Trabajo de Fin de Grado

Grado en Ingeniería Informática

Integrating COMPSs and OmpSs Programming Models to support distributed heterogeneous computing environments

En colaboración con Barcelona Supercomputing Center (BSC)

27 de junio de 2019

Autor: Marc Domínguez de la Rocha

Director: Rosa M. Badia

Co-director: Jorge Ejarque

Convocatoria: Primavera 2019

Facultat d'Informàtica de Barcelona (FIB)

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Resumen

Hoy en día la gran variedad de recursos de cómputo (GPUs, ASICs, FPGAs) puede llegar a ser abrumador y aún más las distintas maneras de programarlos. Los programadores que trabajan en entornos heterogéneos tienen que lidiar con más de un modelo y herramientas para conseguir utilizar todos los recursos. En este proyecto queremos facilitar el uso de entornos distribuidos heterogéneos llevando a cabo la integración de los modelos de programación *COMPSs* y *OmpSs*. *COMP Superscalar* (*COMPSs*) es un modelo de programación para entornos distribuidos basado en la generación de tareas y *Omp Superscalar* (*OmpSs*) intenta explotar el paralelismo de las aplicaciones dentro de un nodo haciendo uso de todos los recursos presentes. En el proyecto se realiza la integración *COMPSs+OmpSs-2* que permite programar entornos distribuidos heterogéneos de una manera sencilla y se evalúa su rendimiento.

Palabras clave: Entornos distribuidos heterogéneos, Tareas, Paralelismo, COMPSs, OmpSs

Resum

Avui en dia la gran varietat de recursos de còmput (GPUs, ASICs, FPGAs) pot arribar a ser aclaparadora i encara més les diferents formes de programar-los. Els programadors que treballen en entorns heterogenis han de lidiar amb més d'un model i eines per tal d'utilitzar tots els recursos. En aquest projecte volem facilitar l'ús dels entorns distribuïts heterogenis fent una integració dels models de programació *COMPSs* i *OmpSs-2*. *COMP Superscalar* (*COMPSs*) es un model de programació per entorns distribuïts basat en la generació de tasques i *Omp Superscalar* (*OmpSs*) intenta explotar el paral·lelisme de les aplicacions dins d'un node fent ús dels recursos d'aquest. En el projecte es realitza la integració *COMPSs+OmpSs-2* que permet programar entorns distribuïts heterogenis de manera senzilla i s'avalua el seu rendiment.

Paraules clau: Entorns distribuïts heterogenis, Tasques, Paral·lelisme, COMPSs, OmpSs

Abstract

Nowadays, variety of computing resources (GPUs, ASICs, FPGAs) can be overwhelming and even more the different ways of programming them. Programmers working in these distributed heterogeneous environments have to cope with more than a programming model and tools to make profit of these resources. In this project, we want to facilitate the use of distributed heterogeneous environments by doing an integration of programming models *COMPSs* and *OmpSs-2*. *COMP Superscalar* (*COMPSs*) is a task-based programming model for distributed environments and *Omp Superscalar* (*OmpSs*) exploits parallelism inside the node using the resources within. In the project the integration *COMPSs+OmpSs-2* is done, it allows the programming of distributed heterogeneous environments in an easy way and a performance evaluation is also done.

Keywords: Distributed heterogeneous environments, Task-based, Parallelism, COMPSs, OmpSs

Agradecimientos

Primero quisiera agradecer a mi madre Ingrid y a mi hermano Raul todo lo que vivimos y compartimos mientras pudimos, cada día pienso en vosotros. Gracias a mi padre Miguel Angel, a mi hermana Jessica, a mi hermano Miguel Angel, a mi abuela Teresa y a mi tío Raimundo, gracias por cuidarme y llevarme hasta quien soy hoy, gracias también a mi pareja Judit que últimamente es la que más me soporta, sois todo lo que tengo y lo mejor que tendré nunca.

Gracias a Rosa y Jorge por darme la oportunidad de realizar este proyecto y tener la responsabilidad de dirigirlo, lo único que queda por escrito es este trabajo pero las enseñanzas que me llevo son muchas más. Gracias a todos los compañeros de la sala por los consejos técnicos y no tan técnicos, estar en cualquier otra sala no hubiera sido lo mismo. Gracias a Joel, Mario, Oleksandr y Aleix por hacer más amenas y divertidas las horas de comer, los fines de semana de trabajo y en general cualquier momento libre del día, gracias también a todas las personas que aprecio y que me aprecian, llevo un poco de cada uno dentro.

Por último quiero dar gracias al equipo de *OmpSs-2* por todo el soporte brindado y a todas las personas que administran las máquinas del *BSC* que he estado utilizando estos meses.

Ha sido un placer, gracias.

Índice

Capítulo 1

Introducción

1.1. Contexto

Hoy en día se tiende a tener distintos recursos de cómputo en un solo dispositivo, por ejemplo, en un teléfono móvil ya tenemos al menos un procesador y una tarjeta gráfica, pero no tan sólo en el ámbito más cotidiano (aunque no nos demos cuenta), sino que en los más profesionales y especializados está siendo también cada vez más común la heterogeneidad de los recursos de computación en servidores, *clusters* y supercomputadores.

Programar estos dispositivos, que cuentan con esta heterogeneidad no es una tarea sencilla, un procesador no se programa de la misma manera que un acelerador, ni dos aceleradores tienen por qué programarse de la misma manera, existen una gran variedad de modelos de programación que nos brindan estas facilidades pero esta gran variedad se torna abrumadora y a veces repercute en la portabilidad de las aplicaciones. Es por esto que nos hemos decidido a hacer este proyecto, para mejorar en la medida de lo posible la facilidad de desarrollo y portabilidad de una aplicación en estas plataformas cada vez más complejas.

Si se facilita a la comunidad este tipo de programación, no solo habría una disminución del esfuerzo a realizar por parte del programador (hasta ahora, abismal), si no que mejoraría el desempeño de las aplicaciones que utilizasen todos los recursos que hay a su disposición en una máquina. Esto motiva la realización del proyecto, que pretende facilitar la gestión de estos recursos de computación en entornos distribuidos, brindando un método robusto y eficiente para programar aplicaciones en estos entornos.

Este proyecto es un Trabajo de Fin de Grado del Grado en Ingeniería Informática, especializado en el área de Ingeniería de Computadores. El grado es impartido por la *Facultat d'Informàtica de Barcelona (FIB)* centro perteneciente a la *Universitat Politècnica de Catalunya (UPC)*.

El proyecto se ha desarrollado conjuntamente con el *Barcelona Supercomputing Center (BSC)*, hemos estudiado como integrar los modelos de programación *COMPSs* y *OmpSs* para alcanzar este objetivo, se ha implementado y evaluado un prototipo.

1.1.1. Actores

Los actores en este proyecto son las personas que tomen parte en él, ya sea de manera directa o indirecta. Con esto quiero decir, que tanto las personas que han tomado parte en el desarrollo *per se* como las personas que se nutran de este, son los actores.

- **Desarrollador:** La figura del desarrollador **es** la persona que ha trabajado de manera más directa en el proyecto. En este proyecto únicamente hay un desarrollador que ha elaborado la documentación que leerás a continuación y ha dado forma a los objetivos tangibles del proyecto.
- **Director y codirector:** Pese a que el desarrollador ha tenido el papel principal en el proyecto, el director y el codirector han guiado a este en el camino abierto que es el desarrollo

del proyecto. Periódicamente ha habido reuniones de seguimiento donde se han supervisado las actividades del proyecto y su correcta realización.

- **Barcelona Supercomputing Center:** De manera directa el centro otorga al desarrollador un lugar de trabajo y un equipo informático. Por otra parte, cuenta con el soporte por parte de los equipos de desarrollo de *COMPSs* (del cual forma parte el desarrollador) y de *OmpSs* para cualquier incidencia relacionada con su *software* y derivados.

- **Beneficiarios:** El proyecto facilita el desarrollo de aplicaciones paralelas en entornos heterogéneos distribuidos, de manera que se espera que los beneficiarios sean cualquier desarrollador de esta, sea de la organización que sea.

1.2. Estado del arte

Existen muchos modelos de programación, véanse *OpenMP* [5], *OmpSs* [7], *MPI* [16], *COMPSs* [3] y un largo etcétera. De alguna manera el objetivo en común fue y es aprovechar los cada vez más abundantes recursos en las máquinas, que finalmente no sólo han crecido en abundancia si no en diversidad. La filosofía sigue siendo la misma, sacar el mayor rendimiento posible a nuestras máquinas.

Para esto son necesarios modelos de programación que nos den la posibilidad de utilizar los recursos y nos ayuden a explotar la posible sinergia entre estos en ciertas aplicaciones.

La diferencia más elemental entre los modelos, es en como se gestiona la memoria, o más bien cómo se estructura en el modelo.

- **Memoria compartida:** En un modelo con memoria compartida como *OpenMP* o *OmpSs*, el paralelismo se consigue a través de hilos o *threads* que ejecutan porciones de código. Los *threads* dentro de una aplicación comparten la memoria, es decir que el espacio de direcciones es el mismo para todos. De esta manera cualquier modificación será inmediatamente visible en el resto de *threads*.

- **Memoria distribuida:** En un modelo con memoria distribuida esta no es compartida entre los procesos que llevan a cabo el cómputo (en memoria compartida hilos), esto es que tienen su propio espacio de direcciones y en caso de necesitar un dato en concreto se debe efectuar una comunicación. Existen distintas formas de implementar un modelo con memoria distribuida.

  Paso de mensajes: *MPI* es un modelo de programación con memoria distribuida por paso de mensajes (*Message Passing Interface*) , cada proceso goza de un espacio de direcciones único y el programador es consciente de esto cuando programa la aplicación, es encargado de reservar la memoria y también de efectuar las comunicaciones, todo esto mediante llamadas a una *API*.

  Memoria global: A diferencia del anterior, el programador tiene una visión global de la memoria, pero cada proceso tiene su propio espacio de direcciones, en caso de que intente acceder a memoria perteneciente al espacio de direcciones de otro proceso, se hará la transferencia de uno a otro. Modelos de programación que hagan uso de memoria global son *UPC* [1], y *GASPI* [11] y *COMPSs* [3].

  Entornos heterogéneos: La memoria se encuentra dividida entre la correspondiente al *host* (el procesador) y el *device* (el dispositivo, que será un acelerador), para que este dispositivo compute hay que enviarle todos los datos que vaya a necesitar, por lo que la memoria entre el procesador y el acelerador está distribuida, modelos que hagan uso son *CUDA* [12] y *OpenCL* [14], por ejemplo.

La integración de los modelos propuestos *COMPSs* y *OmpSs* nos otorgará la posibilidad de utilizar todos los recursos de la máquina, e incluso de otras máquinas, ya que estaríamos usando la conjunción de un modelo basado en memoria compartida con uno en memoria distribuida, a continuación se ahonda en las características de ambos.

1.3. COMPSs

COMPSs está desarrollado por el grupo *Workflows and Distributed Computing* que pertenece al departamento de *CS - Computer Science*.

COMPSs es un modelo de programación para entornos distribuidos basado en la generación de tareas. El objetivo es hacer más sencilla la programación de aplicaciones y su ejecución en entornos distribuidos (clústers y *clouds*, por ejemplo). La generación de tareas la efectua un programa principal ejecutado en secuencial donde las tareas se especifican mediante la anotación de las funciones que se deseen.

El modelo está dotado de un sistema de *runtime* que coexiste con la aplicación. A medida que el programa principal encuentra las tareas y las envía al *runtime* este genera un grafo de tareas y se detectan las dependencias entre estas para ejecutarlas en el orden correcto. Dado que está preparado para ejecutarse en entornos distribuidos el *runtime* también detecta cuando son necesarias transferencias entre nodos.

1.3.1. Modelo de programación

El *runtime* de *COMPSs* está implementado en *Java* por lo cuál se soporta dicho lenguaje, además se desarrollaron los *bindings* de *Python* [15] y *C/C++* para facilitar el portaje de aplicaciones en estos lenguajes a *COMPSs*.

En este proyecto centraremos nuestros esfuerzos en *C* y *C++*. Para desarrollar una aplicación de *COMPSs* en *C/C++* necesitamos el programa principal (ejecutado en secuencial), una interfaz que especificará las funciones que *a posteriori* serán tareas, y el código que realmente implementan estas tareas.

Veamos en orden de enumeración ejemplos de los componentes de una aplicación.

```
1 interface ejemplo {
2     void funcionEjemplo(in int a, out int[a] array_a);
3 };
```

Código 1.1: Interfaz de la aplicación 'ejemplo'.

La interfaz de la imagen superior define una función llamada *funcionEjemplo* con un parámetro de entrada y uno de salida, las palabras clave *in* y *out* respectivamente otorgan estas propiedades a los parámetros, también se puede utilizar *inout*, el parámetro será de entrada y salida. Cualquier llamada a la función será ejecutada como tarea.

```
1      compss_on();
2
3      int a = 10;
4      int* array_a;
5
6      funcionEjemplo(a, array_a);
7
8      compss_wait_on(array_a);
9
10     compss_off();
```

Código 1.2: Fracción del código del programa principal

Esta imagen muestra el código del programa principal. Es tan sencillo como prometía, encendemos el *runtime* de *COMPSs*, preparamos los parámetros de la función, ejecutamos y finalmente esperamos a los parámetros de salida y apagamos el *runtime*.

```
1 void funcionEjemplo(int a, int array_a[]) {
2     for (int i = 0; i < a; ++i) {
3         array_a[i] = a;
4     }
5 }
```

Código 1.3: Implementación de la función 'funcionEjemplo'

Las tareas se implementan como funciones que serán ejecutadas por los *workers*. Salvo convenciones en los nombres de los ficheros para hacer la compilación de la aplicación esto es todo lo necesario para desarrollar una aplicación en *COMPSs* de *C/C++*.

1.3.2. Modelo de ejecución

El modelo de ejecución es sencillo, muy similar al modelo *thread-pool*, al iniciar el *runtime* se levanta el *master* y un conjunto de *workers*, a medida que se vayan generando tareas se estudiará qué *workers* están libres y si cumplen los requisitos para ejecutar dicha tarea, y en ese caso la ejecutarán.

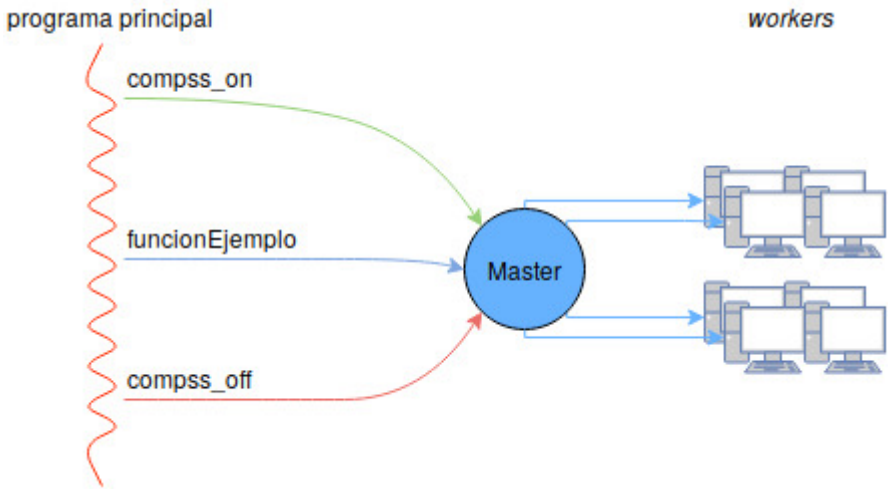

Figura 1.1: Modelo de ejecución, basado en la aplicación 'ejemplo'.

La imagen muestra lo que sucede al ejecutar la aplicación de ejemplo de la sección 1.3.1, las líneas rojas curvas indican procesos (o bien *threads*), lo importante sucede entre las flechas etiquetadas como compss_on y compss_off, la ejecución de la tarea. Se pide ejecutar la tarea funcionEjemplo al *runtime* de *COMPSs* y se decide en qué *worker* se ejecutará. Hasta aquí podríamos pensar que es indéntico al *thread-pool*. Nótese, que esto no es así, por el hecho de que no tenemos por qué hablar de una misma máquina, sino que pueden ser máquinas distintas, como se puede ver en el grupo de ordenadores de la imagen.

En *C/C++* existen dos tipos de *worker*, uno **no persistente** y otro **persistente**.

- **No persistente:** Para cada tarea que se quiere ejecutar en uno de estos workers se debe crear el *thread* y unas *pipes* para hacer *Inter-process communication* (IPC).
- **Persistente:** Este tipo de *worker* se levanta una vez al inicio de la aplicación y espera recibir las tareas a ejecutar.

1.4. OmpSs

OmpSs es desarrollado por el grupo *PM - Programming Models*, perteneciente también al departamento de *CS - Computer Science*.

OmpSs es un modelo de programación que intenta explotar el paralelismo de las aplicaciones de una manera sencilla y aprovechando al máximo los recursos de la máquina [7].

El nombre del modelo proviene de *OpenMP* y *StarSs* (modelo que desarrolló el *BSC*), este integra funcionalidades presentes en ambos. Por parte de *OpenMP* se quiere tomar la facilidad de paralelizar una aplicación secuencial insertando pragmas, y de *StarSs* el modelo de ejecución basado en un *thread-pool* y que permite la ejecución de código en más recursos que únicamente el procesador, es decir, que ofrece fácil gestión del resto de recursos de cómputo (*GPUs*, *FPGAs*, ...) [13] [9].

1.4.1. Modelo de programación y ejecución

El modelo de programación de *OmpSs* se basa en la generación de tareas sencillamente insertando pragmas en código secuencial y a su vez facilitando la gestión de recursos heterogéneos. Veamos un pequeño ejemplo que muestre estas facultades.

```
1 for (int i = 0; i < N; ++i) {
2     for (int j = 0; j < N; ++j) {
3         for (int k = 0; k < N; ++k) {
4             #pragma omp target device(cuda) \
5                                 copy_deps     \
6                                 ndrange(2,N,N,32,32)
7             #pragma omp task inout([N*N]C) in([N*N]A, [N*N]B)
8             multiply_partitions_GPU(A[i*N+k],
9                                     B[k*N+j],
10                                    C[i*N+j],
11                                    n);
12        }
13    }
14 }
```

Código 1.4: Multiplicación de un bloque de una matriz utilizando GPUs.

El código muestra una multiplicación de matrices por bloques. Con el primer pragma se indica que el dispositivo objetivo es una tarjeta gráfica que soporte *CUDA*, y el segundo la declaración de una tarea y el tamaño de los bloques junto a las dependencias de esta.

El modelo de ejecución consiste en un *thread-pool*, es decir, al generar tareas se escogerán *threads* del *pool* (entendámoslo como un conjunto de *threads*) para ejecutarlas.

Cabe decir que para compilar una aplicación de *OmpSs*, se utiliza el compilador *source-to-source Mercurium* y el runtime *Nanos++* para gestionar el paralelismo, es decir, la creación de tareas, sincronización entre estas, etc.

1.5. Trabajo previo

Actualmente existe la posibilidad de desarrollar aplicaciones que utilicen *OmpSs* dentro de *COMPSs*.

Para poder interactuar con el *runtime* de *OmpSs Nanos++*, necesitamos gestionarlo nosotros manualmente o bien utilizar los *pragmas* que nos propone el modelo de programación y compilar con *Mercurium*. Entonces, asegurándonos de registrar los *workers* en el *runtime* y compilando su código fuente con *Mercurium* aseguramos la interacción con el *runtime* y por lo tanto la integración de ambos modelos.
Es posible desarrollar aplicaciones, pero con ciertas limitaciones. Dado que el worker *no persistente*, se debe levantar para cada tarea conseguimos aislar las tareas entre sí, cada tarea tendrá un *runtime* de *OmpSs* para el mismo, pero este mismo motivo añade mucho *overhead* para la granularidad de las tareas de *OmpSs*. El *worker persistente* carece del *overhead* de poner en marcha el *runtime* de *OmpSs* pero también pierde la característica de aislar cada tarea del resto.

El factor de aislamiento se vuelve todavía más crítico cuando en *OmpSs* efectuar un *taskwait* (*pragma* que espera hasta que las tareas acaben) conlleva a que la espera se haga sobre todas las tareas de *OmpSs* que estén en el *runtime* por lo cual si no contamos con el aislamiento esperaremos a todas, hayan sido creadas o no por la misma tarea de *COMPSs*. Para solucionarlo hay que efectuar

accesos concurrentes sobre los datos que estemos tratando y hacer que el *taskwait* tenga como dependencia esos datos, así no esperaremos a absolutamente todas las tareas.

Estos problemas pretenden arreglarse integrando *OmpSs-2*, que ofrece un *runtime* nuevo llamado *Nanos6* y nuevas características que vienen con este cambio como por ejemplo:

- **Liberación de las dependencias:** Las dependencias en esta versión se liberan de manera temprana, es decir, una vez una tarea acaba con un dato lo notifica al *runtime* y este se encarga de notificar a las tareas que quedan libres de esta dependencia.
- **Relajación de las dependencias:** Ahora se pueden utilizar nuevos pragmas para determinar dependencias más suaves, no tan estrictas como en la versión anterior.
- **Ejecución de funciones de manera asíncrona:** La *API* del nuevo *runtime Nanos6* permite ejecutar de manera asíncrona funciones en forma de tarea a partir de un puntero a una función, este método proporciona aislamiento del resto de tareas.

Hemos listado las que de alguna manera han sido clave para el proyecto, *OmpSs-2* implementa muchas otras mejorías a parte de estas [10]. El proyecto estudia en los capítulos 4 y 3 si los problemas planteados han sido solucionados y el rendimiento que desempeña la nueva integración.

1.6. Alcance del proyecto

En esta sección se declaran las intenciones del proyecto (qué se pretende hacer), mediante una serie de requerimientos que harán que el proyecto pueda ser acabado con éxito, y unos objetivos que marcarán el desarrollo de este. También los posibles riesgos que surjan (y las soluciones de estos) y la metodología de trabajo que se llevará a cabo.

1.6.1. Requerimientos

Los requerimientos necesarios para este proyecto son:

- La nueva integración con *OmpSs-2* no debe romper la actual compatibilidad con *OmpSs*.
- El rendimiento de las aplicaciones desarrolladas con *COMPSs+OmpSs-2* debe mejorar.
- Todas las modificaciones sobre *COMPSs* deben ajustarse al grupo de *Workflows and Distributed Computing*.
- Cualquier requerimiento impuesto (o aconsejado) por el *BSC* formará parte de esta lista.

1.6.2. Objetivos

El objetivo principal de este proyecto es reformular la integración de *COMPSs* con *OmpSs* para solucionar los problemas actuales, ya sea integrándolo de nuevo pero esta vez con *OmpSs-2* o bien idear otra manera de resolver las problemáticas. La siguiente lista muestra la posible descomposición de objetivos:

- Aprender a utilizar la API (*Application Programming Interface*) de *Nanos6*.
- Eliminar o bien reducir las problemáticas planteadas en la sección 1.5.
- Mejorar la gestión de recursos heterogéneos en las aplicaciones desarrolladas en *COMPSs* integrando *OmpSs-2*.
- En caso de conseguir el resto de objetivos plantear la integración en *Java* y *Python*.

1.6.3. Riesgos

Durante el desarrollo del proyecto pueden surgir problemas, para mejorar la reacción ante ellos listaremos los posibles riesgos y las respectivas soluciones.

1.6.3.1. Problemas con el material de desarrollo

Se podría romper el equipo en el cual se desarrolla el proyecto, pongamos que de una pantalla, un teclado y un ordenador se rompe el último. Se perdería todo el avance del proyecto, incluso documentación.

