible: http://www.eclipse.org/cdt/ [2012]

[54] Mattson, T.G., Sanders, B.A. & Massingill, B.L. 2004, Patterns for Parallel Programming, 1ª edn, Addison-Wesley Professional.

[55] ONTSi. 2011. *Penetración de ordenador en hogares.* Disponible: http://www.ontsi.red.es/ontsi/es/indicador/penetraci%C3%B3n-de-ordenador-en-hogares [2012]

[56] PassMark Software. 2012. PassMark CPU Benchmarks. Disponible: http://www.cpubenchmark.net/common_cpus.html [2012]

[57] Barney, B. & Livermore, Lawrence. 2012, 16.07.2012-última actualización, *Introduction to Parallel Computing*. Disponible: https://computing.llnl.gov/tutorials/parallel_comp/ [2012]

[58] OpenMP Architecture Review Board. 2012, 05.2012-última actualización, *OpenMP Application Program Interface.* Disponible: http://www.openmp.org/mp-documents/spec30.pdf [2012]

[59] Anónimo. 2010, 19.07.2010-última actualización, *OpenCL.* Disponible: http://opencl.codeplex.com/wikipage?title=OpenCL%20Tutorials%20-%201 [2012]

[60] Vandenbout, Dave. 2008, 21.05.2008-última actualización, *My first CUDA program!*. Disponible: http://llpanorama.wordpress.com/2008/05/21/my-first-cuda-program/ [2012]

[61] Geva, Robert. 2011, 22.08.2011-última actualización, *Language of the Month: Intel's Cilk Plus*. Disponible: http://www.drdobbs.com/article/print?articleId=231400279&siteSectionName= [2012]

[62] Intel Corporation. 2007, *Intel Threading Building Blocks Tutorial*. Disponible: http://cache-www.intel.com/cd/00/00/30/11/301132_301132.pdf [2012]

[63] NVIDIA Corporation. *An OpenACC Example*. Disponible: http://developer.nvidia.com/cuda/openacc-example-part-1 [2012]

[64] Munshi, Aaftab. 2008, *OpenCL*. Disponible: http://s08.idav.ucdavis.edu/munshi-opencl.pdf [2012]

[65] Gliwa, Peter. 2003, 01.08.2003-última actualización, *CParser – A Simple File Parser*. Disponible: http://www.codeguru.com/cpp/cpp/cpp_mfc/parsing/article.php/c4069/CParsermdashA-Simple-File-Parser.htm [2012]

[66] Anónimo. *clang – a C language family frontend for LLVM*. Disponible: http://clang.llvm.org/index.html [2012]

[67] Anónimo. 2012, 28.05.2012-última actualización, *Interfaz de Paso de Mensajes*. Disponible: http://es.wikipedia.org/wiki/Interfaz_de_Paso_de_Mensajes [2012]

[68] Labarta, Jesus. 2009, *Hybrid/Heterogeneous Programming With StarSs*. Disponible: https://www.bscmsrc.eu/media/events/barcelona-multicore-workshop-2010/jesus-labarta-abstract [2012]

[69] Wachsmann, Alf. 2006, 22.03.2006-última modificación. *Mixing MPI and OpenMP Tutorial*. Disponible: http://www.slac.stanford.edu/comp/unix/farm/mpi_and_openmp.html [2012]

[70] Mueller, Oliver. 2012, 09.08.2012-última modificación. *On-The-Fly C++*. Disponible: http://blog.coldflake.com/posts/2012-08-09-On-the-fly-C%2B%2B.html [2012]

[71] Lattner, Chris. 2011, 19.12.2011-última modificación. *NVIDIA CUDA 4.1 Compiler Now Built on LLVM*. Disponible: http://blog.llvm.org/2011/12/nvidia-cuda-41-compiler-now-built-on.html [2012]

[72] Anónimo. 2012, 23.08.2012-última modificación. *Clang*. Disponible: http://en.wikipedia.org/wiki/Clang [2012]

[73] 1999, 14.12.1999-última modificacón. *Ley de Protección de datos*. Disponible: http://www.boe.es/boe/dias/1999/12/14/pdfs/A43088-43099.pdf [2012]

[74] Anónimo. 2008, 7.04.2008-última modificación. *Salario medio de un programador*. Disponible: http://www.tufuncion.com/trabajo-programador [2012]