Solución: Pese a que la pérdida del ordenador es importante, todo el código del proyecto será subido al *GitLab* de *WDS*, y la documentación al *GitHub* personal del desarrollador, por lo cual se podría recuperar todo el proyecto.

1.6.3.2. Problemas con los clústers y supercomputadores

Si por casualidad, caen los clústers y supercomputadores en los cuales se medirá el rendimiento del proyecto, se pararía la obtención de las métricas.

Solución: En este caso, como no resultaría lo mismo ejecutarlo en local en mi ordenador, debería optar por realizar otras tareas hasta que el equipo del *BSC* solucione los inconvenientes.

1.6.3.3. Aparición de errores en la implementación

Cualquier proyecto esta lleno de errores en la implementación, hay que saber encontrarlos y solucionarlos lo más rápido posible, pero puede entorpecer el proyecto.

Solución: Activaremos los *flags* de *debug* para poder evitar el mínimo error y en caso de su aparición utilizaremos *gdb* (*GNU Debugger*) para encontrarlo.

1.6.3.4. Problemas con OmpSs/OmpSs-2

Dado que se realizará una integración de otro proyecto del *BSC* el desarrollador puede encontrarse con dificultades relacionadas con este a lo largo del proyecto.

Solución: Después de haber intentado solucionarlo por sus propios medios se pondrá en contacto con el grupo de *Programming Tools* con la descripción del error e intentos de solucionarlo.

1.6.4. Metodología

Para decidir la metodología a utilizar, hay que tener en cuenta que el proyecto consta únicamente de tres personas que se envolverán en él. El desarrollador, que hará todo el desarrollo tangible, el director y el co-director.

La metodología que más se ciñe a las características del "equipo" es la iterativa. Consiste en planear las tareas a realizar y hacer una predicción de qué se conseguirá hacer en cortos periodos de tiempo llamados iteraciones. Además de estas predicciones, se consultará el estado del proyecto a menudo y se realizarán las tareas, una vez acabada una iteración empezará la siguiente, así hasta finalizar el proyecto.

Esta metodología nos permitirá reaccionar rápidamente a los imprevistos, además de hacer un bueno monitoreo del estado del proyecto, por lo cual se ha decidido emplearla.

1.6.4.1. Herramientas

Para efectuar el seguimiento del proyecto junto a mi director y codirector, se empleará un *workspace* de *Slack* para las comunicaciones directas (decidir dónde y cuándo hacer las reuniones por ejemplo), y *Trello* para gestionar las tareas a desempeñar en cada iteración. Como ya se ha mencionado anteriormente, para el control de versiones del proyecto, código y documentación se utilizará respectivamente *GitLab* y *GitHub*.

Capítulo 2

Integración de OmpSs-2 en C/C++ COMPSs

Con tal de realizar una buena integración necesitamos conocer bien cómo funcionan y/o cómo están hechos los componentes a integrar. Con tal de adquirir los conocimientos necesarios, vamos a indagar en la estructura interna del *binding* de *C/C++* de *COMPSs* y a entender cómo se desarrolla y compila una aplicación. También deberemos ver cómo funciona la *API* del *runtime* de *OmpSs-2 Nanos6* y el proceso habitual de desarrollo y compilado de una aplicación que utiliza el modo librería.

2.1. Estructura de los bindings

El *runtime* de *COMPSs* fue desarrollado en *Java*, por lo que si queremos soportar cualquier lenguaje (salvo el propio *Java*), necesitamos de alguna manera establecer comunicación con ese lenguaje. Es decir, necesitaremos un mecanismo que nos permita ejecutar código de este lenguaje a soportar, por supuesto, en ambas direcciones.

Actualmente, *COMPSs* cuenta con los *bindings* de *C/C++* y *Python* (usualmente conocido como *PyCOMPSs*), que utilizan estos mecanismos descritos anteriormente. Para efectuar la ejecución entre *Java* y *C/C++* se utiliza la *Java Native Interface*, dado que desde *Python* se puede utilizar la *Python-C API* para , con un componente intermedio entre *Python* y *Java* escrito en *C/C++* (utilizando la *JNI*) permitiríamos efectuar llamadas desde *C/C++* y *Python* al *runtime* y al revés.

La siguiente imagen muestra la estructura general de los *bindings* actuales. Las cajas representan componentes de la arquitectura, el color de cada caja ha sido escogido para representar un lenguaje, por lo que dos cajas del mismo color que estén conectadas directamente con un flecha no requieren de mecanismos adicionales.

2.2. Modelo de ejecución en el binding C/C++

Para poder obtener el modelo de ejecución introducido en la sección 1.3.2, necesitaremos compilar dos binarios, uno para el *master* y otro para los *workers*. La aplicación que desarrolle el usuario, tiene que ser transparente al *runtime*, se necesita una interfaz indicando las tareas a detectar al ejecutar la aplicación. Entonces, una vez la compilemos, se generará automáticamente (a partir de ahora *autogenerar*) código para las tareas, que gestionarán la comunicación con el *runtime*.

La siguiente imagen describe de manera visual cómo se compila una aplicación.

Para facilitar el compilado de la aplicación se utiliza el script *compss_build_app*, y para hacer la generación de código a partir de la interfaz, el binario *compss_generator*. La aplicación una vez desarrollada por el usuario, compilada sin *COMPSs* por el medio también funcionaría, pero sencillamente las tareas serían ejecutadas *in situ* en el *master*. Lo que pretendemos hacer al introducir

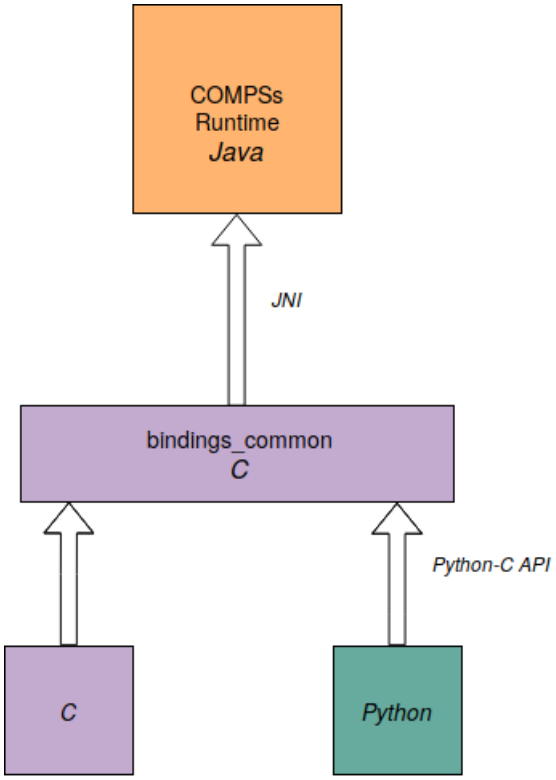

Figura 2.1: Estructura de los bindings de COMPSs.

COMPSs es que el *master*, en vez de ejecutar la tarea, comunique al *runtime* que se debe ejecutar una tarea, y este se encargue de ejecutarla en un *worker*, todo de manera transparente al usuario.

En la figura 2.2 aparecen los ficheros generados con *compss_generator* y la interfaz, estos ficheros son el *stubs* y *executor* (siempre del estilo, *ejemplo-stubs.cc* y *ejemplo-executor.cc* donde ejemplo es el nombre de la aplicación compilada). El fichero *stubs* corresponde con la parte del *master* que se encargará de comunicarse con el *runtime* para registrar la tarea. Esto lo consigue implementando las funciones de las tareas con el código para gestionar su registro, es decir, sustituyendo el código que sería propio de la ejecución de cada tarea por un código para efectuar el registro en el *runtime*. Una vez registrada la tarea, eventualmente el *runtime* designará a un *worker* a ejecutarla. El *worker* recibirá la tarea que debe ejecutar, los datos necesarios y más parámetros, el *executor* ha sido *autogenerado* con el código principal de la aplicación, por lo que conoce como gestionar los datos y parámetros recibidos para ejecutar la tarea pertinente.

2.3. Modo librería: nanos6_spawn_function

En la sección 2.2 dimos una visión más interna y propia de desarrollador del *binding* de *C/C++*. Sobre *OmpSs-2* necesitamos saber, como se desarrolla una aplicación, en concreto con el modo librería.

Esta funcionalidad, nos permite registrar una tarea en el *runtime* de *OmpSs-2* de manera asíncrona, con lo cuál podremos brindar a cada tarea y sus sucesoras un entorno aislado del resto. Gracias a esto, conseguiremos arreglar el problema con la migración de *threads*.

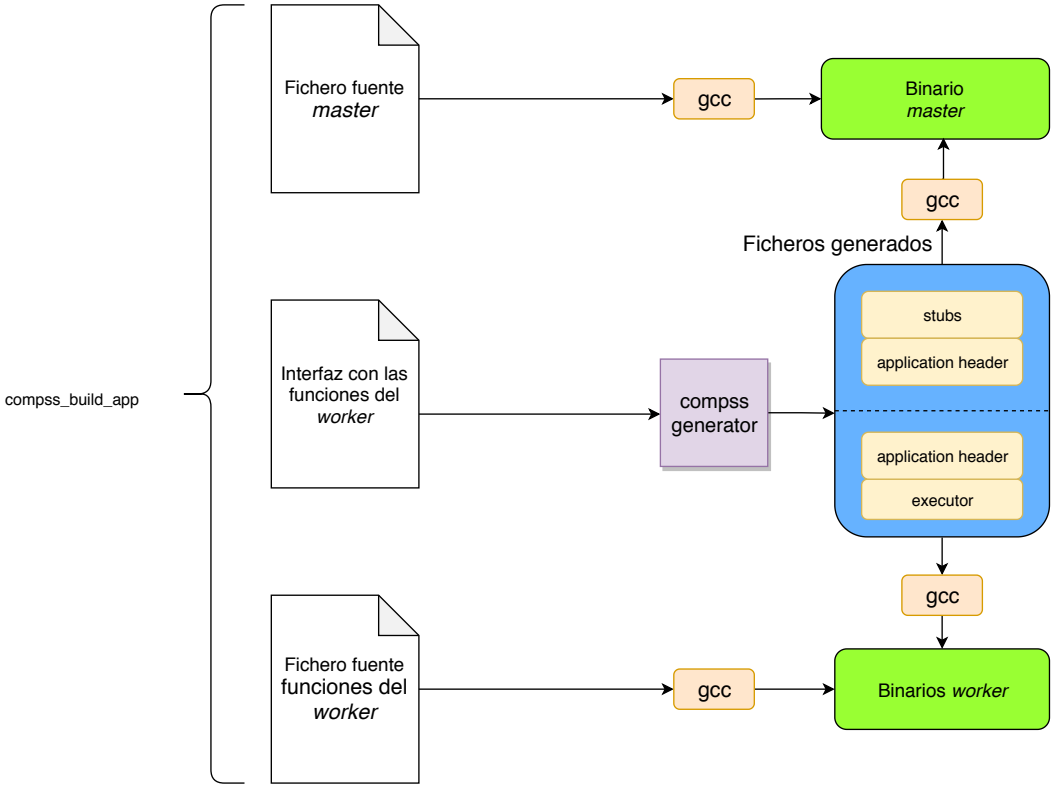

Figura 2.2: Proceso de compilación de una aplicación COMPSs C/C++.

```
1 void nanos6_spawn_function(
2     void (*function)(void *),
3     void *args,
4     void (*completion_callback)(void *),
5     void *completion_args,
6     char const *label
7 );
```

Código 2.1: Definición de la función nanos6_spawn_function.

La anterior imagen muestra la definición de la función que nos permitirá registrar una función **function** con argumentos **args** en el *runtime*. Una vez ejecutada la función registrada como tarea, se ejecutará la función **completion_callback** con argumentos **completion_args** (mecanismo conocido como *callback*, de ahí el nombre), el argumento de la función *label* sirve para etiquetar la tarea con el nombre que contenga.

2.3.1. Ejemplo

Con tal de entender como funciona un programa que utilice el modo librería, vamos a ver un ejemplo sencillo. Este programa de ejemplo, hará *spawn* de una función y dentro de ésta se generarán tareas con dependencias entre ellas.

```
1  int nanos6_ejemplo(int* a) {
2
3      int local_a;
4
5      #pragma oss task shared(local_a) out(local_a)
6      {
7          local_a = a[0];
8      }
9
10     #pragma oss task shared(local_a) inout(local_a)
11     {
12         local_a = local_a * 4;
13     }
14
15     #pragma oss taskwait
16
17     return local_a;
18 }
```

Código 2.2: Función a ser registrada como tarea.

En la imagen 2.2 podemos ver la función **nanos6_ejemplo**, ésta tiene como parámetro un puntero a enteros **a** y retorna un valor del tipo **int**. La función será registrada como tarea desde otro fichero. El cálculo que se realiza en la función no tiene la menor importancia, es solo un **ejemplo**. Por supuesto, esta función será compilada con *Mercurium*, si no fuera el caso, la función se ejecutaría correctamente pero no se generarían las tareas de dentro de la función, por lo cuál la ejecución sería secuencial.

```
1      char const *error = nanos6_library_mode_init();
2      if (error != NULL)
3      {
4          fprintf(stderr, "Error initializing Nanos6: %s\n", error);
5          return 1;
6      }
7
8      condition_variable_t cond_var = COND_VAR_INIT;
9
10     ejemplo_wrapper_args_t args;
11
12     int A = 1;
13     args.array = &A;
14
15     nanos6_spawn_function(nanos6_wrapper, &args,
16                           condition_variable_callback, &cond_var,
17                           "spawned ejemplo");
18
19     wait_condition_variable(&cond_var);
20
21     printf("%d\n", args.ret);
22
23     nanos6_shutdown();
```

Código 2.3: Gestión del modo librería desde el main.

Las funciones *nanos6_library_mode_init()* y *nanos6_shutdown()* efectúan respectivamente el encendido y apagado del *runtime*, el código que vemos entre estas dos llamadas es el correspondiente para registrar una función como tarea utilizando el modo librería. Dado que la función nanos6_spawn_function espera como un puntero los parámetros a pasar a la función, se deben incluir todos dentro de una estructura que los pueda contener, el *struct ejemplo_wrapper_args_t* tiene los campos **a** y **ret** que corresponden al parámetro de la función *nanos6_ejemplo* y valor de

retorno de ésta.

Un detalle que no se ha comentado es que la ejecución de este código será efectuada por *pthreads*, el estándar *POSIX* de *threads*. Dado que la ejecución de la tarea tendrá lugar de manera **asíncrona** es necesario un mecanismo de sincronización, la implementación de *threads* que utilizamos (y por tanto los mecanismos de sincronización dependerán de estos).

Requerimos de un mecanismo de sincronización por que la función que se registra como tarea la ejecuta un *thread* diferente al que la registra, se hace uso de la librería *pthread* [4].

2.3.2. Mecanismo de sincronización

Un *thread* perteneciente al *worker* de *COMPSs* registra una tarea y un segundo thread perteneciente a *OmpSs-2* espera a poder ejecutar alguna tarea. El primer *thread* una vez haya registrado la tarea procederá a bloquearse, el segundo cuando acabe de ejecutar la tarea registrada, ejecutará un *callback* donde desbloquearemos al primer *thread*.

El mecanismo abstracto es sencillo, pero la implementación real depende de los *Pthreads*. Las estructuras utilizadas para realizar la sincronización son *mutexes* y *condition variables*. Un *mutex* es una estructura que sirve para definir regiones de exclusión mútua mediante métodos de adquisición y liberación de la región, en este mecanismo se utilizan tan sólo por que son necesarios para utilizar *condition variables*. Las *condition variables* nos permiten que un *thread* espere a un evento de manera no bloqueante, si no, habría que hacer *polling* y consumir tiempo de cómputo inútilmente.

Será con las funciones *pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mutex)* y *pthread_cond_signal(pthread_cond_t * cond)* con las que respectivamente bloqueemos al primer *thread* después de registrar la tarea y lo desbloqueemos al acabar de ejecutarla.

Dado que para utilizar *condition variables* se necesitan *mutexes* y se recomienda utilizar una variable auxiliar para evitar abrazos mortales (en el caso de que, el segundo *thread* ya ha intentado desbloquear al primero, pero este aún no se había bloqueado, cuando se bloquee, se bloqueará para siempre), se crea una estructura de datos para contener todas estas variables llamada *condition_variable_t*.

```
1 typedef struct {
2     pthread_mutex_t _mutex;
3     pthread_cond_t _cond;
4     int _signaled;
5 } condition_variable_t;
```

Código 2.4: Definición de la estructura de datos condition_variable_t.

```
1 void wait_condition_variable(condition_variable_t *cond_var)
2 {
3     pthread_mutex_lock(&cond_var->_mutex);
4     while (cond_var->_signaled == 0) {
5         pthread_cond_wait(&cond_var->_cond, &cond_var->_mutex);
6     }
7     pthread_mutex_unlock(&cond_var->_mutex);
8 }
```

Código 2.5: Función wait_condition_variable

En la imagen 2.3 se utiliza la función *wait_condition_variable(&cond_var)*, la llamada a *pthread_cond_wait* está dentro de una región crítica para prevenir que nadie más se bloquee sobre la misma *condition variable* y esta se encuentra dentro de un *while* por que puede ocurrir que el *thread* se desbloquee sin haber sido desbloqueado por otro *thread*, la variable *cond_var->_signaled* será 0 hasta que el *thread* que produzca el desbloqueo lo modifique, así garantizaremos que el desbloqueo es intencionado.

```
1  void condition_variable_callback(void *untyped_arg)
2  {
3      condition_variable_t *cond_var = (condition_variable_t *)
4                                       untyped_arg;
5
6      pthread_mutex_lock(&cond_var->_mutex);
7      cond_var->_signaled = 1;
8      pthread_cond_signal(&cond_var->_cond);
9      pthread_mutex_unlock(&cond_var->_mutex);
10 }
```

Código 2.6: Callback de la tarea registrada

La imagen anterior muestra el *callback* que se ejecuta al finalizar la ejecución de la tarea, modifica el valor de la variable *cond_var->_signaled* y efectúa un *pthread_cond_signal* para despertar al primer *thread*.

Como es necesario utilizar una estructura auxiliar para pasar los parámetros mediante un puntero, necesitaremos también una función intermedia con la cuál llamar a la función que realmente queremos registrar como tarea. Habitualmente a este tipo de funciones intermedias se les llama *wrapper*. En la siguiente imagen veremos la función que actúa como *wrapper* de la función *nanos6_ejemplo*.

```
1   void ejemplo_wrapper(void *untyped_arg)
2  {
3      ejemplo_wrapper_args_t *args =
4          (ejemplo_wrapper_args_t *) untyped_arg;
5      args->ret = nanos6_ejemplo(args->a);
6  }
```

Código 2.7: Wrapper de la función nanos6_ejemplo.

El puntero de tipo *void* puede contener cualquier tipo de estructura, aprovechando que sabemos que únicamente deberá contener el tipo *ejemplo_wrapper_args_t* se hace un *cast* para interpretarlo como la estructura deseada. Se asigna **ret** al valor de retorno y se pasa **a** como parámetro.

2.3.3. Compilado

El modo librería nos permite desacoplar de alguna manera la parte que pertenece a *OmpSs-2* de la que no. Es decir, ahora podemos tener un código principal encargado de hacer *spawn* de funciones que no tiene por qué estar compilado con *Mercurium* y el *flag --ompss-2*.

Un *Makefile* para compilar el ejemplo 2.3.1 sería:

```
1  all: ejemplo
2
3  # Codigo principal
4  main-ejemplo.o : main-ejemplo.c nanos6-ejemplo.h nanos6-helpers.h
5      $(CC) $(CFLAGS) -c main-ejemplo.c -o $@
6
7  # Mecanismos de sincronizacion
8  nanos6-helpers.o : nanos6-helpers.c nanos6-helpers.h
9      $(CC) $(CFLAGS) -c nanos6-helpers.c -o $@
10
11 # Nanos6 (codigo a compilar con Mercurium y ompss-2)
12 nanos6-ejemplo.o :  nanos6-ejemplo.c nanos6-ejemplo.h
13     $(MCC) $(CFLAGS) -c  nanos6-ejemplo.c -o $@
14
15 # Linking
16 ejemplo: main-ejemplo.o nanos6-ejemplo.o nanos6-ejemplo.o
17     $(CC) $^ -o $@ $(LIBS)
```

Código 2.8: Makefile para compilar el ejemplo.

Se divide el *Makefile* en cuatro etapas, la primera compila el *main-ejemplo.c* utilizando el compilador definido por la variable de entorno *CC*, en nuestro caso será *g++*, la segunda etapa compila las funciones utilizadas para los mecanismos de sincronización descritos anteriormente en el fichero *main-helpers.c*, la tercera compila el código que contiene la o las funciones a ser ejecutadas como tarea con el compilador definido por la variable de entorno *MCC*, que es *Mercurium*, y por último en una fase de *linking* se enlazan los objetos compilados entre sí, y además con el objeto *nanos6-library-mode.o* y la librería *nanos6*, esto es necesario por que el objeto generado al compilar el *main-ejemplo.c* utiliza las llamadas para encender el modo librería descritas en la sección 2.3.1.

2.4. Integración

Con todo lo que se ha explicado acerca de *OmpSs-2* y la funcionalidad del modo librería, se puede realizar la integración, esta sección muestra la forma final que han tomado los componentes al hacer la integración. Una vez realizada la integración, el proceso de compilación es el de la imagen 2.3.

La única parte de toda la aplicación que hay que compilar con *Mercurium* y el *flag --ompss-2* es el fichero fuente que contiene las funciones del *worker*. Con este fichero crearemos una librería que llamaremos *libfunctions.a*, conjuntamente con las dependencias del modo librería de *OmpSs-2* y los ficheros autogenerados para el *worker* se compilarán los binarios *worker_c* y *nio_worker_c*. El resto de binarios, objetos y librerías se pueden compilar con *gcc* o cualquier otro compilador, esto hace mucho más flexible todo el proceso de compilado de una aplicación *COMPSs+OmpSs-2*.

El nuevo proceso de compilado mantiene las características del anterior ya que genera los mismos binarios que antes y pasa por etapas similares para compilarlos, de esta manera la ejecución de una aplicación se lleva a cabo como antes. Hemos hecho la integración *COMPSs+OmpSs-2* de manera transparente al resto de componentes, lo cuál es muy positivo.

En el ejemplo para utilizar el modo librería 2.3, aprendimos como activar el *runtime* de *OmpSs-2* en este modo y como ejecutar las tareas, por lo que bastará con activarlo en los *workers* (persistente y no persistente) y modificar la generación del código *executor* (queremos hacerlo en el *worker*) para generar la misma estructura del ejemplo en cada tarea de la interfaz de la aplicación. En la imagen 2.4 se ve de manera visual lo que vimos en la sección 2.3 aplicado para la integración, se omiten las funciones para el mecanismo de sincronización ya que son siempre las mismas.

Entonces, cada vez que al *worker* se le pida ejecutar el código asociado a una tarea en concreto, si ha sido compilada para *OmpSs-2* acabará pasando por este código, de manera que se llamará al *nanos6_spawn_function* y se ejecutará en el *runtime* de *OmpSs-2*. Está claro que *OmpSs-2*

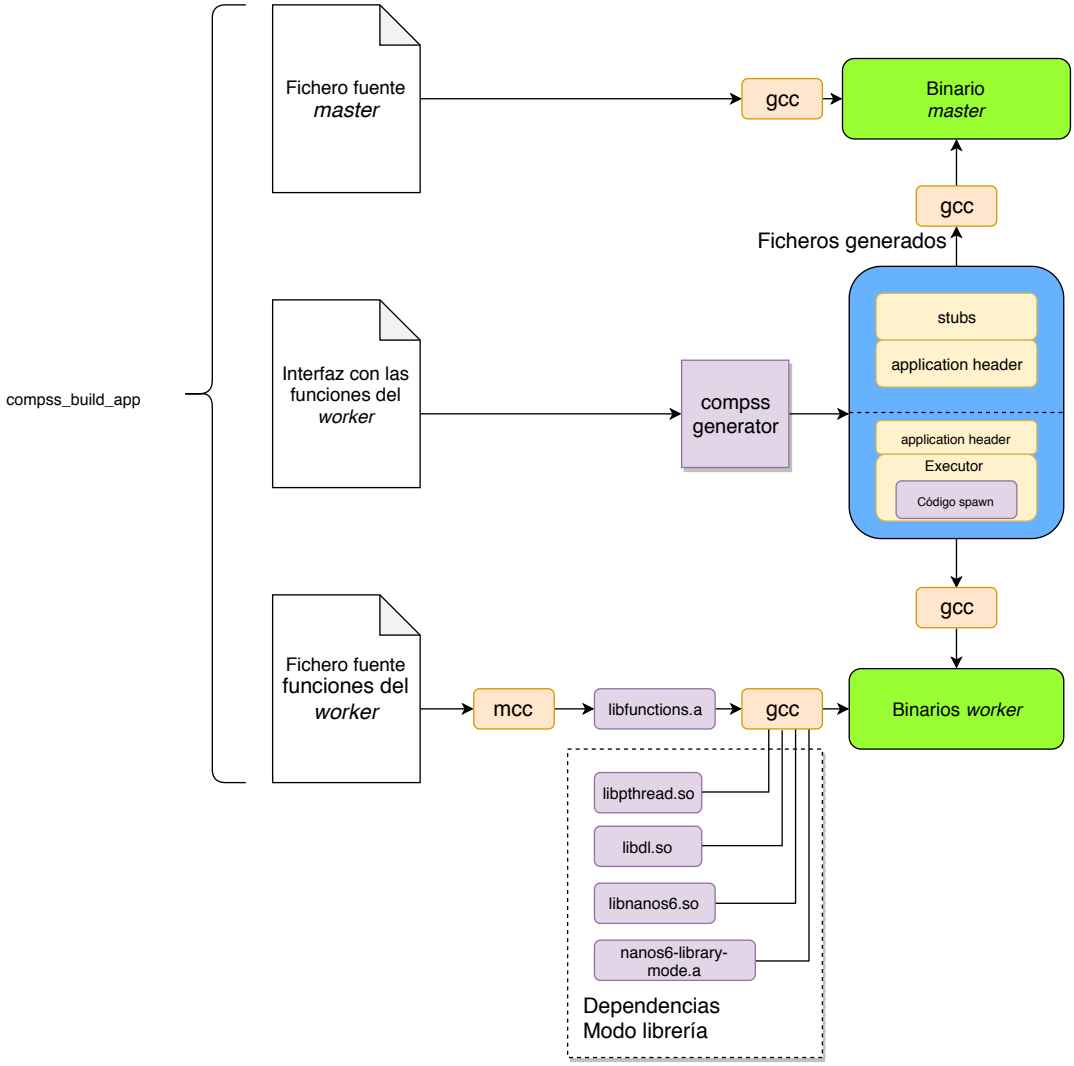

Figura 2.3: Proceso de compilación de una aplicación COMPSs+OmpSs-2 C/C++.

tomará parte únicamente en los nodos del tipo *worker*, ya que es donde se ejecutarán las tareas detectadas por el *master*, entonces con la activación de *OmpSs-2* en los nodos *worker* conseguiremos la siguiente estructura y modelo de ejecución.

Durante la ejecución del programa principal se irá generando un grafo con las tareas que se deben ejecutar, eventualmente cada una de estas será enviada a un nodo *worker* (en la figura tan sólo aparecen dos, pero podrían ser más) y este ejecutará la tarea de *COMPSs* en forma de tarea de *OmpSs-2*. La ejecución de las tareas de *OmpSs-2* generará otro grafo, dónde figurarán tanto las tareas que han sido creadas mediante el mecanismo de *spawn* como las que puedan ser creadas dentro de las anteriores, y finalmente las tareas de este grafo serán ejecutadas. En la imagen 2.5 se explica visualmente el modelo de ejecución, el color de las tareas de *COMPSs* representa una tarea en concreto y se puede seguir su recorrido a lo largo de la ejecución.

Si juntamos la compilación y la ejecución en una sola figura resulta en la imagen 2.6. A pesar que no da tantos detalles de cada componente como en las imágenes 2.3 y 2.5, en esta imagen se ve claramente para qué se utiliza cada componente y cuál es el efecto que tiene en cada fase.

También en la integración se ha añadido soporte a los tipos enumerados y a la inclusión de cabeceras en la interfaz. Lo primero nos permite dar soporte a más tipos de datos y lo segundo es esencial para cuando se quieren utilizar librerías externas que requieren cabeceras. Ahora en una interfaz se puede utilizar *enum* e incluir cabeceras, lo cuál se hace como en la imagen 2.7.

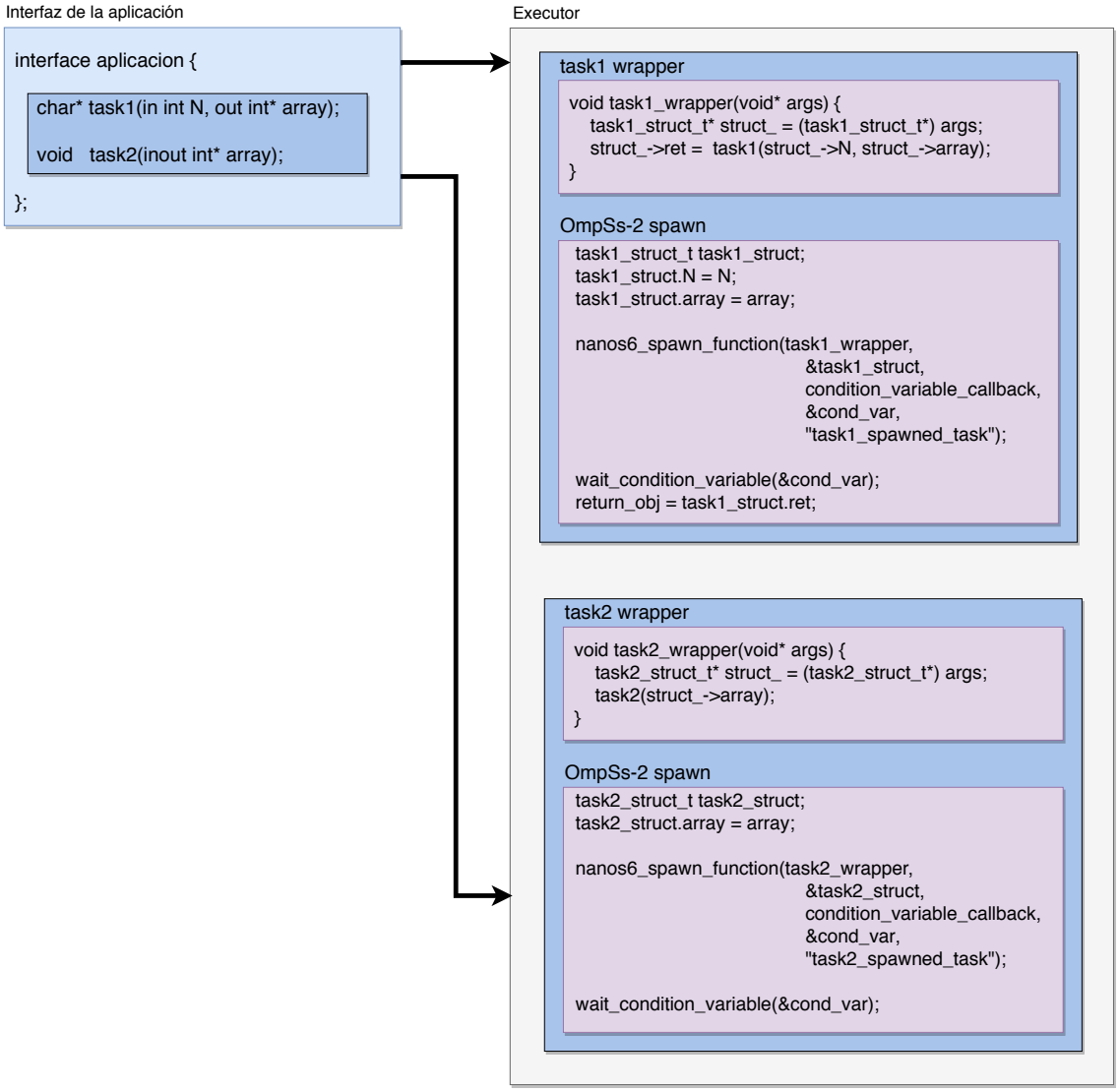

Figura 2.4: Generación del código para ejecutar las tareas de COMPSs en OmpSs-2.

Con tal de dar una explicación más sencilla, hemos evitado poner todo el código en esta sección, si se quiere más detalle o alguna parte no se entiende, en el apéndice C se encuentra una explicación técnica y con código de todo relativo a la integración.

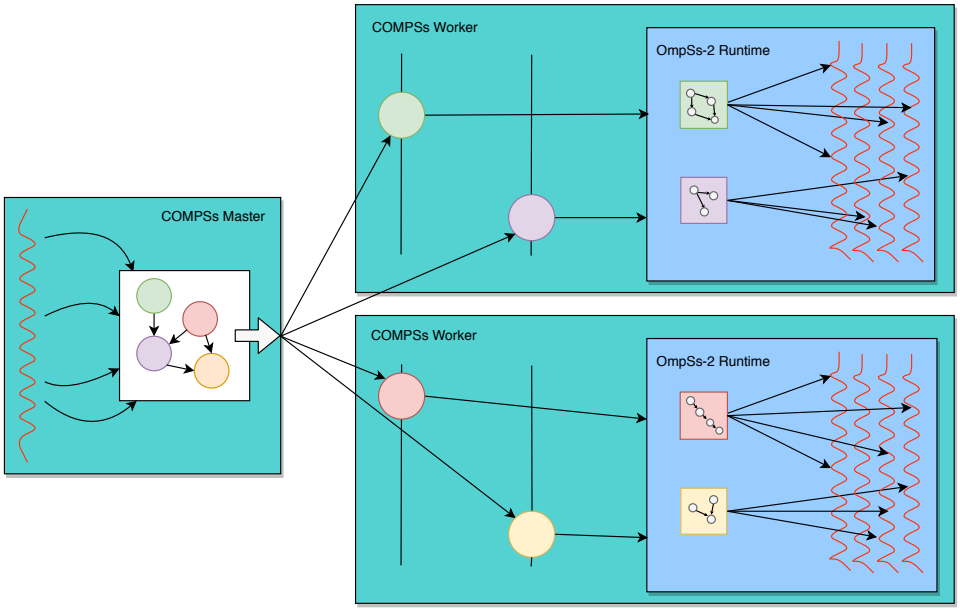

Figura 2.5: Estructura y modelo de ejecución master-worker de la integración COMPSs+OmpSs-2

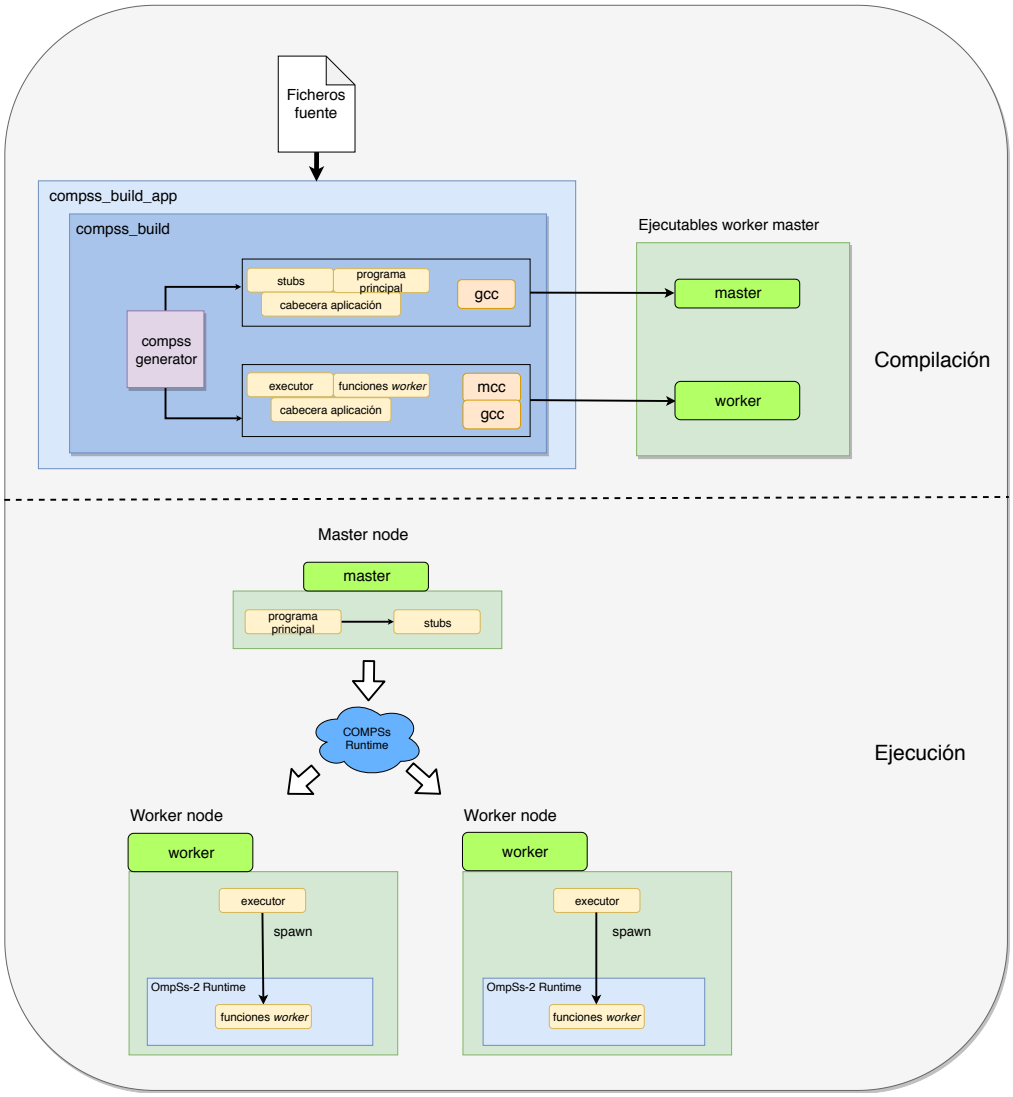

Figura 2.6: Fases de compilación y ejecución de una aplicación COMPSs+OmpSs-2.

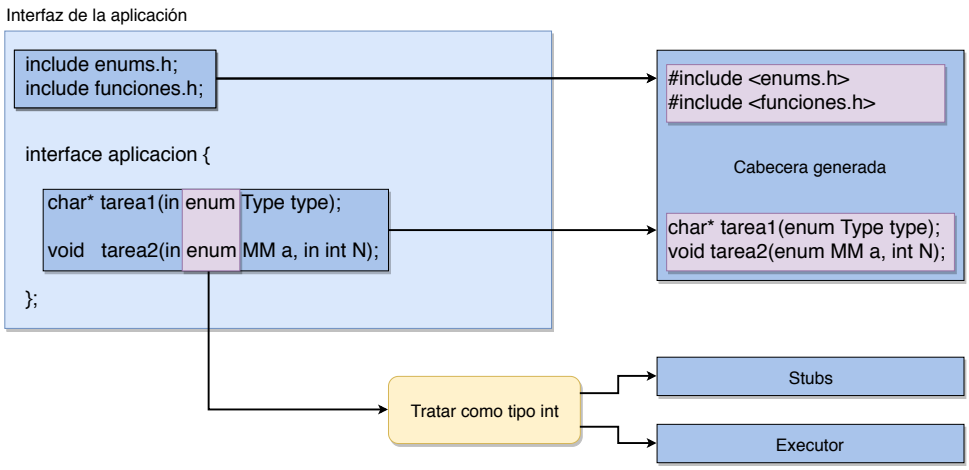

Figura 2.7: Uso del tipo enum e incluir cabeceras en la interfaz de una aplicación.

Capítulo 3

Evaluación

En este capítulo describiremos las aplicaciones que hemos desarrollado para evaluar la integración *COMPSs+OmpSs-2*. Las aplicaciones con las que evaluamos la integración son *K-Means* y *Cholesky*. Con *K-Means* hemos realizado un estudio de escalabilidad en el *cluster CTE-Power* y con *Cholesky* una comparativa entre *COMPSs* y *COMPSs+OmpSs-2* en el supercomputador *MareNostrum4*.

3.1. Aplicaciones

3.1.1. K-Means

K-Means es un algoritmo utilizado para hacer *clustering* sobre un grupo de datos, es decir, agrupar los datos en *clusters*. Se hace el cálculo de la distancia de cada dato a todos los *clusters* para determinar cuál es el más cercano y asignarlo a este. Una vez hecho esto se vuelve a calcular el centro de cada *cluster*. Todo este proceso conforma una iteración de *K-Means*, que se itera hasta que converge, esto es que la distancia entre los centros asignados en dos iteraciones consecutivas de cada *cluster* es cercana a 0.

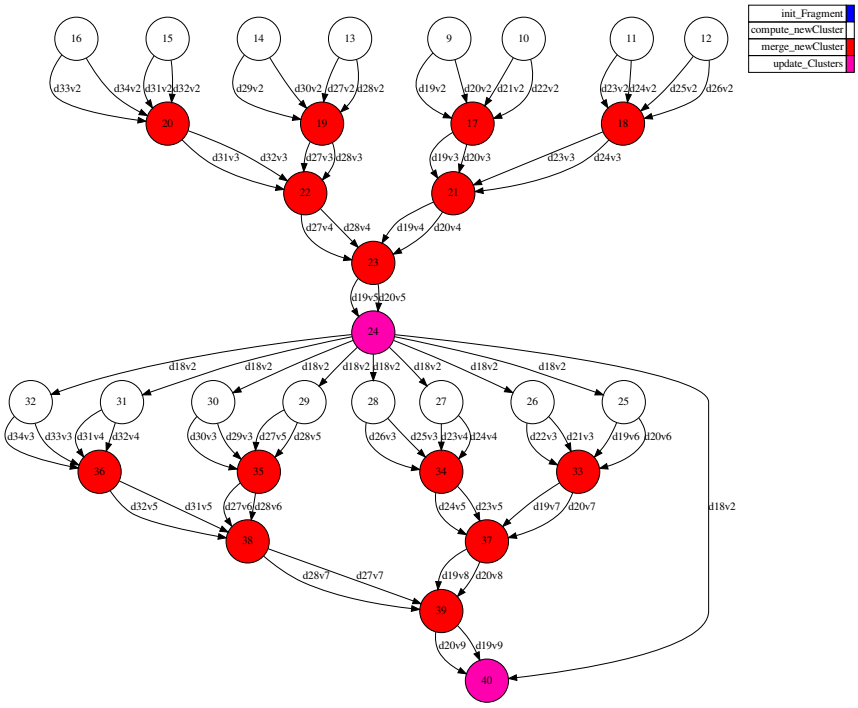

Figura 3.1: Grafo de dependencias entre tareas de K-Means.

Tal y como hemos desarrollado la aplicación tiene 4 tareas, que son *init_Fragment*, *compute_newCluster* (tiene una implementación en CPU y otra en GPU), *merge_newCluster* y *update_Clusters*. En la implementación se generan de manera aleatoria fragmentos, que son agrupaciones de los datos que debemos *clusterizar*. Con esta clasificación en fragmentos conseguimos que la granularidad de las tareas de *COMPSs* sea lo suficientemente gruesa. El cálculo de las distancias entre los datos de un fragmento y los *clusters* en la tarea *compute_newCluster*, está implementado con *OmpSs-2* tanto para la tarea de CPU como para la de GPU.

La imagen 3.1 muestra un grafo de dependencias de una ejecución de *K-Means* donde los datos están agrupados en 8 fragmentos, el número de *clusters* a formar es 50 y se efectúan dos iteraciones. Para no sobrecargar este apartado se incluye en el apéndice D.1 la interfaz y el programa principal de la aplicación.

3.1.2. Cholesky

Cholesky es un método de descomposición aplicable cuando la matriz es simétrica definida positiva, entonces esta puede ser descompuesta como el producto de una matriz triangular inferior y su traspuesta. El método *Cholesky* se utiliza para resolver sistemas de ecuaciones lineales, es similar a la descomposición LU y es el doble de eficiente.

La aplicación ha sido desarrollada con una visión de la matriz por bloques en dos niveles con tal de distribuirla entre los nodos de la máquina y poderla repartir entre los recursos dentro de un mismo nodo. La primera descomposición es efectuada por *COMPSs*, la segunda se efectúa de manera transparente a nosotros ya que la realiza una librería para operaciones de álgebra lineal programada en *C* con *OmpSs-2*. Esta librería es *LASs*[1] (*Linear Algebra routines on OmpSs*), en la implementación de cada función que utilizamos se realiza una partición de los bloques que provienen de *COMPSs* y se generan tareas de grano fino al nivel de *OmpSs-2*. Las funciones que implementa *LASs* serán utilizadas como tareas a nivel de *COMPSs* y después una vez ejecutadas harán uso de *OmpSs-2*, por lo que realmente desarrollar una aplicación utilizando librerías externas es muy sencillo.

Tiene 5 tareas, *generate_block*, *ddss_dpotrf*, *ddss_dtrsm*, *ddss_dgemm*, *ddss_dsyrk*, *generate_block* se utiliza para generar los bloques de manera distribuida (así el *master* no tiene reservar memoria para toda la matriz, que podría ser muy grande), las otras 4 tareas son operaciones de algebra lineal que utilizadas de la forma correcta aplicarán el método de descomposición de *Cholesky*. En el apéndice D.2 se encuentra la interfaz y el programa principal de la aplicación.

La imagen 3.2 muestra un grafo de dependencias de una ejecución de la aplicación *Cholesky* con una matriz de 64 bloques con 4096x4096 elementos cada uno.

3.2. Entornos para el estudio

Para la realización del estudio del rendimiento se han utilizado dos máquinas que el *Barcelona Supercomputing Center* pone a nuestra disposición, que son *CTE-Power* y *MareNostrum4*.

3.2.1. CTE-Power

CTE-Power es un *cluster* basado en la tecnología de procesadores *IBM Power9*. Tiene 2 nodos de *login*, son los nodos desde donde operan los usuarios y 52 nodos de cómputo.

Cada nodo tiene los siguientes componentes:

- 2 x *IBM Power9 8335-GTH* @ 2.4GHz (3.0GHz en modo turbo), cada procesador con 20 cores cada uno con 4 *threads*.
- 512GB de memoria organizada en 16 *dimms* de 32GB @ 2666MHz.
- 4 x *NVIDIA* V100 con 16GB *HBM2*.
- 2 x SSD que proporcionan 1.9TB de almacenaje local.

[1] https://pm.bsc.es/mathlibs/lass

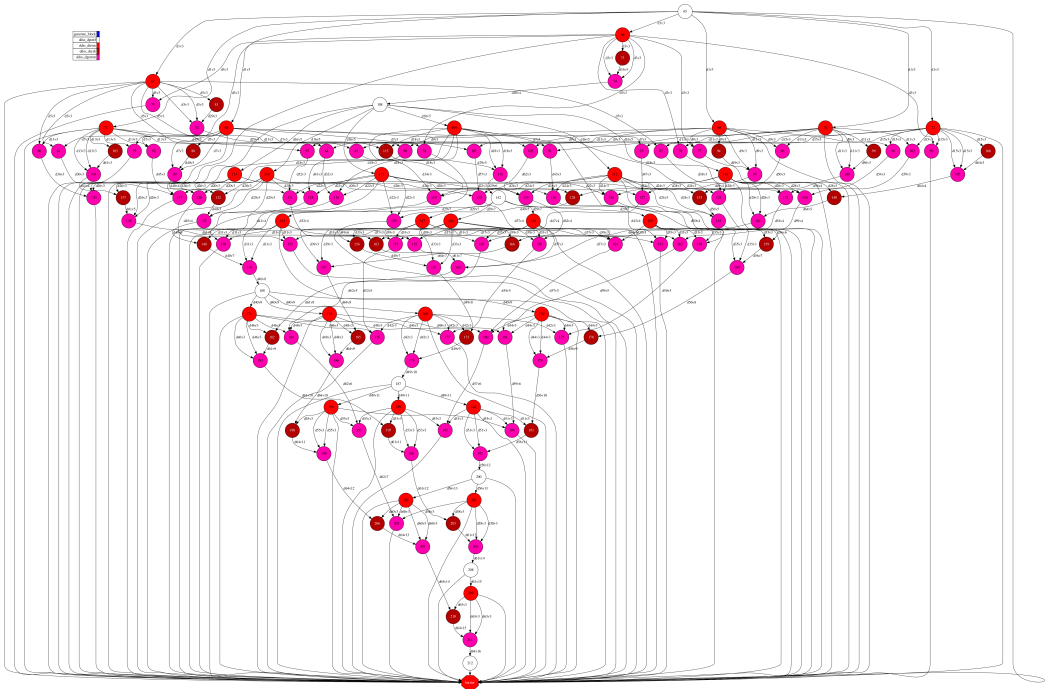

Figura 3.2: Grafo de dependencias entre tareas de Cholesky.

- 2 x *NVM Express* 3.2TB.
- Interfaz de red *Mellanox*.
- Sistema de ficheros GPFS a través de fibra a 10GBit.

Nosotros utilizaremos tan sólo 10 nodos, por lo que tendremos a nuestra disposición en la mayor ejecución 1600 CPUs y 40 GPUs *NVIDIA V100*.

3.2.2. MareNostrum4

MareNostrum4 es un supercomputador basado en la tecnología de procesadores *Intel Xeon Platinum* de la generación *Skylake*. Tiene 5 nodos de *login* y 3.456 nodos de cómputo.

Cada nodo tiene los siguientes componentes:

- 2 x *Intel Xeon Platinum 8160 @ 2.10GHz* con 24 cores.
- Hay nodos con distintos tipos de memoria:
  - 1,88 GB/core de memoria.
  - 7,93 GB/core de memoria.
- Un SSD que proporciona 200 GB de almacenaje local.
- 100 Gbit/s *Intel Omni-Path HFI Silicon 100 Series PCI-E adapter*.
- 10 Gbit *Ethernet*.

Igual que con *CTE-Power* tan sólo utilizaremos 10 nodos, por lo tanto tendremos en la mayor ejecución 480 CPUs.

3.3. K-Means

La evaluación con *K-Means* ha consistido en hacer un test de *strong scaling*, que consiste en ejecutar la aplicación con un tamaño de problema fijo y aumentar el número de recursos de cómputo, y otro de *weak scaling* donde aumentaremos de manera proporcional el tamaño del problema y los recursos de cómputo. En el apartado 3.2.1 describimos la máquina que utilizamos en estos test, haremos uso de todas las *CPUs* y *GPUs* de cada nodo.

3.3.1. Strong scaling

La imagen 3.3 muestra una gráfica que tiene en el eje vertical el tiempo y en el horizontal el número de nodos. El tamaño del problema que se ha utilizado es de 400 fragmentos de 50 dimensiones y 200000 puntos cada uno, el número de *clusters* a formar es 50 y se efectúan 5 iteraciones. Se han tomado 10 muestras de cada ejecución, la línea azul muestra la media de las muestras para el mismo número de nodos, la de color naranja muestra el tiempo ideal que debería tardar según el número de nodos. En cada punto de la línea azul se muestra la diferencia entre el máximo valor y el mínimo que se han tomado en las muestras.

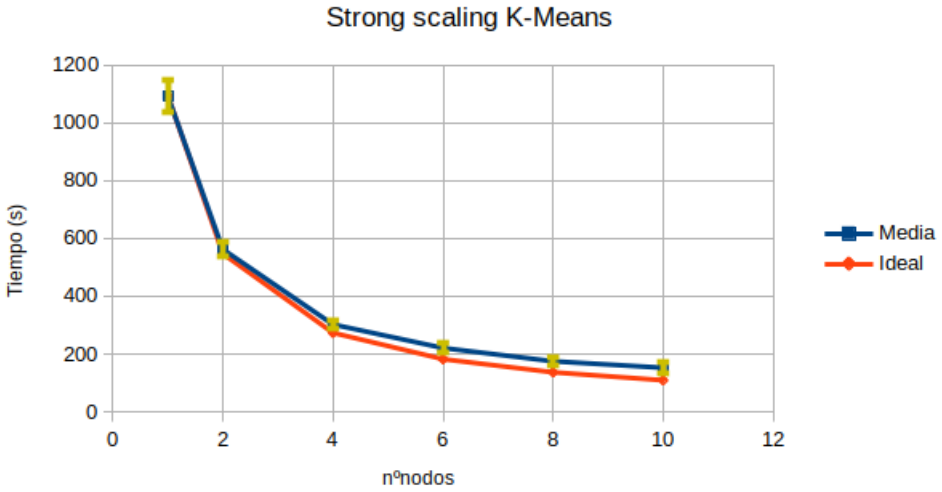

Figura 3.3: Tiempo de ejecución al aumentar el número de recursos.

Es bueno que la línea azul sea lo más parecida a la línea naranja, eso nos indicaría que la aplicación escala perfectamente, ya que cada vez que aumentamos los recursos el tiempo de ejecución se reduciría proporcionalmente. La imagen 3.4 muestra la misma información que 3.3 dispuesta en forma de ganancia, se puede ver que se comporta bien hasta que en los 4 nodos empieza a dejar de acercarse al ideal.

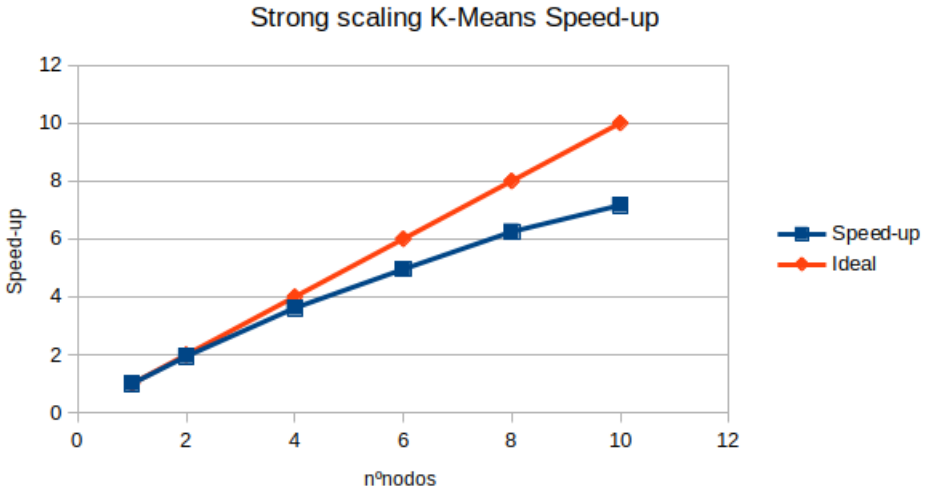

Figura 3.4: Ganancia al aumentar el número de recursos.

Esta disminución de la ganancia cobra sentido cuando en el grafo de la aplicación 3.1 vemos que en la fase de computar los *clusters* nuevos (tareas de color blanco) la aplicación es totalmente paralela pero en la fase de re-asignar los centros de los *clusters* (tareas de color rojo) deja de serlo.

Esta característica de la aplicación hace que nunca podamos alcanzar la ganancia ideal ya que hay tramos o fases de esta que no son completamente paralelos, aunque estemos utilizando todos los recursos que hay a nuestra disposición tenemos que esperar a ejecutar un nivel de re-asignación antes de empezar el siguiente.

3.3.2. Weak scaling

En un test de *weak scaling*, esperamos que los tiempos de ejecución sean siempre los mismos, ya que el tamaño del problema es siempre proporcional a los recursos de cómputo. Esto sería perfecto pero hay que efectuar comunicación entre nodos y gestionar todo el sistema, por lo que no podemos mantener este escenario ideal. Se han tomado 10 muestras de cada ejecución, línea azul marca el valor ideal (que toma este valor de la primera ejecución), y la naranja la eficiencia respecto al valor ideal. Los valores utilizados para calcular la eficiencia y el valor ideal se han hecho con la media de las 10 ejecuciones. La imagen 3.5 muestra cómo al aumentar el número de nodos la eficiencia baja del ideal (que el tiempo de ejecución sea siempre el mismo), pero se mantiene alrededor del mismo valor a partir de los 6 nodos.

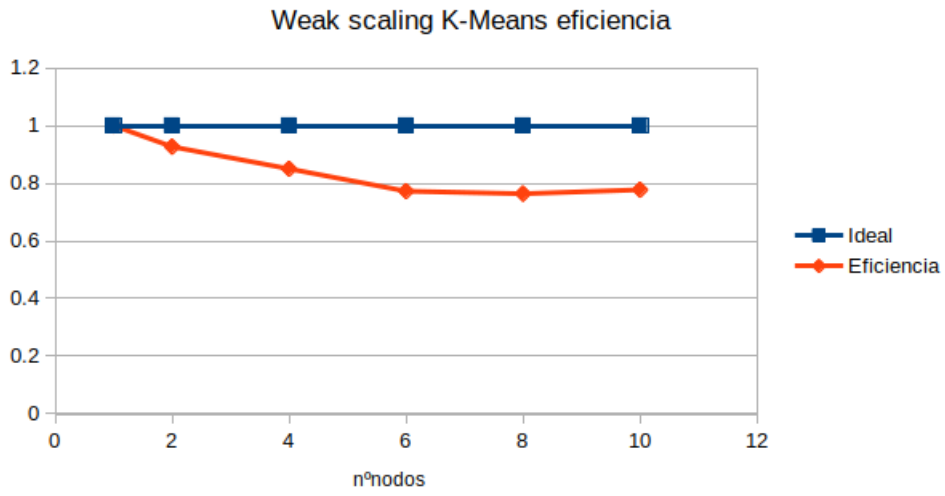

Figura 3.5: Eficiencia al mantener proporcional el tamaño del problema y el número de recursos.

3.4. Cholesky

Para evaluar la integración con la aplicación *Cholesky* hemos decidido comparar la integración *COMPSs+OmpSs-2* con *COMPSs* puro, en el primer test de la evaluación hemos variado el número de bloques y el tamaño de estos bloques. La evaluación con esta aplicación se lleva acabo en un nodo de *MareNostrum4* que se han descrito en la sección 3.2.2.

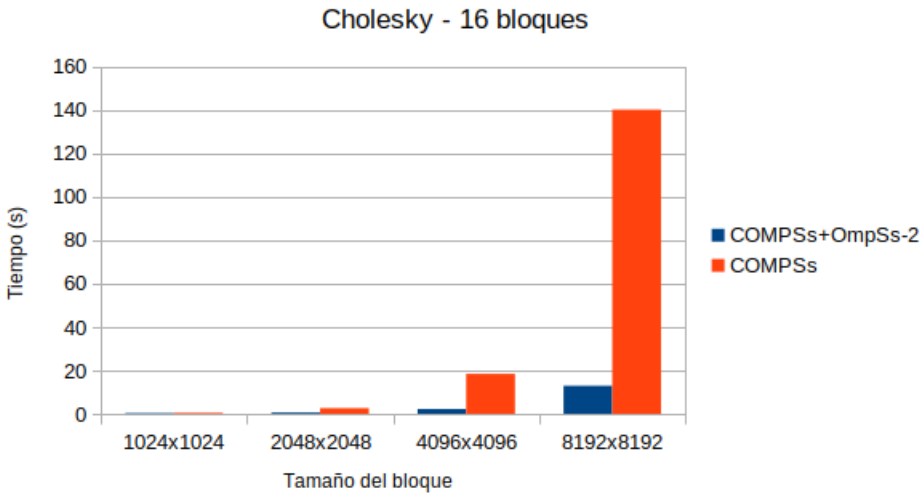

Figura 3.6: Tiempo de ejecución de Cholesky con 16 bloques.

La imagen 3.6 muestra el tiempo de ejecución de la versión *COMPSs+OmpSs-2* y *COMPSs* con 16 bloques, los tamaños de bloque aparecen en el eje horizontal. A medida que se aumenta el tamaño de cada bloque la librería *LASs* tiene más margen para partir los bloques y generar tareas, pero la versión *COMPSs* ejecuta todo absolutamente en secuencial, haciendo que cada vez que la ganancia sea más notoria. En la imagen 3.7 se muestra la ganancia de *COMPSs+OmpSs-2* respecto la versión de *COMPSs*.

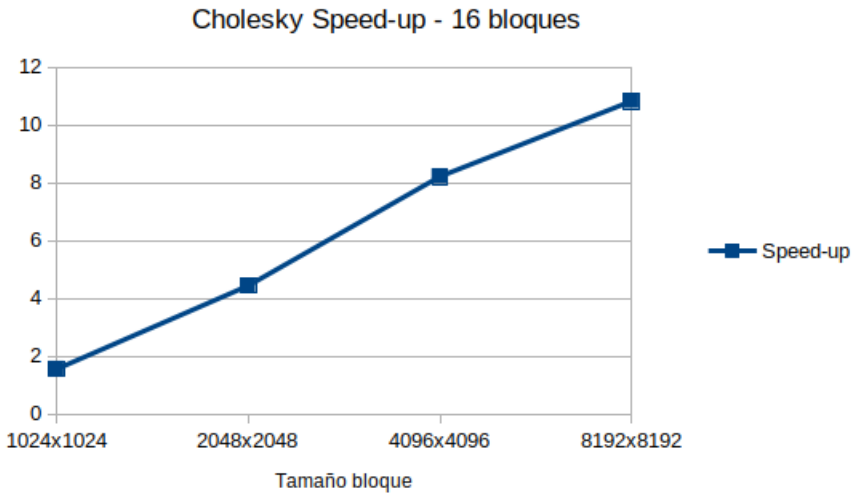

Figura 3.7: Ganancia de Cholesky con COMPSs+OmpSs-2 respecto la versión COMPSs.

A pesar de que en las imágenes 3.6 y 3.7 se muestra una diferencia grande entre la versión con y sin *OmpSs-2*, esto es así por que el número de tareas que *COMPSs* puede generar y están libres de dependencias y por tanto puede ejecutar a la vez no es superior o igual al número de procesadores que hay en un nodo, por este motivo no estamos aprovechando todo el nodo. En cambio en la versión con *OmpSs-2* partimos los bloques y generamos tareas que los procesen, estas utilizan los procesadores que en cualquier otro caso estarían inactivos.

Tal y como hemos visto, las gráficas anteriores no eran justas para *COMPSs* ya que no permitían utilizar todos los procesadores del nodo. El siguiente test pretende poner ambas versiones en un escenario justo, el número de bloques se aumenta a 64 y los tamaños de bloque se mantienen como en el anterior test.

La imagen 3.8 muestra los tiempos de ejecución de ambas versiones, por el mismo motivo que antes al aumentar el tamaño de bloque las diferencias entre ambas se vuelven más notorias.

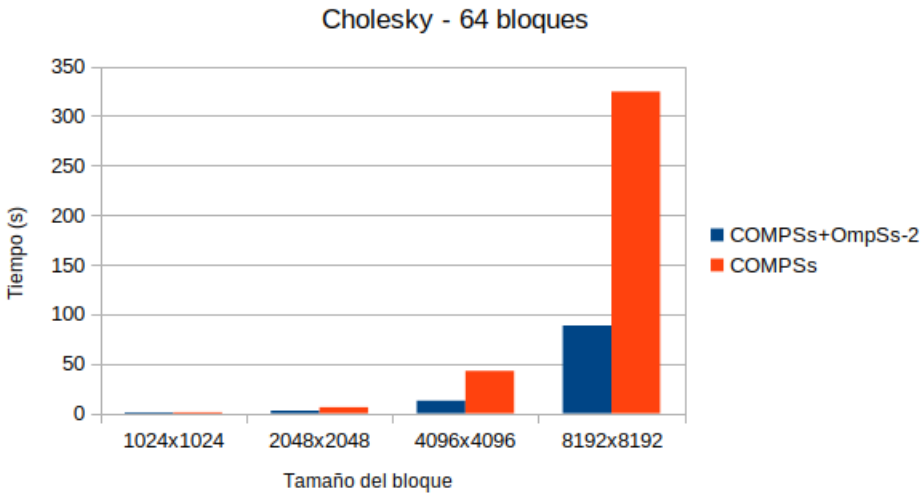

Figura 3.8: Tiempo de ejecución de Cholesky con 64 bloques.

Es en la ganancia de la imagen 3.9 donde se puede apreciar que la comparación es más justa que la anterior, ahora como máximo obtenemos una ganancia alrededor de 3.5, antes estaba cerca de 10. Ha sido el aumento del número de bloques lo que ha permitido que la versión de *COMPSs* aproveche todo un nodo. Aún así, si vemos la imagen 3.2 se puede ver como en la aplicación el número de tareas disminuye a medida que se acaba la aplicación, por lo que nos volvemos a encontrar en el mismo escenario anterior, en el cual no aprovechábamos todos los recursos del nodo. *COMPSs+OmpSs-2* vuelve a sacar provecho de este escenario, puesto que aunque haya menos tareas de granularidad gruesa, estas generan de granularidad finas al nivel de *OmpSs-2* con lo cual se puede seguir aprovechando los recursos.

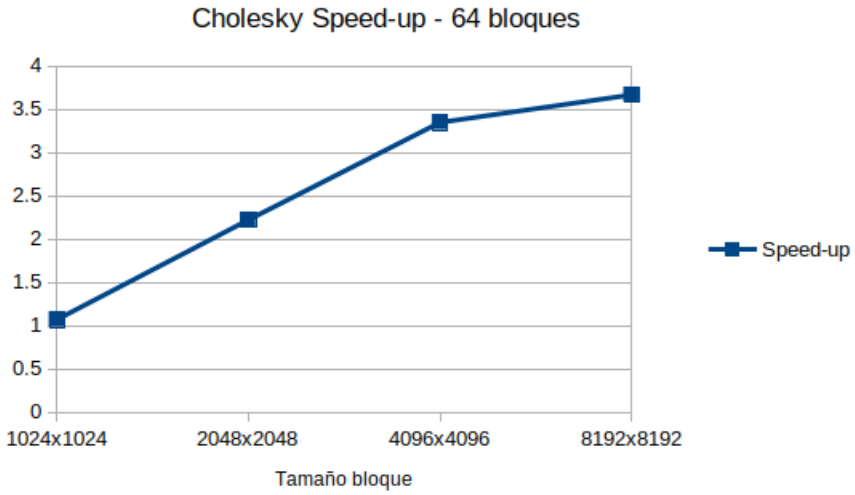

Figura 3.9: Ganancia de Cholesky con COMPSs+OmpSs-2 respecto la versión COMPSs.

Capítulo 4

Limitaciones de la integración COMPSs+OmpSs-2

En la sección 1.5 se enunciaron los problemas que tuvimos con la integración inicial *COMPSs+OmpSs*. En este apartado, se enuncian las limitaciones de la integración *COMPSs+OmpSs-2*. En la anterior encontrábamos que las tareas de *OmpSs* generadas dentro de una tarea de *COMPSs* no tenían ningún tipo de aislamiento para el *worker persistente*, y cuando se efectuaban esperas esto repercutía a absolutamente todas las tareas de *OmpSs*. Utilizando el modo librería de *OmpSs-2* proporcionamos a las tareas generadas dentro de una tarea de *COMPSs* un ámbito propio, por lo cual las esperas tan sólo afectaran a las generadas dentro de esta, y no al resto.

Aún así, hemos encontrado limitaciones en la integración, para localizarlas hemos desarrollado una aplicación con dos tareas de diferente granularidad, con *Extrae* y *Paraver* veremos como se comporta la integración ante este caso. A grandes rasgos, la aplicación básicamente ejecuta 30 tareas de granularidad fina y gruesa, ésta se emula haciendo una espera bloqueante con *usleep* de 50000 y 300000 microsegundos respectivamente. La implementación se encuentra en el apéndice E.

Cuándo el número de CPUs no es suficiente para la cantidad de tareas que *OmpsS-2* crea, es propenso a la entremezcla.La planificación de las tareas no tiene por qué seguir ningún tipo de orden, la tarea que se ejecutará después no tiene por qué ser sucesora de esta o compartir predecesor, no hay ningún tipo de vínculo, se ejecuta lo que se cree mejor para explotar el paralelismo. Entonces una CPU que para nosotros tenía la expectativa de ejecutar tres tareas puede acabar ejecutando una mayor cantidad e incluso de una duración diferente. Este entremezclado puede derivar en una duración irregular de las tareas a nivel de *COMPSs*, la siguiente imagen muestra una traza de *COMPSs* y de *OmpSs-2*.Entonces, si ejecutamos esta aplicación con un número de CPUs bajo y obtenemos una traza, veremos que las tareas se entremezclan, haciendo que una tarea que inicialmente debía tardar del orden de 50000 microsegundos acabe tardando 6 veces más.

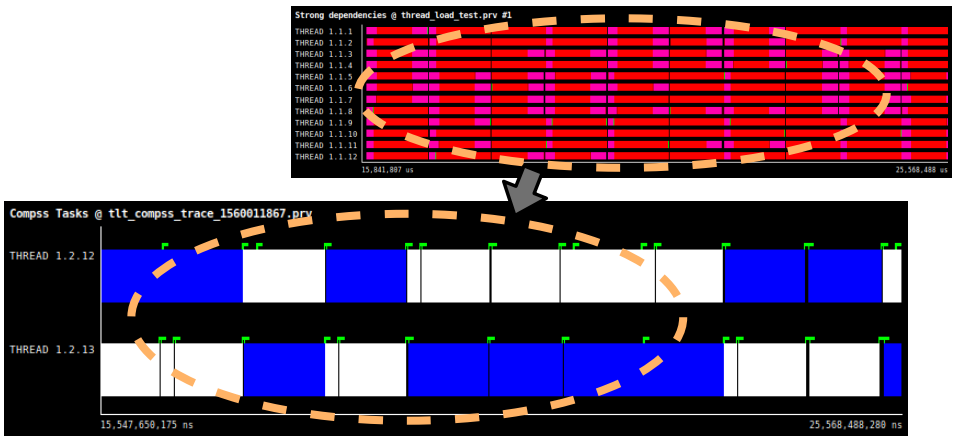

Figura 4.1: Duración irregular de las tareas de *COMPSs*.

La imagen 4.1 muestra el comportamiento descrito, la traza superior corresponde a *OmpSs-2*, es una parte de la aplicación, se puede apreciar el entremezclado de las tareas de granularidad fina y gruesa (rosa y rojo respectivamente), en la traza inferior de *COMPSs* se observa que el entremezclado provoca que la duración de las tareas no sea uniforme, en algunos casos las de color blanco que son de granularidad fina tardan lo mismo o incluso más que las de color azul que son de granularidad gruesa. Recordemos que dentro de un mismo nodo, se comparte el *runtime* de *OmpSs-2* por lo cuál las tareas de *COMPSs* que se ejecutan en un mismo nodo están sujetas a *OmpSs-2* y no tienen entornos aislados entre sí. El hecho de que las tareas de *COMPSs* tarden más o menos, es así por que *OmpSs-2* constantemente ejecuta tareas en las CPUs que le parece mejor para explotar el paralelismo dentro del nodo, haciendo que una tarea de *COMPSs* que podría ejecutar tan sólo lo que a ella le concierne ejecute también trabajo del resto, prolongando su duración (lo que vimos en la traza 4.1).

Si aumentamos el número de CPUs, este comportamiento no debería repercutir de una manera tan drástica, con el aumento de recursos la duración de las tareas será más uniforme.

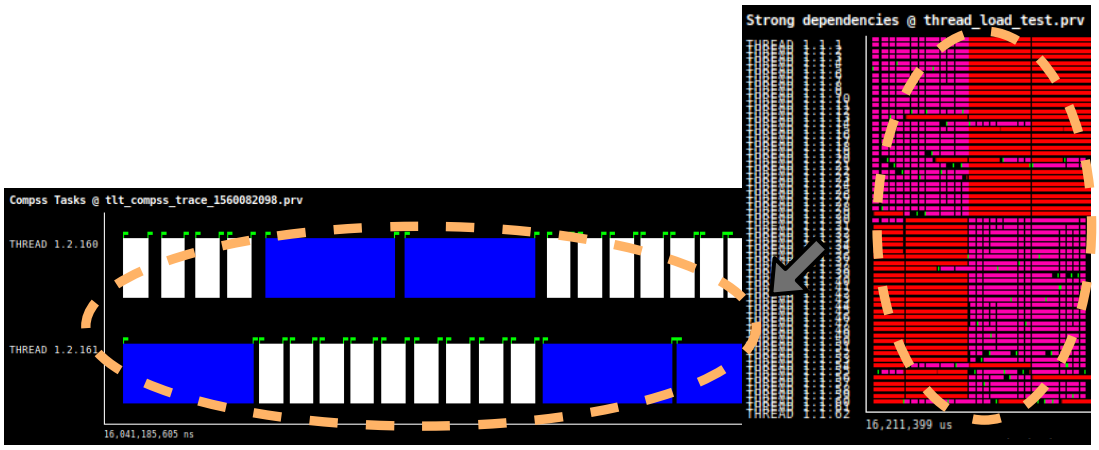

Figura 4.2: Duración regular de las tareas de *COMPSs*.

En la ejecución de la imagen 4.2 hay muchos más recursos, y como hemos anticipado esto hace que la duración de las tareas sea mucho más regular y por norma general no haya anomalías. Si la comparamos con la traza 4.1, la duración de las tareas es mucho más regular, siempre son de la misma magnitud, y nunca una tarea de granularidad fina dura más que una gruesa. Esto se debe a que tenemos más recursos, por lo cuál no aparece el entremezclado de tareas y no se nos alargan las tareas de *COMPSs*. Hay una diferenciación mucho más clara entre los dos tipos de tareas, y por lo general no existen estos conflictos por recursos que comentábamos. De todas formas, el comportamiento negativo que se ha enunciado con el primer par de trazas puede suceder eventualmente, ya que la manera en la que se gestionan los recursos no ha cambiado, pero mejora mucho.

Para eliminar esta limitación, habría que otorgar a cada tarea generada a través de la función *nanos6_spawn_function* un subconjunto de los recursos o organizar la planificación con prioridades descendientes entre tareas consecutivas generadas vía *spawn*, con la primera opción las tareas de *OmpSs-2* generadas en una tarea de *COMPSs* no harían ni sufrirían interferencias ya que tienen recursos propios asignados, y con la segunda opción se daría prioridad a la ejecución de tareas en el orden original de generación, por lo que tampoco habría interferencias con ni sobre el resto.

Capítulo 5

Conclusiones

A lo largo de este proyecto hemos podido ver todo lo relacionado con realizar una integración entre dos modelos de programación, seguramente cada integración sea un mundo, pero con este proyecto hemos aprendido a realizar una y seguramente todo esto sea aplicable en otra integración. En concreto se ha visto cómo atacar el problema, conocer los dos modelos lo suficiente como para saber cómo se tiene que realizar la integración y averiguar exactamente qué papel desarrolla cada uno en esta integración. Además se ha aprendido a desarrollar aplicaciones que hagan uso de la integración *COMPSs+OmpSs-2*, que ha requerido entender y saber utilizar los modelos de programación *COMPSs* y *OmpSs-2*. También hemos realizado una evaluación del rendimiento, conociendo dónde están los límites que ponen las aplicaciones y dónde los que pone la integración.

Se ha iniciado este proyecto con tal de brindar facilidad para el uso de las plataformas distribuidas heterogéneas, y se ha conseguido. Todo el esfuerzo puesto sobre la integración nos ha permitido conseguir gestionar estas plataformas de una manera muy sencilla, no sólo facilita la gestión si no que permite mejorar el rendimiento de las aplicaciones para estas plataformas, tal y como se demuestra en la sección 3. Las modificaciones que se han realizado sobre *COMPSs* para realizar la integración han intentado ser mínimas y lo más claras posibles, así cualquier persona que tenga que trabajar con el código pueda ser partícipe sin que sea necesariamente un quebradero de cabeza.

Esperamos que esta facilidad que hemos introducido motive a otros investigadores a utilizar nuestro modelo de programación *COMPSs* conjuntamente con *OmpSs-2* e incluso nos lleve a colaborar con ellos, ya que somos los más indicados, tenemos conocimiento pleno acerca de la integración.

Como trabajo futuro, sería fantástico integrar también el modelo *OmpSs-2* en *Java* y *Python*, de esta manera lenguajes interpretados podrían gozar de la velocidad de un lenguaje compilado como es *C* y del paralelismo que *OmpSs-2* nos brinda. Sería también interesante comparar la eficiencia energética de aplicaciones desarrolladas para *COMPSs+OmpSs-2* y *COMPSs*, así veríamos que hacer uso de la variedad de recursos de los entornos que utilizamos hoy en día no vale la pena tan sólo en términos de rendimiento si no también en términos de eficiencia energética.

A nivel personal el proyecto ha complementado la mayoría de conocimientos adquiridos durante el grado, se ha mucho sobre modelos de programación para sistemas distribuidos (*COMPSs*) y paralelos (*OmpSs-2*), se ha tenido que utilizar *Extrae* y *Paraver* a un nivel más avanzado y también se ha aprendido cómo modificar un compilador hecho con *Yacc* y *Lex*. Además me ha permitido iniciarme en el mundo de la investigación gracias a la confianza de mi director y co-director.

Capítulo 6

Informe de sostenibilidad

6.1. Dimensión ambiental

- **¿Has cuantificado el impacto ambiental de la realización del proyecto? ¿Qué medidas has tomado para reducir el impacto? ¿Has cuantificado esta reducción?**

  Por desgracia no ha habido tiempo para cuantificar el impacto ambiental del proyecto. Se podría haber estimado el consumo de energía de *CTE-Power* y *MareNostrum4* al ejecutar los experimentos, en caso de haberlo estimado, las únicas medidas posibles para reducir el impacto hubieran sido minimizar el uso de estas máquinas, usarlas solo cuando estuviera completamente seguro que el experimento iba a funcionar, de esta manera el consumo se reduciría drásticamente a utilizar todos los recursos desde 1 a 10 nodos en ambas máquinas.

- **¿Si hicieras de nuevo el proyecto, ¿Podrías realizarlo con menos recursos?**

  Desde luego podría haber invertido menos tiempo conociendo de antemano qué me depara en cada fase del proyecto, pero los recursos utilizados son estrictamente los necesarios, no falta ni sobra ninguno.

- **¿Qué recursos estimas que se usarán durante la vida útil del proyecto? ¿Cuál será el impacto ambiental de estos recursos?**

  Es difícil conocerlo, durante la vida útil del proyecto puede ser utilizado en clústers y supercomputadores de manera que los recursos no son pocos, pero el proyecto no hace uso de recursos por si solo, siempre será en el contexto de otro proyecto o investigación científica. Quizá el único recurso estricto es mantener las versiones del proyecto en la nube con controles de versión como *GitHub* o *GitLab*, donde el impacto ambiental viene derivado del consumo de sus servidores.

- **¿El proyecto permitirá reducir el uso de otros recursos? ¿Globalmente, el uso del proyecto mejorará o empeorará la huella ecológica?**

  El proyecto pretende facilitar la programación en entornos distribuidos heterogéneos, por lo que el tiempo que se utilizarán estos entornos se verá reducido, por lo cual estaremos ayudando a reducir los recursos de otros proyectos. Definitivamente, el uso del proyecto mejorará la huella ecológica.

- **¿Podrían producirse escenarios que hiciesen aumentar la huella ecológica del proyecto?**

  La huella ecológica del proyecto durante el desarrollo podría verse afectada en el caso que hubiera un desvío notorio en las fases de experimentación donde hay que utilizar máquinas que suponen un consumo energético elevado. *A posteriori* no puede ser que se aumente la huella ecológica intrínseca del proyecto, la posible huella viene dada por el uso que los usuarios hagan del proyecto.

6.2. Dimensión económica

- **¿Has cuantificado el coste (recursos humanos y materiales) de la realización del proyecto? ¿Qué decisiones has tomado para reducir el coste? ¿ Has cuantificado este ahorro?**

  Se ha cuantificado el coste de recursos humanos y materiales durante la realización del proyecto, para reducir el coste de los recursos humanos se ha intentado hacer las reuniones con el director y co-director las veces justas, minimizar el uso de recursos humanos solo cuando sea estrictamente necesario, para los recursos materiales se ha intentado utilizar una porción de las máquinas asequible y se ha intentado minimizar el uso de estas. Estas medidas se han tomado desde el primer momento por lo que el presupuesto ya se ajusta a ellas, no ha habido ningún ahorro al respecto.

- **¿Se ha ajustado el coste previsto al coste final? ¿Has justificado las diferencias (lecciones aprendidas)?**

  El coste se ha visto afectado en el hecho que tuvimos que descartar el uso del *MinoTauro* por que las tarjetas gráficas que utilizan son incompatibles con los *drivers* más nuevos de *NVIDIA* por lo que *OmpSs-2* no las soporta. Para reemplazar la máquina decidimos utilizar *MareNostrum4*, lo que nos llevo a aumentar el coste final. Pero el coste previsto del proyecto aparte de la desviación que comentada no ha sufrido ningún desajuste, se ajusta completamente.

- **¿Qué coste estimas que tendrá el proyecto durante su vida útil? ¿ Se podría reducir este coste para hacerlo más viable?**

  El proyecto no tiene ningún coste una vez desarrollado, si hubiera uno sería coste de infraestructura para mantenerlo en la nube de *GitHub* o *GitLab* pero es un servicio gratuito para instituciones de carácter público. El coste no se puede reducir más.

- **¿Se ha tenido en cuenta el coste de los ajustes/actualizaciones/reparaciones durante la vida útil del proyecto?**

  No, dado que es una pieza de *software* que integra dos modelos de programación no debería necesitar de actualizaciones durante su vida útil que no vengan hechas por los propios equipos de desarrollo de *COMPSs* o *OmpSs-2*, aún así, si hiciera falta se añadió el porcentaje de contingencia para poder encarar situaciones similares.

- **¿Podrían producirse escenarios que perjudicasen la viabilidad del proyecto?**

  El proyecto no pretende lucrarse por lo que realmente la viabilidad no podría verse afectada. La integración *COMPSs+OmpSs-2* pretende ser utilizada en el marco de investigaciones científicas.

6.3. Dimensión social

- **¿La realización de este proyecto ha implicado reflexiones significativas a nivel personal, profesional o ético de las personas que han intervenido?**

  Por suerte el proyecto no ha llevado a ninguna persona implicada a tener que reflexionar de esta manera. Las reflexiones realizadas han sido sólo en el marco definido única y exclusivamente por el proyecto.

- **¿Quién se beneficiará del uso del proyecto?**

  Todo usuario que lo utilice, principalmente destinado a personal de investigación que quiera programar en entornos distribuidos heterogéneos de una manera más sencilla y efectiva.

- **¿Hay algún colectivo que puede verse perjudicado por el proyecto? ¿ En qué medida?**

  Desde luego que no, el proyecto no puede afectar de ninguna manera a ningún colectivo, podría afectar a nivel competitivo pero no tiene como objetivo ser vendido en el mercado y cuenta con licencias que permiten su uso, modificación y distribución de manera gratuita.

- **¿En qué medida soluciona el proyecto el problema planteado inicialmente?**

  El proyecto soluciona el problema tal y como estaba planeado desde que se inició la gestión del proyecto.

- **¿Podrían producirse escenarios que hiciesen que el proyecto fuese perjudicial para algún segmento particular de la población?**

  El proyecto por si mismo no supone ninguna amenaza para ningún segmento de la población, solo el uso que le den los usuarios podría ser perjudicial. Pero no creo que seamos responsables del uso que hagan los usuarios cuando brindamos una herramienta que únicamente pretende hacer más fácil la programación de entornos distribuidos heterogéneos.

- **¿Podría crear el proyecto algún tipo de dependencia que dejase a los usuarios en posición de debilidad?**

  De ninguna manera. El proyecto no incluye características que creen dependencia entre este y el usuario, en caso de que alguien desarrollase este tipo de dependencia sería un caso muy particular a tratar.

Apéndices

Apéndices A

Planificación temporal

Este apéndice presenta cómo planificamos los cuatro meses de duración que tiene el proyecto, desde Febrero de 2019 hasta Junio de 2019 y cómo han sucedido realmente. Se especificarán las tareas a realizar junto a su durada aproximada y la duración que creemos que han tenido finalmente, también se han tenido en cuenta las posibles desviaciones en la realización de estas.

A.1. Especificación de las tareas

Detallamos a continuación las tareas a realizar.

A.1.1. GEP - Gestión de proyectos

La primera tarea del proyecto es la gestión de este, se han elaborado cuatro entregables que sintetizan la temática del proyecto, los objetivos, como se realizaría cuando se planificó y como se ha realizado, la metodología que se ha seguido, las actividades que se han realizado, y un estudio de sostenibilidad.

- **Elaboración del primer entregable:** En este primer apartado se un contexto, el estado del arte del proyecto, los objetivos, requerimientos, riesgos y una metodología para desarrollar el proyecto en sí. La duración aproximada es de unas 24 horas.
- **Elaboración del segundo entregable:** En este apartado se definen las actividades y duración de estas. La duración aproximada es de 6 horas.
- **Elaboración del tercer entregable:** En este apartado se ha realizado la auto-evaluación sobre la sostenibilidad además de un análisis sobre la gestión económica y la sostenibilidad del proyecto. La duración aproximada es de 18 horas.
- **Elaboración del cuarto entregable:** En este último apartado se ha preparado una presentación oral y se ha confeccionado el documento final, que serán los tres anteriores revisados y corregidos con la orientación del *feedback* del profesor de , director y co-director. La duración aproximada es de 12 horas.

Con estos cuatro apartados se finaliza el primer bloque de tareas. En princpio, la duración estipulada del GEP es de 75 horas, la duración final ha sido de 60 horas por lo que tuvimos tiempo para revisar y repasar estos apartados.

Para realizar esta actividad, se ha utilizado un ordenador, *GitHub* para subir la documentación, *Kile* para redactar el documento en *LaTeX*,*Trello* para organizar las actividades en forma de tarjetas, *Gantter* para elaborar el diagrama de *Gantt* y *Google Drive*.

A.1.2. Uso de la API de Nanos6

Para poder llevar acabo exitosamente la integración, se necesita entender qué hace y saber utilizar la *API* de *Nanos6*. Requiere mirar documentación e interactuar con los desarrolladores de *Nanos6*.

Hemos aprendido a utilizar la llamada *nanos6_spawn_function*, que nos permite ejecutar una función como tarea. Para poder utilizarla, necesitamos levantar el *runtime* de manera manual en un programa compilado con *gcc* (*GNU C Compiler*) o bien *g++* cuando se utilice *C++*, y efectuar la llamada a una función externa compilada con *mcc* (*Mercurium*), ya que estará anotada con pragmas de *OmpSs-2*.

La duración de esta tarea ha sido de 60 horas. Los recursos necesarios son un ordenador con un compilador nativo de *C*, otro de *C++*, y *Mercurium* y *Nanos6* instalados.

A.1.3. Integrar OmpSs-2 en el binding de C/C++

La tarea principal que da sentido al proyecto es esta, comprende el estudio y la integración de *OmpSs-2* en el *binding* de *C/C++*. Requiere del estudio de la estructura interna de *COMPSs* por una banda y del *binding* por otra, hemos tenido que decidir dónde se puede inicializar el *runtime* de *Nanos6* y cuándo se debe apagar. También se necesita hacer la llamada a la *API* en el *worker*, cosa que ha habido que estudiar dónde situar dentro del código.

El primer paso ha consistido en analizar dónde tiene más sentido que hagamos la gestión del *runtime* de *Nanos6*, y cómo hacerla. Para se ha repasado el código de *COMPSs* y se ha determinado qué hace cada componente de este.

En segundo lugar se ha implementado toda la gestión. Claro que también ha habido que efectuar la llamada a la *API* y verificar el funcionamiento de la integración, ha sido lo que más tiempo nos ha llevado sin duda alguna.

Esta tarea ha sido la que más tiempo nos ha llevado, pero nos ha conocimiento pleno de cómo funciona el modo librería y el *binding* de*C/C++*. La duración ha estado alrededor de 96 horas. Los recursos necesarios han sido un ordenador con *COMPSs* instalado, un compilador nativo de *C*, otro de *C++*, y *Mercurium* y *Nanos6* instalados.

A.1.4. Estudiar la integración de OmpSs-2 en Java y binding de Python

Finalmente esta tarea se descartó, fue preferible invertir tiempo en el resto de funcionalidades y evaluar la integración de la mejor manera posible. Aún así, se invirtió tiempo en sopesar posibilidades para realizar tanto la integración de *Java* como la de *Python*, por lo que la duración ha sido alrededor de 15 horas.

A.1.5. Desarrollo de una aplicación que use COMPSs+OmpSs-2

En esta tarea se han desarrollado dos aplicaciones, *K-Means* y *Cholesky* utilizando *COMPSs+OmpSs-2*, se han utilizado para evaluar el rendimiento en otra tarea. Hemos empleado aproximadamente 84 horas en desarrollar ambas aplicaciones y testear que funcionasen. Los recursos necesarios son un ordenador con *COMPSs* instalado, un compilador nativo de *C*, otro de *C++*, y *Mercurium* y *Nanos6* instalados

A.1.6. Estudio del rendimiento

Utilizando las aplicaciones desarrolladas en la tarea anterior hemos estudiado el rendimiento de la integración. Se ha hecho uso de opciones de *COMPSs* para medir cuánto tarda cada tarea enviada a un nodo.

Estudiar el rendimiento ha incluido intentar optimizar al máximo todas las pérdidas de rendimiento en la medida de lo posible, por lo cual el tiempo aproximado para llevarla a cabo, ha sido de 84 horas. Los recursos necesarios son un ordenador con *COMPSs* instalado, un compilador nativo de *C*, otro de *C++*, y *Mercurium* y *Nanos6* instalados.

A.1.7. Limitaciones de COMPSs+OmpSs-2

Dado que este proyecto parte de la premisa de resolver unos problemas concretos con el rendimiento, se efectúa un estudio muy concreto hacia estos problemas para ver si están resueltos o no. Que este estudio vaya mal o no, no afecta realmente al proyecto, ya que se intentará ver si *OmpSs-2* mejora respecto a *OmpSs* de todas formas.

A.1.8. Redactar la memoria

Por último se ha redactado la memoria del proyecto además de preparar todo el material audiovisual para la defensa de este. La duración de esta actividad ha sido de unas 72 horas. Los recursos que utilizados son *Kile* para redactar el documento en *LaTeX*, *GitHub* para guardar la documentación y *LibreOffice* para el apoyo audiovisual que se utilizará durante la defensa.

A.2. Dependencias

La siguiente tabla define la relación de dependencia entre las tareas que conciernen a la gestión del proyecto.

Tarea dependiente	Tarea predecesora
Contexto	-
Estado del arte	Contexto
Objetivos, requerimentos, riesgos	Estado del arte
Metodología	Objetivos, requerimentos, riesgos
Definir actividades	Metodología
Estimar tiempos	Definir actividades
Autoevaluación sobre la sostenibilidad	Estimar tiempos
Análisis del proyecto	Autoevaluación sobre la sostenibilidad
Confeccionar documento final	Análisis del proyecto
Preparar presentación	Confeccionar documento final

Tabla A.1: Relación de dependencia para las tareas de la gestión del proyecto.

La tabla anterior muestra la relación de dependencia, se respetará esta relación ya que las tareas a elaborar se agrupan y tienen fecha de entrega por separado.

Tarea dependiente	Tarea predecesora
Uso de la API Nanos6	GEP
Integrar OmpSs-2 en C/C++	Uso de la API
Integrar OmpSs-2 en Java	Integrar OmpSs-2 en C/C++
Integrar OmpSs-2 en Python	Integrar OmpSs-2 en Java
Estudio previo del rendimiento	Integrar OmpSs-2 en Python
Desarrollo aplicación COMPSs+OmpSs-2	Estudio previo del rendimiento
Estudio del rendimiento	Desarrollo aplicación COMPSs+OmpSs-2
Redactar la memoria	Estudio del rendimiento

Tabla A.2: Relación de dependencia para las tareas de implementación del proyecto.

Salvo por algún motivo que implique bloquear una tarea, no se deberán adelantar tareas dependientes a las predecesoras, entre estos posibles motivos se contemplan errores en la implementación que nos bloqueen y se puedan ir haciendo otras cosas y cambios generales en las tareas a realizar.

A.3. Estimación temporal de las tareas y recursos necesarios

En el momento en el que se han enumerado y explicado las tareas se ha comentado la duración temporal y los recursos necesarios para cada actividad. En las siguientes dos secciones se recopilan estos datos en forma de tabla.

A.4. Estimación temporal de las tareas

Tarea	Estimación temporal (horas)
Gestión del proyecto	60
Uso de la API Nanos6	60
Integrar OmpSs-2 en C/C++	96
Integrar OmpSs-2 en Java y Python	93
Desarrollo aplicación COMPSs+OmpSs-2	84
Estudio del rendimiento	84
Redactar la memoria	72
Total	549

Tabla A.3: Estimación temporal de las tareas.

A.5. Duración real de las tareas

Todas las tareas han durado lo esperado a excepción de la tarea que consiste en realizar la integración de *OmpSs-2* en *Java* y *Python*, nos hemos limitado a hacer el estudio preliminar, por lo que ha durado 15 horas.

Tarea	Estimación temporal (horas)
Gestión del proyecto	60
Uso de la API Nanos6	60
Integrar OmpSs-2 en C/C++	96
Integrar OmpSs-2 en Java y Python	15
Desarrollo aplicación COMPSs+OmpSs-2	84
Estudio del rendimiento	84
Redactar la memoria	72
Total	471

Tabla A.4: Estimación temporal de las tareas.

A.6. Recursos necesarios para las tareas

Tarea	Recursos necesarios
Gestión del proyecto	GitHub, Kile, Trello, Gantter, Google Drive
Uso de la API Nanos6	Compilador de C y C++, Mercurium, Nanos6
Integrar OmpSs-2 en C/C++	COMPSs, Compilador de C y C++, Mercurium, Nanos6
Integrar OmpSs-2 en Java y Python	COMPSs, Compilador de C y C++, Mercurium, Nanos6
Estudio del rendimiento	COMPSs, Compilador de C y C++, Mercurium, Nanos6
Redactar la memoria	GitHub, Kile, LibreOffice

Tabla A.5: Recursos necesarios para las tareas.

En la tabla anterior, se muestran los recursos estrictamente necesarios para realizar cada tarea, sin embargo, se ofrece ahora una lista de los recursos *hardware* y *software* que se han utilizado para la realización del proyecto en general. Además hay que tener en cuenta todos los recursos humanos necesarios.

A.6.1. Recursos hardware

- **Ordenador portátil:** Proporcionado por el *BSC*, Dell Latitude 7480 Intel® Core™ i7-6650U Processor (Dual Core, 4M Cache, 2.2GHz,15W, vPro), 512GB SSD (*Solid State Drive*), Intel® HD Graphics 540 y 16 GB de memoria *RAM*.

- **Pantalla:** Es habitual la configuración de portátil con una pantalla para simular una torre, la pantalla externa es también Dell, el modelo Professional P2217H.
- **Periféricos:** Todos los periféricos, ratón y teclado en este caso.
- **Clústers:** Con tal de medir el rendimiento ejecutando la aplicación que se ha desarrollado haciendo uso de la integración, necesitaremos un clúster con una arquitectura heterogénea. En la lista de candidatos se encontraban *MinoTauro* y *CTE-Power*, pero al final se descartó *MinoTauro* por problemas de compatibilidad y se utilizó *MareNostrum4*
- **Puesto de trabajo:** El equipo de *WDC* se encuentra en el edificio *K2M*, allí es donde el desarrollador tiene su puesto de trabajo y ha desarrollado la mayoría del proyecto.

A.6.2. Recursos software

- **Ubuntu 18.04:** Con tal de desarrollar se necesita un sistema operativo, el portátil tiene instalado *Ubuntu 18.04*.
- **GitHub y GitLab:** Para efectuar un control de versiones sencillo y eficaz, se ha utilizado el *GitHub* personal del desarrollador para gestionar la documentación y el *GitLab* del grupo *WDC* para gestionar el código.
- **Editores:** Para editar código en *Java* se ha utilizado *IntelliJ IDEA*, para *Python PyCharm*, ambos de *JetBrains*, y para C y C++ se ha usado *Vim*.
- **Terminal:** La gran parte del tiempo ha estado entre terminales haciendo implementaciones y probando su funcionamiento, para hacer uso de un terminal utilizamos el emulador de terminales *Terminator*.
- **Planificación y organización:** Para hacer el diagrama de *Gantt* se ha utilizado *Ganter* como extensión para *Google Drive*. Además para organizarse y emplear la metodología iterativa se utilizará *Trello*.
- **Compiladores y gestores de proyectos:** Para compilar código en C y C++ se usa *gcc* y *g++* respectivamente, para todo código que use *OmpSs-2* se ha utilizado *Mercurium*. El proyecto de *COMPSs* está gestionado con *Maven*, de esta manera se pueden generar todos los ficheros de *Java* de manera sencilla.
- **Software del proyecto:** Para poder desarrollar el proyecto, se precisa de una instalación de *COMPSs*, *Nanos6* y *Mercurium*. Además, para medir el rendimiento se utiliza *Extrae* y *Paraver*. Con fines de *debugging* se ha utilizado *gdb*.
- **Editores de texto:** Para escribir el *LaTeX* se utiliza *Kile*.

A.6.3. Recursos humanos

- **Director y co-director:** Han efectuado el seguimiento del proyecto de manera rutinaria y han ayudado a que el desarrollador sea capaz de llevarlo a cabo.
- **Soporte:** Al utilizar *software* de diversos proyectos, todas las personas que han ayudado al desarrollador a solucionar problemas han sido recursos necesarios del proyecto.
- **Desarrollador:** Persona encargada de llevar a cabo en última instancia el proyecto.

A.7. Valoración de alternativas y plan de acción

En un proyecto de este tipo, es probable que haya desviaciones respecto el plan original. Esto es normal, tan sólo hay que saber cómo actuar ante estas desviaciones. Cualquier error en una implementación puede acarrear tiempo de más para solucionarlo, e incluso algo que se implementó hace mucho puede influir en las del futuro, por ello nuestra planificación intenta ser flexible, aún así, debemos planear como actuar en estos casos.

- Si una tarea dura menos de lo esperado, sencillamente hay que coger la siguiente de la planificación y empezar a hacerla. Que una tarea dure menos que otra nos puede aportar un margen de acción muy útil.
- Si una tarea dura más tiempo de lo esperado, habrá que planteárselo de dos maneras, o bien se acorta otra tarea con tal de ajustarnos a la planificación o bien se intentan reducir lo mínimo posible los objetivos del proyecto para poderlo acabar en el tiempo establecido.

La tarea que más tiempo puede llevar dada la aparición de imprevistos es la de integrar *OmpSs-2* en el *binding* de C. Dado que la documentación aún está en una fase un tanto primeriza y no siempre tenemos por qué contar con el apoyo de soporte, por lo cuál habrá que invertir tiempo extra en ese caso.

El resto de tareas van un poco de la mano de esta anterior, no debería haber ninguna complicación extra. Como mucho en el estudio del rendimiento podemos encontrar resultados que no nos gusten o no acaben de agradar del todo, pero es parte del proyecto, se intentará mejorar dentro del tiempo estipulado.

Por tanto, siempre que falte tiempo para realizar una tarea se intentará equilibrar entre el resto, ya que el riesgo de sufrir un imprevisto es bastante bajo.

Para ser más previsores, en las reuniones de seguimiento se intentarán prever estos posibles problemas durante la realización del proyecto.

De hecho, en un punto del proyecto descubrimos que *MinoTauro* no era compatible con *OmpSs-2* si queríamos hacer uso de las *GPUs*, por esto tuvimos que rectificar y decidir utilizar *MareNostrum4* y sacar provecho tan sólo de las *CPUs*. También, el hecho de no haber realizado la integración en *Java* y *Python* supone de alguna manera una desviación, pero ya estaba prevista.

Apéndices B

Gestión económica y de sostenibilidad del proyecto

B.1. Autoevaluación sobre la sostenibilidad

Todos los alumnos que hayan realizado un trabajo de final de grado este cuatrimestre, saben qué supone el impacto de un proyecto sobre la sostenibilidad. Es difícil para mí, analizar de manera crítica y de alguna manera metódica estos impactos que se describen en la encuesta.

En cuanto a conocer los impactos, tener sentido común e intentar siempre reducir los impactos que de alguna manera sean insostenibles, sí que hay un conocimiento pleno y ganas de mantenerlo, pero desconozco los métodos más avanzados y profesionales para abordar la reducción de impactos negativos y aumentar la sostenibilidad de un proyecto.

Los aspectos ambientales son de alguna manera los que resultan más intuitivos, de manera que no resulta difícil desenvolverse en primer momento, pero no conozco en profundidad el tema.

En cuánto a aspectos sociales, de igual manera que los ambientales, soy capaz de reconocer cuando un proyecto impacta de manera negativa en la sociedad (que puede suceder aunque aparentemente parezca beneficioso en primer momento). Pero cuando se trata de poner sobre la mesa un estudio preciso sobre estos aspectos, no soy capaz, desconozco la teoria.

Sobre aspectos económicos es donde más fallo, pese a que tengo cierta formación gracias a asignaturas impartidas en la *FIB*, me resulta complicado entender y comprender como a mí me gustaría estos aspectos. Aún así tengo la voluntad y las ganas de ser capaz de analizarlo de manera correcta.

Básicamente, se quiere poder analizar de igual manera los tres aspectos, ambiental, social y económico, existe una voluntad fehaciente que por desgracia puede resultar insuficente cuando se ahonda en el análisis.

B.2. Gestión económica y sostenibilidad

A lo largo de esta sección, se detallará la gestión económica del proyecto. Se estudiarán los costes tanto directos e indirectos y las posibles desviaciones en términos económicos. Además se estudiará también el componente que queda olvidado en la mayoría de proyectos informáticos, la sostenibilidad de este.

B.2.1. Costes directos e indirectos

Los costes directos de este proyecto, vienen a partir de los recursos definidos para el proyecto y las actividades que se llevan a cabo en este. Es importante tener en cuenta que por mucho que los recursos tenga un coste directo en el proyecto y estén asociados a este, los recursos tienen una vida útil por lo cuál habrá un grado de amortización del coste por recurso. Estos costes se recapitulan a continuación en forma de tabla.

Unidades	Unidad	Precio/Unidad(€)	Vida útil (años)	Amortización (€/h)
Recursos hardware				
Dell Latitude 7480	1	1500	4	0.21
Dell Professional P2217H	1	250	4	0.03
Periféricos	1	50	4	0.07
Total	-	0	-	0.31
Recursos software				
Ubuntu 18.04	1	0	-	0
GitHub	1	0	-	0
GitLab	1	0	-	0
Terminator	1	0	-	0
Gantter	1	0	-	0
Trello	1	0	-	0
Google Drive	1	0	-	0
Kile	1	0	-	0
LibreOffice	1	0	-	0
Maven	1	0	-	0
GNU Compiler Collection	1	0	-	0
GDB	1	0	-	0
Extrae	1	0	-	0
Paraver	1	0	-	0
OmpSs-2	1	0	-	0
COMPSs	1	0	-	0
Total	-	0	-	0

Tabla B.1: Costes directos divididos en recursos hardware y software.

Los recursos *GitHub*, *IntelliJ IDEA* y *PyCharm*, no añaden ningún coste al proyecto ya que se utilizan versiones y subscripciones para estudiantes. Por otra parte *Gantter* proporciona una versión de prueba de 30 días, por lo cuál tampoco añade ningún coste.

En la anterior tabla se han omitido los dos clústers, ya que querríamos costearlos a €/h y no por unidad. Se puede considerar que *MareNostrum4* y *CTE-Power* no suponen ningún coste al proyecto, ni de adquisición de este ni eléctrico, ya que son proporcionados por el *BSC*, aún así para proporcionar una visión más realista se ha seleccionado una máquina de *Amazon Web Services AWS* con características similares a cada clúster, con tal de seleccionar un precio €/h adecuado. Para *MareNostrum4* se ha considerado que la instancia de modelo *r5.12xlarge* a un coste de 2.7 €/h, y para *CTE-Power* la instancia *Amazon EC2 P3* modelo *p3.8xlarge* a un coste de 10.85 €/h.

En cuanto a costes directos solo nos falta comentar los que provienen de los recursos humanos del proyecto.

	Horas (h)	Precio/Hora(€)	Total(€)
Desarrollador	489	10	4890
Director	55	30	1650
Codirector	50	30	1500
Soporte	20	20	400
Total	-	-	8440

Tabla B.2: Costes directos provenientes de recursos humanos.

Es difícil cuantificar las horas que entran en soporte, dado que este puede ser efectuado por varias personas con distintos sueldos, se ha optado por pensar que se dedicará un total de 20 horas y que el prefio esta en la media entre un desarrollador y un director.

Los costes indirectos del proyecto engloban el alquiler de la oficina (edificio K2M) el gasto energético de esta , su debido mantenimiento la contratación de servicios como internet, u otros

servicios que se ofrezcan al conjunto de trabajadores en general. Del enlace [6] se extrae que la superfície de la primera planta del edifico K2M es de $445,6 m^2$ y de [8] que el coste para una empresa es de 17.8 €/m^2 al mes. El gasto en consumo eléctrico de la planta se da por hecho que entra en el precio estipulado por el alquiler.

	Meses	m^2	Precio mensual(€/m^2)	Total(€)
Alquiler oficinas	4	445,6	17,8	31.726,72
Total	-	-	-	31.726,72

Tabla B.3: Costes indirectos derivados del alquiler de las oficinas.

B.2.2. Imprevistos y contingencias

En la planificación temporal se tuvo en cuenta que todos los proyectos sufren de desviaciones e imprevistos. Las desviaciones pueden ser anticipadas efectuando las reuniones de seguimiento, pero los imprevistos son inevitables. Evidentemente, ante la aparición de cualquiera de estos dos, habría un aumento de los costes directos e indirectos (el tiempo dedicado por los recursos humanos aumenta, por lo tanto el coste, etc).

En la siguiente sección, se detallarán los costes a nivel de recurso y uso temporal de estos por cada actividad a desarrollar en el proyecto. Con tal de no proponer un presupuesto que pueda quedar negativo por culpa de imprevistos, se dedicará un porcentaje extra de las tareas a poder sobrevenir el posible coste. En el mejor de los casos no tendremos imprevistos o bien será una cantidad tan pequeña que el porcentaje extra reservado por tarea nos permitirá costearnos los imprevistos. Por otra banda, si tenemos un imprevisto con cada tarea es muy posible que no seamos capaces de cubrir los costes (en función del porcentaje que se reserve, claro).

B.2.3. Presupuesto

	Uds.	Precio(€o €/h)	Vida útil(años)	Amortización(€/h)	Precio(€)
Costes directos					17.034,58
Gestión del proyecto	60 horas				620,45
Dell Latitude 7480	1	1500	4	0,2841	17,05
Dell Professional P2217H	1	250	4	0,0473	2,84
Periféricos	1	50	4	0,0095	0,57
Ubuntu 18.04	1	0	-	-	0
GitHub	1	0	-	-	0
Gantter	1	0	-	-	0
Trello	1	0	-	-	0
Google Drive	1	0	-	-	0
Kile	1	0	-	-	0
LibreOffice	1	0	-	-	0
Desarrollador	1	10	-	-	600
Uso de la API Nanos6	60 horas				620,45
Dell Latitude 7480	1	1,500	4	0,2841	17,05
Dell Professional P2217H	1	250	4	0,0473	2,84
Periféricos	1	50	4	0,0095	0,57
OmpSs-2	1	0	-		0
COMPSs	1	0	-	-	0
GNU Compiler Collection	1	0	-	-	0
Desarrollador	1	10	-	-	600
Integrar OmpSs-2 en C/C++	96 horas				992,72
Dell Latitude 7480	1	1,500	4	0,2841	27,27
Dell Professional P2217H	1	250	4	0,0473	4,54
Periféricos	1	50	4	0,0095	0,912
OmpSs-2	1	0	-	-	0

COMPSs	1	0	-	-	0
GNU Compiler Collection	1	0	-	-	0
Desarrollador	1	10	-	-	960
Integrar OmpSs-2 en Python y Java	93 horas				961,7
Dell Latitude 7480	1	1,500	4	0,2841	26,42
Dell Professional P2217H	1	250	4	0,0473	4,40
Periféricos	1	50	4	0,0095	0,88
OmpSs-2	1	0	-	-	0
COMPSs	1	0	-	-	0
GNU Compiler Collection	1	0	-	-	0
Desarrollador	1	10	-	-	930
Desarrollo de una aplicación COMPSs+OmpSs-2	84 horas				868,63
Dell Latitude 7480	1	1500	4	0,2841	23,86
Dell Professional P2217H	1	250	4	0,0473	3,97
Periféricos	1	50	4	0,0095	0,80
OmpSs-2	1	0	-	-	0
COMPSs	1	0	-	-	0
GNU Compiler Collection	1	0	-	-	0
Desarrollador	1	10	-	-	840
Estudio del rendimiento	84 horas				12.250,63
Dell Latitude 7480	1	1,500	4	0,2841	23,86
Dell Professional P2217H	1	250	4	0,0473	3,97
Periféricos	1	50	4	0,0095	0,80
MareNostrum4	10	2,7	-	-	2268,0
CTE-Power	10	10,85	-	-	9,114
OmpSs-2	1	0	-	-	0
COMPSs	1	0	-	-	0
GNU Compiler Collection	1	0	-	-	0
Extrae	1	0	-	-	0
Paraver	1	0	-	-	0
Desarrollador	1	10	-	-	840
Redactar la memoria	72 horas				720
GitHub	1	0	-	-	0
Kile	1	0	-	-	0
LibreOffice	1	0	-	-	0
Desarrollador	1	10	-	-	720
Costes indirectos					31.726,72
Alquiler oficinas K2M	-	-	-	-	31.726,72
Servicios	-	-	-	-	-
Total acumulado					48.761,3
Contingencia	10 %				53.637,43
Total sin IVA					53.637,43
Total con IVA	21 %				64.901,29

En la anterior tabla de detalla el presupuesto final del proyecto, se define la contingencia y se aplica el impuesto *IVA*. En esta última tabla no siempre se ha hecho referencia a todos los recursos, esto ha sido así siempre que el recurso tuviera un papel secundario (por ejemplo editores o terminales, que son cosas a elección del desarrollador...) y el coste fuera nulo.

B.3. Planificación y diagrama de Gantt correspondiente

		Name	Duration	Start	Finish	Predecessors
1		**Proyecto**	90days	02/18/2019	06/21/2019	
2		**Gestión del proyecto**	11days	02/18/2019	03/04/2019	
3		**Elaboración del primer entregable**	4days	02/18/2019	02/21/2019	
8		**Elaboración del segundo entregable**	1day	02/22/2019	02/22/2019	3
11		**Elaboración del tercer entregable**	3days	02/25/2019	02/27/2019	8
14		**Elaboración del cuarto entregable**	3days	02/28/2019	03/04/2019	11
15		Confeccionar documento final	2days	02/28/2019	03/01/2019	
16		Preparar presentación	1day	03/04/2019	03/04/2019	15
17		**Implementación**	39days	03/05/2019	04/26/2019	14
18		Uso de la API de Nanos6 (offloading)	10days	03/05/2019	03/18/2019	8
19		**Integrar OmpSs-2**	29days	03/19/2019	04/26/2019	18
20		Integrarlo en C/C++	16days	03/19/2019	04/09/2019	18
21		Integrarlo en Java	13days	04/10/2019	04/26/2019	20
22		Integrarlo en Python	13days	04/10/2019	04/26/2019	20
23		Estudio previo del rendmiento	6days	04/29/2019	05/06/2019	19
24		Desarrollo/Portaje de aplicación a COMPSs+OmpSs-2	14days	04/29/2019	05/16/2019	17
25		Estudio del rendimiento	14days	05/17/2019	06/05/2019	24
26		Redactar la memoria	12days	06/06/2019	06/21/2019	25

Figura B.1: Tareas definidas para el diagrama de Gantt.

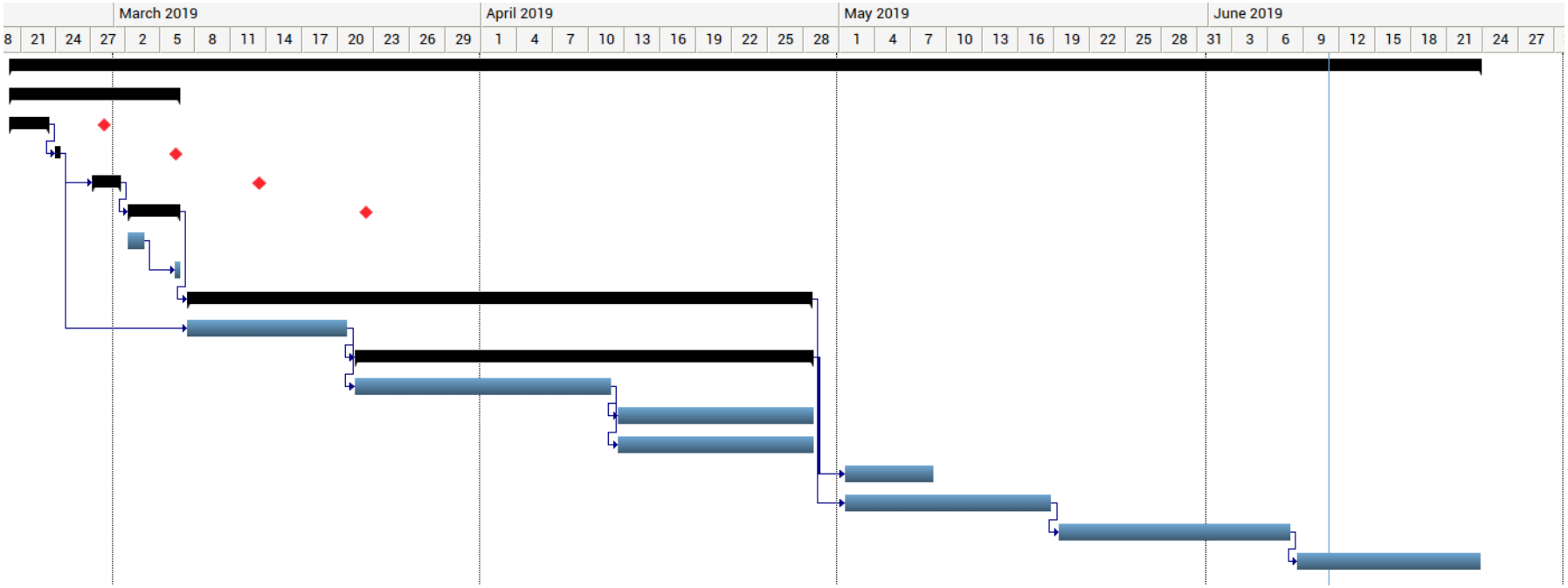

Figura B.2: Diagrama de Gantt del proyecto.

Apéndices C

Código para realizar la integración COMPSs+OmpSs-2

C.1. Ejecutar tareas de COMPSs como tareas de OmpSs-2

Para la implementación de los *workers* hay dos ficheros distintos *nio_worker_c.cc* y *worker_c.cc* respectivamente para el persistente y el no persistente. En cada uno de estos ficheros deberemos añadir el código para que cuando los compilemos con la opción de *OmpSs-2* y *COMPSs* los ejecute (en función de si es persistente o no) y activen el modo librería.

```
1
2  #ifdef OMPSS2_ENABLED
3  #include <pthread.h>
4  #include <nanos6/bootstrap.h>
5  #include <nanos6/library-mode.h>
6  #endif
7
8      ...
9
10 #ifdef OMPSS2_ENABLED
11     char const *error = nanos6_library_mode_init();
12     if (error != NULL)
13     {
14         fprintf(stderr, "Error initializing Nanos6: %s\n", error);
15         return 1;
16     }
17     std::cout << "[C-BINDING] Nanos6 initialized" << std::endl;
18 #endif
19
20     ...
21
22 #ifdef OMPSS2_ENABLED
23     nanos6_shutdown();
24     std::cout << "[C-BINDING] Nanos6 shutdown" << std::endl;
25 #endif
```

Código C.1: Código necesario para la gestión del modo librería de Nanos6 en nio_worker_c.cc y worker_c.cc

El código mostrado en la imagen C.1 se divide en tres bloques divididos por puntos suspensivos, el código que hubiera entre estos, depende de si el fichero es *nio_worker_c.cc* o *worker_c.cc* y es independiente de ellos. Se hace uso de macros del preprocesador de *C*, la macro *#ifdef VAR*

comprueba en tiempo de compilación si la variable *OMPSS2_ENABLED* está definida, si lo está las líneas contenidas entre la primera y el *#endif* serán incluidas en el fichero, sino no lo serán.

En cuánto a los tres bloques, el primero incluye cabeceras necesarias para *Nanos6*, y el resto han sido ya explicados en la sección 2.3.

En el caso de *OmpSs*, hay que registrar exactamente los *threads* de un proceso que ejecutarán tareas y también hay que compilar necesariamente toda la aplicación con *Mercurium* y el *flag --ompss*, pero por otra parte, no hay necesidad de añadir más código a parte del que se encarga de registrar los *threads*. En *OmpSs-2* como vimos en el ejemplo, es necesario hacer una gestión más compleja en cuánto a generación de código, pero es independiente del *thread* en el cuál se ejecuta.

Con tal de cambiar el código que se genera en el *executor*, hay que hacer modificaciones en el programa que compila la interfaz del programa para generar el *stubs*, el *executor* y el resto de ficheros necesarios. En la estructura que tenemos nosotros del compilador, el fichero que contiene el código para autogenerarlo todo es *c-backend.c*, en el cuál deberemos hacer unas modificaciones.

Vamos a centrarnos en las modificaciones que harán que podamos generar el *struct* personalizado para cada tarea y la llamada a *nanos6_spawn_function*, pero cómo se genera el mecanismo de sincronización (*pthread_cond_wait*, *pthread_cond_signal* y las estructuras de datos necesarias) es trivial, ya que no varía según la aplicación que se compile, es siempre el mismo.

Con tal de generar el *struct* para cada tarea, incluiremos su definición en los ficheros que hacen la función de cabecera. Cada *struct* será único para cada tarea (podría tener los mismos campos, pero nunca el mismo nombre). Para generar el *struct* de la tarea, necesitamos tener información sobre la función que se ejecutará, contamos con una estructura *function* que almacena toda la información necesaria.

La imagen C.2 contiene el código necesario para incluir en la cabecera la definición del *struct* de una función en concreto. Se recorrerán los argumentos de la función y se comprobará si tiene retorno, habrá un campo del *struct* por cada argumento que tenga la función y uno adicional en caso de que tenga retorno.

```
1  static void generate_struct_nanos6_wrapper(FILE *outFile,
2  Types current_types,
3  function *func) {
4  argument *arg;
5  fprintf(outFile, "typedef struct {\n", func->methodname);
6
7  char* return_type = construct_returntype(func);
8
9  if (func->return_type != void_dt && return_type != NULL) {
10 fprintf(outFile, "\t%s ret;\n", return_type);
11 }
12
13 arg = func->first_argument;
14 while (arg != NULL) {
15 fprintf(outFile, "\t%s;\n", construct_type_and_name(arg));
16 arg = arg->next_argument;
17 }
18 fprintf(outFile, "} %s_struct_t;\n", func->methodname);
19 }
```

Código C.2: Función para generar la definición de los structs.

Para generar el código del *wrapper* lo primero que necesitamos hacer es generar el nombre de la función, que es del estilo "function_wrapper" y tiene por argumento un puntero a *void* que apunta al *struct* que contiene los argumentos para la función que vamos a ejecutar y el valor de retorno de esta en caso que tenga. Entonces, si la función tiene retorno distinto del tipo *void*, la ejecutaremos asignándole el valor al *struct*, en caso contrario tan sólo la ejecutaremos, si algún

valor de los argumentos se modifica, la única posibilidad para que esto se refleje en *COMPSs* es que fuera un puntero, por lo que quedará modificado también dentro del *struct*.

La imagen C.3 muestra como está hecho.

```
1
2  static void generate_nanos6_wrapper(FILE *outFile, function* func) {
3  fprintf(outFile, "void %s_wrapper(void* args) {\n",
4  func->methodname);
5
6  fprintf(outFile,
7  "\t%s_struct_t* struct_ = (%s_struct_t*) args;\n",
8  func->methodname,
9  func->methodname);
10
11 char* func_to_execute;
12 int printed_chars = 0;
13
14 if (func->return_type != NULL && func->return_type != void_dt) {
15 fprintf(outFile, "\tstruct_->ret = ", func->return_type);
16 }
17
18 if ((func->classname != NULL) && (func->access_static == 0)) {
19 printed_chars = asprintf(&func_to_execute,
20 "\t%s->%s(",
21 func->name,
22 func->methodname);
23 } else {
24 printed_chars = asprintf(&func_to_execute,
25 "\t%s(", func->name);
26 }
27
28 if (printed_chars < 0) {
29 asprintf_error(func_to_execute,
30 "ERROR: Not possible to
31 generate method execution.\n");
32 }
33
34 fprintf(outFile, "%s", func_to_execute);
35
36 argument* args = func->first_argument;
37
38 int first = 1;
39 while (args != NULL) {
40 if (first) {
41 first = 0;
42 }
43 else {
44 fprintf(outFile, ", ");
45 }
46 fprintf(outFile, "struct_->%s", args->name);
47
48 args = args->next_argument;
49 }
50
51 fprintf(outFile, ");\n");
52 fprintf(outFile, "}\n");
53 }
```

Código C.3: Función para generar el wrapper de la tarea.

Ahora ya tenemos lo que necesitamos, *structs* para pasar los argumentos, el *wrapper* del que haremos *spawn* y la propia tarea que se acabará ejecutando, tan sólo falta hacer la llamada a *nanos6_spawn_function*. Para hacer esto, tenemos que declarar el *struct* y asignarle los valores que vienen de *COMPSs*, acto seguido se generará la llamada a *nanos6_spawn_function* con el *wrapper* como función a ejecutar desde *OmpSs-2*, la dirección del *struct* que contiene los argumentos y todo lo necesario para efectuar los mecanismos de sincronización que vimos en la sección 2.3.2. Una vez hemos esperado a que la tarea se haya ejecutado, si el retorno era distinto del tipo *void* hay que asignarlo al retorno de la tarea a nivel de *COMPSs*. En la imagen C.4 se ve el código que se encarga de generar esta parte.

```
1  fprintf(outFile, "\t\t\t%s_struct_t %s_struct;\n", func->methodname,
2  func->methodname);
3
4  assign_arguments_to_struct(outFile, func);
5
6  char* nanos6_spawner;
7  printed_chars = asprintf(&nanos6_spawner,
8  "\t\t\tnanos6_spawn_function(%s_wrapper,
9  &%s_struct,
10 %s, %s,
11 \"%s_spawned_task\");\n",
12 func->methodname,
13 func->methodname,
14 "condition_variable_callback",
15 "&cond_var",
16 func->methodname);
17
18 if (printed_chars < 0) {
19 asprintf_error(nanos6_spawner,
20 "Not possible to generate function spawn.\n");
21 }
22
23 fprintf(outFile, "%s\n", nanos6_spawner);
24
25 fprintf(outFile, "\t\t\twait_condition_variable(&cond_var);\n");
26
27 if (func->return_type != void_dt && func->return_type != NULL) {
28 char* return_var;
29 printed_chars = asprintf(&return_var,
30 "return_obj = %s_struct.ret;\n",
31 func->methodname);
32
33 fprintf(outFile, "%s", return_var);
34
35 if (printed_chars < 0) {
36 asprintf_error(return_var,
37 "Not possible to generate return variable.\n");
38 }
39 }
```

Código C.4: Código para hacer un spawn de una tarea para OmpSs-2 en COMPSs.

C.2. Entorno de compilado para COMPSs+OmpSs-2

Para compilar una aplicación que utilice *OmpSs* se necesita poner el *flag --ompss* en *Mercurium*, y el *flag --ompss-2* para *OmpSs-2*. En la anterior integración se compilaba la totalidad de los ficheros con *Mercurium* y el *flag --ompss* debido a que no existía el modo librería. Ahora, es estrictamente

necesario compilar con *Mercurium* y el *flag --ompss-2* tan sólo un único fichero, el que contiene las funciones a ejecutar como tareas.

Habrá que modificar el método actual de compilación de una aplicación del *binding* de *C/C++* con tal de que aplique lo relativo a la compilación explicado en la sección 2.3.3. Para el usuario compilar una aplicación con *OmpSs* es muy sencillo, ejecutando en el terminal el comando *compss_ build_ app --ompss ejemplo* empezaría el proceso de compilado del *master* y el *worker* de manera automática, en la nueva integración no puede ser más difícil. Antes de indagar en las modificaciones en concreto en los *scripts* vamos a ver una descripción superficial de cómo se hace actualmente.

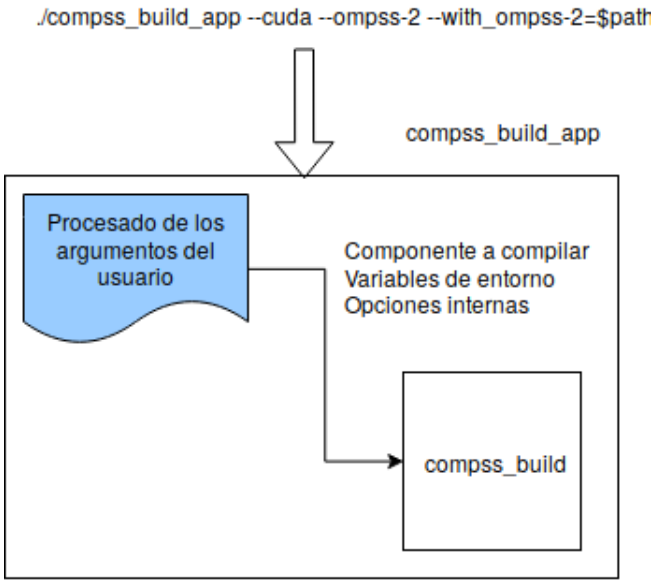

Figura C.1: Estructura superficial de los *scripts* para compilar una aplicación de COMPSs.

En la imagen C.1 vemos que existen dos *scripts*, *compss_ build_ app* y *compss_ build*, el primero es el encargado de recoger los argumentos que el usuario haya introducido para la compilación de su aplicación (si se utiliza *OmpSs*, donde está la instalación de *OmpSs*, si se utiliza *OpenCL*, o *Cuda*, etc) y procesarlos configurando el entorno correcto para la compilación según estos argumentos que ha introducido el usuario. Una vez configurado el entorno se ejecuta el segundo *script compss_ build* tanto para el *master* como para el *worker*, que será realmente quién efectúe la compilación.

Para ello se utiliza un conjunto de herramientas muy populares desarrolladas por *GNU*, *Autotools*. Estas herramientas tienen como objetivo facilitar a los usuarios (que no a los desarrolladores...) la compilación y portabilidad entre plataformas. Pese a que es un campo interesante y muy útil, no entraremos demasiado en detalles, daremos una visión general y entraremos en las modificaciones que se han realizado con tal de poder compilar aplicaciones del tipo *COMPSs+OmpSs-2*.

Como ya hemos visto, *compss_ build_ app* hace de alguna manera de *wrapper* de *compss_ build*, es decir, lo ejecuta con los parámetros adecuados las veces que sea necesario. Este segundo, es el que toca de primera mano *Autotools*, con tal de utilizarlo para compilar una aplicación hay que generar una serie de ficheros. La siguiente imagen nos muestra una porción del *script compss_ build* para generar el *script autogen.sh*, que al ser ejecutado nos generará estos ficheros que *Autotools* necesita.

```
1  /bin/cat > autogen.sh << EOF
2  #!/bin/bash
3  set -e
4
5  /usr/bin/aclocal
6  /usr/bin/automake -a -c
7  /usr/bin/autoconf
8  EOF
```

Código C.5: Porción del script autogen.sh que genera los ficheros base necesarios para Autotools.

La primera y la última línea de la imagen C.5 sirven para escribir el código contenido entre estas en el fichero *autogen.sh*. El *script autogen.sh* es generado por *compss_build* cada vez que se quiere compilar una aplicación y también en función de si se está compilando el *master* o el *worker* (de hecho, se encuentran en sitios distintos), tendrá una forma u otra.

Una vez ejecutada esta porción del *script autogen.sh*, entre otros, esto nos habrá generado un *script configure*, que al ser ejecutado generará un *Makefile* adaptado al entorno que hayamos especificado. Nuestro proceso de compilado requiere ser flexible, ya que permitimos utilizar *OmpSs*, a veces con *Cuda*, a veces con *OpenCL*... Por tanto, necesitamos pasarle al compilador dónde se encuentran las librerías que necesitamos, *flags* adicionales que necesitemos (*--ompss*, *--ompss-2*, *--cuda*...). Por suerte, este *configure*, nos permite pasar variables de entorno a tener en cuenta durante la compilación, además de opciones personalizadas, a continuación del código de la imagen C.5 añadiremos código para que *autogen.sh* ejecute el *script configure* con las opciones que necesitamos.

```
1  if [ "$OMPSS2_ENABLED" == "enabled" ]; then
2  ldflags_aux="$ldflags_aux -L$OMPSS2_DIR/lib
3  -Wl,-rpath,$OMPSS2_DIR/lib"
4  cflags_aux="$cflags_aux -I$OMPSS2_DIR/include"
5  conf_opts="--with-ompss-2"
6  fi
7
8  if [ "$OMPSS_ENABLED" == "enabled" ]; then
9  ldflags_aux="$ldflags_aux -L$OMPSS_DIR/lib"
10 cflags_aux="$cflags_aux -I$OMPSS_DIR/include"
11 conf_opts="$conf_opts --with-ompss"
12 fi
13
14 if [ "$CUDA_ENABLED" == "enabled" ]; then
15 ldflags_aux="$ldflags_aux -L$CUDA_HOME/lib64"
16 cflags_aux="$cflags_aux -I$CUDA_HOME/include"
17 conf_opts="$conf_opts --with-cuda"
18 fi
19
20 /bin/cat >> autogen.sh << EOF
21 ./configure --with-cs-prefix=$CS_HOME $conf_opts \
22 CXXFLAGS="$CXXFLAGS $cflags_aux"      \
23 CFLAGS="$CFLAGS $cflags_aux"          \
24 LDFLAGS="$LDFLAGS $ldflags_aux"
25 EOF
26
27 /bin/chmod +x autogen.sh
```

Código C.6: Porción del script autogen.sh que se encarga de customizar la ejecución de configure.

Anteriormente hemos mencionado que el *script compss_build_app* es el encargado de recolectar los *flags* que nos interesan del usuario en el proceso de compilación y servirlos al *script compss_build*, como se puede ver en la imagen C.6 en función de si las variables de entorno *OMPSS2_ENABLED*, *OMPSS_ENABLED*, o *CUDA_ENABLED* tienen por valor la cadena de ca-

rácteres *enabled*, definirá el valor de los *flags* para la fase de compilación *CXXFLAGS* y *CFLAGS*, que se asignarán via la variable *cflags_aux* y para la fase de *linking LDFLAGS* que se asignará via la variable *ldflags_aux*, además de las opciones personalizadas (que serán pasadas mediante la variable *conf_opts*).

Dos ficheros esenciales en todo este proceso, son *configure.ac* y *Makefile.am*, uno utiliza una serie de *macros* para generar de manera customizada *configure* (por ejemplo, con las opciones mencionadas anteriormente), y el otro indica cuáles son los ficheros que deben ser compilados, cómo y adónde irán una vez compilados. Las modificaciones que explicamos ahora mismo sólo se han aplicado al *worker*, para el *master* no se ha necesitado ningún tipo de modificación. Para declarar una opción personalizada en *configure* se utiliza la macro *AC_ARG_WITH*, que tiene como argumentos el nombre de la opción (ompss, que pasará a ser with-ompss), el mensaje de ayuda para el usuario y lo que hay que hacer en caso que se habilite la opción, la imagen C.7 muestra exactamente cómo se hace.

```
1 AC_ARG_WITH([ompss],
2 AS_HELP_STRING([--with-ompss], [Enables the use of OmpSs and
3 specificates its directory.]),
4 [
5 ompss=true
6 ]
7 )
```

Código C.7: Macros necesarias en configure.ac para habilitar la opcion –ompss en configure.

Para el resto de opciones se utiliza el mismo mecanismo, pero las acciones a realizar si se habilita la opción son distintas. Con tal de compilar una aplicación de *OmpSs-2* necesitamos librerías de *Pthreads*, *dl*[1], *nanos6* (evidentemente para todo lo que concierne al *runtime*), y *custom-nanos6*, que no es más que la adaptación a librería del *nanos6-library-mode.o* que se utilizó en la sección 2.3.3. La imagen C.8 muestra exactamente cómo lo hacemos.

[1] https://en.wikipedia.org/wiki/Dynamic_loading

```
AC_ARG_WITH([ompss-2],
AS_HELP_STRING([ --with-ompss-2], [Enables the use of OmpSs2
and specificates its
directory.]),
[
ompss2=true

AC_CHECK_LIB([pthread],
[pthread_cond_signal],
[],
[
AC_MSG_ERROR([Pthread library not found,
is needed to enable OmpSs2.])
])

AC_CHECK_LIB([dl],
[dlopen],
[],
[
AC_MSG_ERROR([dl library not found,
is needed to enable OmpSs2.])
])

AC_CHECK_LIB([nanos6],
[nanos6_spawn_function],
[],
[
AC_MSG_ERROR([Nanos6 library not found,
is needed to enable OmpSs2.])
])

AC_CHECK_LIB([custom-nanos6],
[nanos6_library_mode_init],
[],
[
AC_MSG_ERROR([nanos6-library-mode.o is needed,
we make a custom library from it.])
])
]
)
```

Código C.8: Macros necesarias en configure.ac para habilitar la opcion –ompss-2 en configure.

En este caso a parte de la macro *AC_ARG_WITH* se utiliza *AC_CHECK_LIB*, esta macro hará que se compruebe si la librería del primer argumento existe y contiene la función del segundo argumento, el tercer y el cuarto argumento indican respectivamente que se hará si se encuentra la librería y que se hará en caso que no se encuentre. Nótese que esta macro, añade la librería a la variable de entorno *LIBS* (librerías que tomarán parte en la fase de *linking*). En el caso de cuda debemos buscar la librería del *runtime* de *Cuda*, que es *cudart*. Se utilizan exactamente los mismos mecanismos que en el resto de opciones, se muestra exactamente en la imagen C.9.

```
1  AC_ARG_WITH([cuda],
2  AS_HELP_STRING([ --with-cuda], [Enables the use of CUDA.]),
3  [
4  cuda=true
5
6  AC_CHECK_LIB([cudart],
7  [cudaMalloc],
8  [],
9  [
10 AC_MSG_ERROR([cudart library not found,
11 is needed to enable CUDA.])
12 ])
13 ]
14 )
```

Código C.9: Macros necesarias en configure.ac para habilitar la opcion –cuda en configure.

Para todas las opciones, en las imágenes C.7, C.8 y C.9 se asigna el valor a una variable ompss, ompss-2 y cuda respectivamente para cada opción al valor *true*. Con la macro *AC_ CONDITIONAL* asignamos a una variable con el nombre del primer argumento el valor que obtengamos del segundo. Estas variables son visibles en el fichero *Makefile.am*, de manera que podremos condicionar la compilación en función de que opciones pase el usuario al *script configure*.

```
1  AM_CONDITIONAL([OMPSS]        , [test x$ompss = xtrue])
2  AM_CONDITIONAL([OMPSS2]       , [test x$ompss2 = xtrue])
3  AM_CONDITIONAL([CUDAOMPSS]    , [test x$cuda = xtrue &&
4  test x$ompss = xtrue])
5  AM_CONDITIONAL([CUDAOMPSS2]   , [test x$cuda = xtrue &&
6  test x$ompss2 = xtrue])
```

Código C.10: Creación de las variables para el Makefile.am.

Esto sería todo lo relacionado con el *configure.ac*, ahora entraríamos con el *Makefile.am*. Recordemos que lo único que necesitamos compilar con *--ompss-2* es el fichero que contiene las funciones a ser ejecutadas como tareas, por lo tanto necesitamos hacer una distinción de los *flags* que se utilizan para compilar este fichero. Para conseguir esto, hemos decidido indicar en el *Makefile.am* que se haga una librería de este fichero y se compile con unos *flags* en especial, para así aislarlo del resto. En la siguiente imagen se muestra como se hace esto.

La primera línea indica que *libfunctions.a* es una librería que no debe ser instalada, y la segunda que el fichero fuente de esta librería es *PACKAGE-functions.cc* donde *PACKAGE* es el nombre del paquete. Las siguientes líneas indican los *flags* base que se utilizarán para compilar los ficheros fuente.

```
1  noinst_LIBRARIES = libfunctions.a
2  libfunctions_a_SOURCES = PACKAGE-functions.cc
3
4  libfunctions_a_CPPFLAGS =  -I../../src -I../../include
5  -Wno-write-strings -I$(JAVA_HOME)/include
6  -I$(JAVA_HOME)/include/linux/ -Wall
7  -I$(CS_HOME)/../bindings-common/include
8  -I$(CS_HOME)/include
9
10 libfunctions_a_CFLAGS =
```

Código C.11: Crear una librería en el Makefile.am.

A esta altura, ya tenemos la base para compilar la librería, pero si no podemos poner los *flags* para *OmpSs-2* o *OmpSs* no habremos conseguido nada. Utilizaremos las variables que hemos

definido anteriormente en C.10. Sencillamente, en función de si las variables correspondientes tienen el valor adecuado, añadimos a las variables adecuadas los *flags* que requerimos. Por ejemplo, si el usuario está compilando con *OmpSs-2*, los *flags* para compilar *libfunctions.a* deben contener *--ompss-2* entre otros. En la imagen C.12 se procede a asignar los *flags* de esta manera.

```
1  if OMPSS2
2  libfunctions_a_CPPFLAGS  += --ompss-2 -DOMPSS2_ENABLED
3  libfunctions_a_CFLAGS    += --ompss-2 -DOMPSS2_ENABLED
4
5  nio_worker_c_CPPFLAGS    += -DOMPSS2_ENABLED
6  worker_c_CPPFLAGS        += -DOMPSS2_ENABLED
7  endif
8
9  if OMPSS
10 libfunctions_a_CPPFLAGS  += --ompss -DOMPSS_ENABLED
11 libfunctions_a_CFLAGS    += --ompss -DOMPSS_ENABLED
12
13 nio_worker_c_CPPFLAGS    += --ompss -DOMPSS_ENABLED
14 worker_c_CPPFLAGS        += --ompss -DOMPSS_ENABLED
15
16 nio_worker_c_LDFLAGS += --ompss
17 worker_c_LDFLAGS      += --ompss
18 endif
19
20 if CUDAOMPSS
21 libfunctions_a_CPPFLAGS  += --cuda -DCUDA_ENABLED
22 libfunctions_a_CFLAGS    += --cuda -DCUDA_ENABLED
23
24 nio_worker_c_CPPFLAGS    += --cuda -DCUDA_ENABLED
25 worker_c_CPPFLAGS        += --cuda -DCUDA_ENABLED
26
27 nio_worker_c_LDFLAGS += --cuda
28 worker_c_LDFLAGS      += --cuda
29 endif
30
31 if CUDAOMPSS2
32 libfunctions_a_CPPFLAGS  += --cuda -DCUDA_ENABLED
33 libfunctions_a_CFLAGS    += --cuda -DCUDA_ENABLED
34 endif
```

Código C.12: Flags de manera condicional en Makefile.am.

Está claro que después de haber compilado la librería habrá que hacer el *linking* de ella con *nio_worker_c* o *worker_c*. Con esto ya habríamos hecho las modificaciones necesarias para poder compilar la aplicación de *COMPSs+OmpSs-2*.

C.3. Soporte para el tipo enum y cabeceras en la interfaz

Un requisito ha sido soportar el tipo *enum* y añadir la posibilidad de incluir cabeceras en la interfaz de una aplicación *C/C++*. Para hacer esto, ha habido que hacer modificaciones en la gramática del compilador y en la generación de código. Este añadido puede ser muy útil para cuando se utilicen librerías externas en la interfaz y se precise de cabeceras adicionales que incluyan tipos que se necesitan.

El compilador está hecho con las herramientas *Lex* (un generador de analizadores léxicos) y *Yacc* (un generador de analizadores sintácticos), es decir, las dos piezas fundamentales para hacer un compilador, ya que uno nos facilitará el reconocimiento de *tokens* en la interfaz y el otro nos permite comprobar que realmente tiene sentido la interfaz y le otorga una semántica.

Para añadir el tipo *enum*, la gramática tiene que reconocer algo del estilo "*in/out enum* tipo nombre_variablez asociarle el comportamiento que el compilador tiene con el tipo entero al del *enum* ya que al final ambos son enteros. De forma similar, para el *include* hubo modificar la gramática para que reconociera el patrón "*include* nombre_cabecera;z al generar el código incluir la cabecera.

En cuánto las modificaciones a hacer en *Lex* son relativas a los *tokens* que el analizador deberá detectar, la siguiente imagen muestra las modificaciones.

```
1  ...
2  header_identfier  [a-zA-Z][0-9a-zA-Z_]*\.[a-z]*
3  %%
4  ...
5  enum            return TOK_ENUM;
6  include         return TOK_INCLUDE;
7  {header_identfier} yylval.name = strdup(yytext); return TOK_HEADER;
8  %%
```

Código C.13: Porción del analizador sintáctico para soportar tipo enum y interface.

Se define *header_identifier* como una expresión regular que identificará correctamente cualquier cabecera del estilo que hemos dicho antes. A continuación se define qué *token* devolverán *enum include* y *header_identifier*. Ahora podremos utilizar esto en *Yacc*.

La siguiente imagen muestra todas las modificaciones hechas en la gramática para dar el soporte. Básicamente se añaden a una estructura de datos el *TOK_HEADER* (*include*) y el *enum_type* que es una regla definida por el *token TOK_ENUM* (*enum*), se crea la regla *includes* con las características descritas y ejecuta la función entre llaves *add_header*, además se añade otra regla nueva a *argument*, que son todos los posibles aspectos (tipo, dirección...) que argumento puede tener en una tarea. Esta nueva regla reconoce una dirección (entrada o salida), un *enum_type*, y dos *TOKEN_IDENTIFIER* que corresponden con el nombre del *enum*, es decir el tipo en sí, y el nombre del argumento. Una vez reconocido este patrón se ejecuta la función entre llaves *add_argument*.

```
1 % union {
2 char         *elements;
3 char         *name;
4 char         *classname;
5 enum datatype    dtype;
6 enum direction   dir;
7 }
8 ...
9 % token TOK_ENUM TOK_HEADER
10
11 % token <name> TOK_IDENTIFIER TOK_HEADER
12 % token <elements> NUMBER
13 % type <dtype> data_type numeric_type array_type enum_type
14 % type <dir> direction
15 ...
16 %%
17 enum_type: TOK_ENUM { $$ = enum_dt; }
18 ;
19 ...
20 includes:
21 | includes TOK_INCLUDE TOK_HEADER { add_header($3); } TOK_SEMICOLON
22 ;
23
24 argument:    direction data_type TOK_IDENTIFIER { add_argument($1, $2, "", $3, NULL); }
25
26 |    direction array_type TOK_LEFT_BRAKET TOK_IDENTIFIER TOK_RIGHT_BRAKET TOK_IDENTIFIER
27                          { add_argument($1, $2, "", $6, $4);}
28
29 |    direction array_type TOK_LEFT_BRAKET NUMBER TOK_RIGHT_BRAKET TOK_IDENTIFIER
30                          { add_argument($1, $2, "", $6, $4);}
31
32 |    direction TOK_IDENTIFIER TOK_IDENTIFIER
33                          { add_argument($1, object_dt, $2, $3, NULL); }
34
35 |    direction enum_type TOK_IDENTIFIER TOK_IDENTIFIER
36                          { add_argument($1, $2, $3, $4, NULL); }
37 ;
38 %%
39 ...
```

Código C.14: Porción de la gramática para soportar tipo enum y interface.

Con el fragmento de la imagen C.14 y el resto del código que no aparece, *Yacc* generará un fichero *C* que es la implementación del analizador sintáctico.

Las funciones *add_header* y *add_argument* están implementadas en C (junto a otras muchas funciones que no veremos), y hacen uso de estructuras de datos que se utilizarán luego al generar el código, la fase que estamos viendo nos permite crear estructuras de datos que representen correctamente la interfaz y serán las que determinen en el paso de generar el código cómo queda este. La imagen C.15 contiene la definición de las estructuras de dato que se irán repitiendo en lo queda de sección.

```
1  struct argument {
2  char *name;
3  char *classname;
4  enum datatype   type;
5  enum direction  dir;
6  enum stream     stream;
7  int passing_in_order;
8  int passing_out_order;
9  char *elements;
10 argument *next_argument;
11 };
12
13 struct constraint {
14 char *name;
15 constraint *next_constraint;
16 };
17
18 struct include {
19 char *name; //Name of the header to include
20 include *next_include;
21 };
22
23 struct function {
24 char *name;
25 int access_static;
26 char *methodname;
27 char *classname;
28 char *return_typename;
29 enum datatype return_type;
30 char *return_elements;
31 argument *first_argument;
32 int argument_count;
33 int exec_arg_count;
34 constraint *first_constraint;
35 function *next_function;
36 };
37
38 struct interface {
39 char *name;
40 function *first_function;
41 };
```

Código C.15: Definición de las estructuras de datos utilizadas en el compilador.

Cuando se llama a *add_header* en C.14 se le pasa el nombre de la cabecera a incluir, si es la primera vez que se ha incluido una cabecera en la interfaz habrá que asignar memoria para la variable *current_include* del tipo *include* y asignarle el nombre de la cabecera. En caso que no fuera la primera vez, se encadenaría la nueva inclusión a la estructura *include* actual en el campo *next_include*. De esta manera tendríamos siempre un puntero a la primera cabecera y en caso de haber más se apuntarían una detrás de otra. La imagen C.16 contiene el código de la función *add_header*.

```
1  void add_header(char* name) {
2      debug_printf("Adding header %s\n", name);
3
4      if (current_include == NULL) {
5          current_include = (include*) malloc(sizeof(include));
6      }
7      else {
8          current_include->next_include = (include*) malloc(sizeof(include));
9          current_include = current_include->next_include;
10     }
11     current_include->name = name;
12
13     if (first_include == NULL) {
14         first_include = current_include;
15     }
16 }
```

Código C.16: Función add_header.

Cada vez que se reconoce un argumento se ejecuta la función *add_argument*, que se encargará de crear las estructuras *argument* y añadirlas a la estructura *current_function*, que es la función que está siendo procesada actualmente, así cuando se hayan procesado todos los argumentos de la función actual está tendrá punteros a todas las estructuras que representan sus argumentos. En la imagen C.17 se han omitido partes del *switch* que correspondían a otros tipos de dato, en cualquier caso el código que se muestra basta para saber cómo se hace con el tipo *enum*.

```
1  void add_argument(enum direction dir, enum datatype dt, char *classname, char *name, char* elements) {
2    argument *new_argument;
3    debug_printf("Add argument %i %i %s %s %s \n", dir, dt, classname, name, elements);
4    assert(current_function != NULL);
5    if (exists_argument(name)) {
6      fprintf(stderr, "%s:%i: Duplicated argument name '%s' in function '%s'\n",
7      get_filename(), line, name, get_current_function_name());
8      has_errors = 1;
9      return;
10   }
11   if (dt == void_dt) {
12     fprintf(stderr, "%s:%i: Invalid parameter type for argument %i in function '%s'\n",
13     get_filename(), line, get_next_argnum(), get_current_function_name());
14     has_errors = 1;
15     return;
16   }
17   new_argument = (argument *)malloc(sizeof(argument));
18   new_argument->name = strdup(name);
19
20     switch (dt) {
21
22     ...
23
24     case enum_dt:
25       new_argument->elements = "0";
26       new_argument->type = enum_dt;
27       new_argument->classname = strdup(classname);
28       break;
29
30     ...
31
32     }
33
34     new_argument->dir = dir;
35     new_argument->passing_in_order = 0;
36     new_argument->passing_out_order = 0;
37     new_argument->next_argument = NULL;
38
39     if (current_argument != NULL) {
40       current_argument->next_argument = new_argument;
41     }
42     else {
43       current_function->first_argument = new_argument;
44     }
45     current_argument = new_argument;
46     current_function->argument_count++;
47
48     switch (dir) {
49     case in_dir:
50     case out_dir:
51       current_function->exec_arg_count++;
52       break;
53     case inout_dir:
54       current_function->exec_arg_count += 2;
55       break;
56     default:
57       break;
58     }
59 }
```

Código C.17: Función add_argument.

Ahora cuando vayamos a generar código ya tendremos en cuenta el *enum* y el añadido de cabeceras. El *enum* se trata igual que el *int*, pero para el caso de las cabeceras se ha tenido que añadir una nueva función a la generación de código.

```
1  static void include_header(include* current_include) {
2      fprintf(includeFile, "#include <%s>\n", current_include->name);
3  }
4
5  static void add_include_headers(include* first_include) {
6      include* include = first_include;
7
8      while (include != NULL) {
9          include_header(first_include);
10         include = include->next_include;
11     }
12 }
```

Código C.18: Funciones para incluir las cabeceras.

En la imagen C.18 están las funciones *include_header* y *add_include_headers*, la primera dada una estructura *include* que contiene el nombre de la cabecera a incluir imprime en el *includeFile* la macro para incluirla, y la segunda dado el primer *include* los recorre e incluye todos.

Apéndices D

Código aplicaciones

En este apéndice se incluye el código principal y la interfaz de las aplicaciones de *COMPSs* que se utilizan para evaluar la integración con *OmpSs-2*.

D.1. K-Means

```
1
2  void kmeans(int numClusters,
3              int numFrags,
4              int objsFrag,
5              int numCoords,
6              int loop_iteration,
7              char* filePath,
8              int isBinaryFile,
9              int is_output_timing)
10 {
11     printf("Execution: K = %d,
12             Frags = %d, NpF = %d,
13             d = %d, max_iters = %d\n",
14             numClusters, numFrags, objsFrag, numCoords, loop_iteration);
15
16     int  i, j, index, loop=1;
17     double  timing, io_timing, clustering_timing;
18
19     if (numCoords<1){
20         fprintf(stderr,"Error reading number of coordinates");
21         exit(-1);
22     }
23     srand(1000);
24
25     float *clusters = (float*) malloc(numClusters*numCoords*sizeof(float));
26     for (i=0; i<numClusters; i++){
27         for (j=0; j<numCoords; j++){
28             clusters[i*numCoords+j] = (float) ((rand()) / (float)((RAND_MAX))*2 - 1);
29         }
30
31     int **newClusterSize = (int **) malloc(numFrags*sizeof(int*));
32     float **newClusters = (float **) malloc(numFrags*sizeof(float*));
33     int **frag_index = (int **)malloc(numFrags*sizeof(int*));
34     float **fragments = (float **)malloc(numFrags*sizeof(float*));
35
36     for (j=0; j<numFrags; j++){
37         frag_index[j] = (int*) malloc(sizeof(int));
```

```
38          newClusters[j] = (float*) malloc(numClusters*numCoords*sizeof(float));
39          newClusterSize[j]=(int*) malloc(numClusters* sizeof(int));
40          for (i=0; i<numClusters; i++){
41           newClusterSize[j][i]=0;
42          }
43      }
44      compss_on();
45
46      if (is_output_timing) io_timing = wtime();
47
48      for (i=0; i<numFrags; i++){
49          fragments[i] = init_Fragment(objsFrag,
50                                       numCoords,
51                                       objsFrag*numCoords,
52                                       filePath,
53                                       frag_index[i]);
54      }
55      compss_barrier();
56
57      if (is_output_timing) {
58          timing            = wtime();
59          io_timing         = timing - io_timing;
60          clustering_timing = timing;
61      }
62
63      do {
64          for (i=0; i<numFrags; i++){
65              compute_newCluster(objsFrag, numCoords,
66                                 numClusters, objsFrag*numCoords,
67                                 numClusters*numCoords,
68                                 frag_index[i], fragments[i],
69                                 clusters, newClusters[i],
70                                 newClusterSize[i]);
71          }
72          int neighbor = 1;
73          while (neighbor < numFrags) {
74              for (int f = 0; f < numFrags; f += 2 * neighbor) {
75                  if (f + neighbor < numFrags) {
76                      merge_newCluster(numCoords, numClusters,
77                                       numClusters*numCoords,
78                                       newClusters[f],
79                                       newClusters[f+neighbor],
80                                       newClusterSize[f],
81                                       newClusterSize[f+neighbor]);
82                  }
83              }
84              neighbor *= 2;
85          }
86          update_Clusters(numCoords, numClusters,
87                          numClusters*numCoords,
88                          clusters, newClusters[0],
89                          newClusterSize[0]);
90
91      } while (loop++ < loop_iteration);
92      compss_barrier();
93
94      if (is_output_timing) {
95          timing            = wtime();
```

```
96          clustering_timing = timing - clustering_timing;
97          printf("\nPerforming **** Regular Kmeans (compss version) ****\n");
98          printf("Input file:      %s\n", filePath);
99          printf("numFrags      =  %d\n", numFrags);
100         printf("numClusters   = %d\n", numClusters);
101         printf("Loop iterations    = %d\n", loop_iteration);
102         printf("I/O time           = %10.4f sec\n", io_timing);
103         printf("Computation timing = %10.4f sec\n", clustering_timing);
104     }
105     compss_off();
106 }
```

Código D.1: Programa principal de la aplicación K-Means.

```
1  interface kmeans{
2
3          @Constraints(ComputingUnits=5);
4          float[size] init_Fragment(in int numCoords,
5                                    in int numObjs,
6                                    in int size,
7                                    in File filename,
8                                    out int[1] ff);
9
10         @Constraints(ComputingUnits=150);
11         void compute_newCluster(in int objsFrag, in int numCoords,
12                                 in int numClusters, in int sizeFrags,
13                                 in int sizeClusters, in int[1] ff,
14                                 in float[sizeFrags] frag,
15                                 in float[sizeClusters] clusters,
16                                 out float[sizeClusters] newClusters,
17                                 out int[numClusters] newClustersSize);
18
19         @Constraints(processors={@Processor(ProcessorType=GPU,
20                                             ComputingUnits=1)});
21         @Implements(compute_newCluster);
22         void compute_newCluster_GPU(in int objsFrag, in int numCoords,
23                                     in int numClusters, in int sizeFrags,
24                                     in int sizeClusters, in int[1] ff,
25                                     in float[sizeFrags] frag,
26                                     in float[sizeClusters] clusters,
27                                     out float[sizeClusters] newClusters,
28                                     out int[numClusters] newClustersSize);
29
30         void merge_newCluster(in int numCoords, in int numClusters,
31                               in int sizeClusters,
32                               inout float[sizeClusters] clusters1,
33                               in float[sizeClusters] clusters2,
34                               inout int[numClusters] newClustersSize1,
35                               in int[numClusters] newClustersSize2);
36
37         void update_Clusters(in int numCoords, in int numClusters,
38                              in int sizeClusters,
39                              inout float[sizeClusters] clusters1,
40                              in float[sizeClusters] clusters2,
41                              in int[numClusters] newClustersSize2);
42 };
```

Código D.2: Programa principal de la aplicación Cholesky.

D.2. Cholesky

```
1  int main(int argc, char** argv) {
2
3      int MSIZE = atoi(argv[1]);
4      int BSIZE = atoi(argv[2]);
5
6      compss_on();
7
8      struct timeval start_io, end_io, start_comp, end_comp;
9      double io_timing, cholesky_timing;
10
11     gettimeofday(&start_io, NULL);
12
13     double** MATRIX = (double**)malloc(sizeof(double*)*MSIZE*MSIZE);
14
15     for (int i = 0; i < MSIZE*MSIZE; ++i) {
16       MATRIX[i] = (double*)malloc(sizeof(double)*BSIZE*BSIZE);
17     }
18
19    for (int i = 0; i < MSIZE; i++) {
20       generate_block(MATRIX[i*MSIZE+i], BSIZE, 1);
21       for (int j = i+1; j < MSIZE; j++) {
22             generate_block(MATRIX[i*MSIZE+j], BSIZE, 0);
23             generate_block(MATRIX[j*MSIZE+i], BSIZE, 0);
24         }
25     }
26
27     compss_barrier();
28
29     gettimeofday(&end_io, NULL);
30
31     io_timing = (end_io.tv_sec - start_io.tv_sec) * 1e6;
32     io_timing = (io_timing + (end_io.tv_usec - start_io.tv_usec)) * 1e-6;
33
34     gettimeofday(&start_comp, NULL);
35
36     for (int k = 0; k < MSIZE; ++k) {
37         ddss_dpotrf(Lower, BSIZE, MATRIX[k*MSIZE+k], BSIZE);
38
39         for (int i = k+1; i < MSIZE; ++i) {
40             ddss_dtrsm(Right, Lower,
41                        Trans, NonUnit,
42                        BSIZE, BSIZE,
43                        BSIZE, MATRIX[k*MSIZE+k], BSIZE,
44                        MATRIX[i*MSIZE+k], BSIZE);
45         }
46
47         for (int i = k+1; i < MSIZE; ++i) {
48
49             ddss_dsyrk(Lower, NoTrans,
50                        BSIZE, BSIZE,
51                        BSIZE, MATRIX[i*MSIZE+k], BSIZE,
52                       -BSIZE, MATRIX[i*MSIZE+i], BSIZE);
53
54             for (int j = i; j < MSIZE; ++j) {
55
56                 ddss_dgemm(NoTrans, Trans,
```

```
57                                  BSIZE, BSIZE, BSIZE,
58                                  BSIZE, MATRIX[i*MSIZE+k], BSIZE,
59                                  MATRIX[j*MSIZE+k], BSIZE,
60                                  -BSIZE, MATRIX[i*MSIZE+j], BSIZE);
61              }
62          }
63      }
64
65      compss_barrier();
66
67      gettimeofday(&end_comp, NULL);
68
69      cholesky_timing = (end_comp.tv_sec - start_comp.tv_sec) * 1e6;
70      cholesky_timing = (cholesky_timing +
71                         (end_comp.tv_usec - start_comp.tv_usec)) * 1e-6;
72
73      printf("\nPerforming **** Cholesky ****\n");
74      printf("I/O time           = %10.4f sec\n", io_timing);
75      printf("Computation timing = %10.4f sec\n", cholesky_timing);
76
77      compss_off();
78  }
```

Código D.3: Programa principal de la aplicación Cholesky.

```
1   include lass.h;
2
3   interface cholesky
4   {
5
6       @Constraints(ComputingUnits=2);
7       void generate_block(inout double[N*N] A, in int N, in int D);
8
9       @Constraints(ComputingUnits=4);
10      int ddss_dpotrf(in enum DDSS_UPLO UPLO,
11                      in int N,
12                      inout double[N*LDA] A,
13                      in int LDA);
14
15      @Constraints(ComputingUnits=4);
16      int ddss_dtrsm( in enum DDSS_SIDE SIDE, in enum DDSS_UPLO UPLO,
17                      in enum DDSS_TRANS TRANS_A,
18                      in enum DDSS_DIAG DIAG,
19                      in int M, in int N,
20                      in double ALPHA, in double[N*N] A, in int LDA,
21                      inout double[M*N] B, in int LDB);
22
23      @Constraints(ComputingUnits=4);
24      int ddss_dgemm( in enum DDSS_TRANS TRANS_A,
25                      in enum DDSS_TRANS TRANS_B,
26                      in int M, in int N, in int K,
27                      in double ALPHA, in    double[M*K] A, in int LDA,
28                      in double[K*N] B, in int LDB,
29                      in double BETA,  inout double[M*N] C, in int LDC );
30
31      @Constraints(ComputingUnits=4);
32      int ddss_dsyrk( in enum DDSS_UPLO UPLO, in enum DDSS_TRANS TRANS,
33                      in int N, in int K,
34                      in double ALPHA, in    double[N*K] A,
```

```
35                          in int LDA, in double BETA,
36                          inout double[N*N] C, in int LDC );
37
38  };
```

Código D.4: Interfaz de la aplicación Cholesky.

Apéndices E

Limitaciones COMPSs+OmpSs-2: código del test

```
1  #include <iostream>
2  #include <c_compss.h>
3  #include "tlt.h"
4  using namespace std;
5
6  int main() {
7      cout << "Encendemos el runtime... " << endl;
8
9      compss_on();
10
11     int* array_a = (int*) malloc(30 * sizeof(int));
12     int* array_b = (int*) malloc(30 * sizeof(int));
13
14     for (int i = 0; i < 30; ++i) {
15
16         coarse_task(array_a, 30);
17
18         fine_task(array_b, 30);
19
20     }
21
22     compss_wait_on(array_a);
23     compss_wait_on(array_b);
24
25     cout << "Apagamos el runtime... "   << endl;
26     compss_off();
27
28 }
```

Código E.1: Programa principal del test

```
1  interface tlt {
2
3  void coarse_task(inout int[N] a, in int N);
4
5  void fine_task(inout int[N] a, in int N);
6
7  };
```

Código E.2: Interfaz de la aplicación de test

```c
#include <unistd.h>
#include <stdio.h>

#define COARSE_GRAIN_SLEEP 300000
#define FINE_GRAIN_SLEEP 50000

void coarse_task(int* a, int N) {

    int i;
    for (i = 0; i < N; ++i) {
        #pragma oss task in([N]a) label(COARSE_GRAIN_TASK)
        {
        a[i] = i;
        usleep(COARSE_GRAIN_SLEEP);
        }
    }
    #pragma oss taskwait
}

void fine_task(int* a, int N) {

    int i;
    for (i = 0; i < N; ++i) {
        #pragma oss task in([N]a) label(FINE_GRAIN_TASK)
        {
        a[i] = i;
        usleep(FINE_GRAIN_SLEEP);
        }
    }
    #pragma oss taskwait
}
```

Código E.3: Funciones a ejecutar como tareas del test

Bibliografía

[1] Berkeley Unified Parallel C. https://upc.lbl.gov/. [Online; accedido 19-Junio-2019].

[2] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[3] Rosa M Badia, Javier Conejero, Carlos Diaz, Jorge Ejarque, Daniele Lezzi, Francesc Lordan, Cristian Ramon-Cortes, and Raul Sirvent. Comp superscalar, an interoperable programming framework. *SoftwareX*, 3:32–36, 2015.

[4] Blaise Barney. Posix threads programming, 2009.

[5] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *Computing in Science & Engineering*, (1):46–55, 1998.

[6] Vicerectorat de Política Universitària. ASSIGNACIÓ D'ESPAIS DE L'EDIFICI K2M. https://wwwbupc.webs.upc.edu/bupc/hemeroteca/2008/b107/05-06-2008.pdf. [Online; accedido 12-Marzo-2019].

[7] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[8] Comissió d'Economia i Infraestructures. Modificació de les tarifes dels espais Parc UPC. https://govern.upc.edu/ca/consell-de-govern/consell-de-govern/sessio-3-2017-de-consell-de-govern/aprovacio-de-la-modificacio-de-les-tarifes-dels-espais-parc-upc/9-11-modificacio-de-les-tarifes-desl-espais-parc-upc.pdf/@@display-file/visiblefile/9.11%20Modificaci%C3%B3%20de%20les%20tarifes%20desl%20espais%20Parc%20UPC.pdf. [Online; accedido 12-Marzo-2019].

[9] Antonio Filgueras, Eduard Gil, Carlos Alvarez, Daniel Jimenez, Xavier Martorell, Jan Langer, and Juanjo Noguera. Heterogeneous tasking on smp/fpga socs: The case of ompss and the zynq. In *2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 290–291. IEEE, 2013.

[10] BSC Programming Models Group. OmpSs-2 Specification. https://pm.bsc.es/ftp/ompss-2/doc/spec/OmpSs-2-Specification.pdf. [Online; accedido 25-Febrero-2019].

[11] Daniel Grünewald and Christian Simmendinger. The gaspi api specification and its implementation gpi 2.0. In *7th International Conference on PGAS Programming Models*, page 243, 2013.

[12] David Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*, pages 836–838. IEEE, 2008.

[13] Florentino Sainz, Sergi Mateo, Vicenç Beltran, Jose L Bosque, Xavier Martorell, and Eduard Ayguadé. Leveraging ompss to exploit hardware accelerators. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pages 112–119. IEEE, 2014.

[14] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[15] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. Pycompss: Parallel computational workflows in python. *The International Journal of High Performance Computing Applications*, 31(1):66–82, 2017.

[16] David W Walker and Jack J Dongarra. Mpi: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.