Extending OmpSs to support dynamic programming

Autor
Marcos Maroñas Bravo

Director
Vicenç Beltran Querol (BSC-CNS)

Co-director
Eduard Ayguadé Parra (DAC)

Ponente
Agustín Fernández Jiménez (DAC)

Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS)
Departamento de Arquitectura de Computadores (DAC)

Grado en Ingeniería Informática

Especialidad
Ingeniería de computadores

28 de Enero de 2015

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Agradecimientos

Me gustaría agradecer a mi director, Vicenç Beltrán, por ofrecerme la oportunidad de trabajar en el BSC e involucrarme en proyectos motivantes. Gracias por creer en mí, tener en cuenta mis ideas y guiarme en los momentos necesarios.

También me gustaría darle las gracias a mi ponente, Agustín Fernández, por el tiempo dedicado a revisar mi trabajo y todos los comentarios que me han ayudado a mejorar mi trabajo. También quiero agradecer la confianza depositada en mí.

Este trabajo hubiera sido mucho más difícil sin el apoyo y soporte de Sergi Mateo, quien me ha ayudado en todo lo que he necesitado durante el desarrollo del proyecto. Muchas gracias, Sergi.

También quiero agradecer al resto de compañeros del BSC la cálida acogida al grupo y todos sus comentarios de ánimo y ayuda, así como el soporte ofrecido cuando ha sido necesario.

Por supuesto, quiero agradecer especialmente a mi madre por apoyarme y creer en mí en todo lo que he hecho. Gracias por la plena confianza.

Por último, no puedo olvidarme de todos mis amigos y familia que me han apoyado en todo momento.

Resumen

Tras un breve resumen sobre programación dinámica y OmpSs, este documento muestra como OmpSs ofrece soporte para memoización y los resultados obtenidos. Los cambios realizados para poder soportar esta técnica de programación dinámica en OmpSs se pueden dividir en la parte del compilador (Mercurium) y la parte del sistema de *runtime* (Nanos++), además de la adición de nueva sintaxis al modelo. Primero, una nueva cláusula, **memo** ha sido añadida al modelo. Usando esta cláusula, el desarrollador pueda demandar la memoización con sólo indicar algunos parámetros acompañando a la cláusula. Una vez el compilador ha leído la cláusula, genera algo de código adicional, mayoritariamente llamadas al sistema de *runtime*, para darle información. Entonces, con la información que recibe de Mercurium, Nanos++ prepara la infraestructura para realizar la memoización. Con este enfoque, los desarrolladores ya no deben temer el uso de la memoización ya que no será más esa técnica que convierte su código en algo ilegible e imposible de mantener. Sin embargo, los beneficios son grandes. En algunos problemas representativos de programación dinámica como el cálculo de la sucesión de Fibonacci o el problema de la mochila obtenemos una ganancia de más de 25000x y 600x respectivamente.

Abstract

After a brief overview of dynamic programming and OmpSs, this document shows how OmpSs supports memoization and the results obtained. The changes done to support dynamic programming in OmpSs can be divided in compiler (Mercurium) side, and runtime (Nanos++) side, besides, of course, adding new syntax to the model. Firstly, a new **memo** clause has been added to the model. Using that clause, the developer requests memoization just indicating some parameters. Once the compiler reads the clause, it generates some extra code, mainly runtime calls, to give information to the runtime system. Then, with the information received from Mercurium, Nanos++ prepares the infrastructure to perform memoization. Using this approach, developers should not be afraid to use memoization since it would not be anymore the technique that turns out their code unreadable and unmaintainable. However, the benefits are big. In some representative dynamic programming benchmarks like Fibonacci sequence calculation or Knapsack problem we get more than 25000x and 600x speedup respectively.

Resum

Després d'un breu resum sobre programació dinàmica i OmpSs, aquest document mostra com OmpSs ofereix suport per memoització i els resultats obtinguts. Els canvis realitzats per a poder suportar aquesta tècnica de programació dinàmica a OmpSs es poden dividir en la banda del compilador (Mercurium) i la banda del sistema de *runtime*, a més a més de l'addició de nova sintaxi al model. Primer, una nova clàusula, **memo** ha estat afegida al model. Fent servir aquesta clàusula, el desenvolupador pot demanar la memoització només indicant alguns paràmetres juntament amb la clàusula. Un cop el compilador ha llegit la clàusula, genera una mica de codi addicional, la majoria crides al sistema de *runtime*, per passar-li informació. Aleshores, amb la informació que rep de Mercurium, Nanos++ prepara la infraestructura per a realitzar la memoització. Amb aquest enfocament, els desenvolupadors ja no han de tenir por de fer servir la memoització ja que no serà mai més una tècnica que converteixi el codi en quelcom il·legible i impossible de mantenir. No obstant això, els benificis són grans. En alguns problemas representatius de programació dinàmica com ara el càlcul de la successió de Fibonacci o el problema de la motxilla obtenim un guany de més de 25000x i 600x respectivament.

Índice

Índice de figuras

Índice de cuadros

1. Introducción

En la actualidad, la mayoría de dispositivos electrónicos cuentan con multiprocesadores. Desde los supercomputadores hasta los teléfonos móviles los incorporan. En consecuencia, es necesario usar técnicas de paralelismo que nos permitan exprimir al máximo esos recursos de una forma eficiente.

Por otro lado, las técnicas de programación dinámica pueden mejoran notablemente el rendimiento de un subconjunto de las aplicaciones resueltas con un esquema de divide y vencerás (*divide&conquer*). Sin embargo, no existen muchas aproximaciones que combinen programación dinámica y paralelismo.

Puesto que el uso de programación dinámica y paralelismo, por separado, ya consiguen grandes mejoras de rendimiento, parece lógico pensar que al combinarlas se podría mejorar todavía más.

1.1. Motivación

La motivación personal para realizar este trabajo surge del interés por los siguientes ámbitos:

- Computación de altas prestaciones
- Programación dinámica

El interés por la **computación de altas prestaciones** surge ya en la parte final de mis estudios. La primera asignatura que trata el paralelismo, llamada también así, Paralelismo (PAR), es fundamental para mi interés en el tema. En ella se tratan los aspectos fundamentales de la programación paralela. Se introducen los problemas que puede generar el uso de la concurrencia y como solucionarlos; la descomposición en tareas de una aplicación y la descomposición de datos. Pero lo que más llama mi atención es el incremento de rendimiento que se consigue usando estas técnicas. Más adelante, en la asignatura de Programación y Arquitecturas Paralelas, es cuando más crece mi interés sobre esta disciplina de la informática. En ella se profundiza más y se introduce el modelo de programación OmpSs. A estas alturas decido que quiero realizar investigación en este campo para mi trabajo final de grado.

Por otra parte, la **programación dinámica** es un método que, aunque se introduce en la asignatura Programación Consciente de la Arquitectura (PCA), yo no conocía, puesto que no pude cursar la asignatura. No obstante, algunos compañeros que sí pudieron cursarla me hablaron de este método de optimización y creí que mi trabajo final de grado sería perfecto para profundizar en el tema.

Además, la programación dinámica suele ser una tarea complicada y laboriosa. En consecuencia, facilitar esta tarea al desarrollador es necesario. Asimismo, la combinación de memoización y paralelismo es un tema sin tratar en la literatura. Más aún si se puede conseguir mediante directivas de compilador, que no afectan a la semántica de la aplicación, y un sistema de *runtime* que permiten paralelizar y memoizar una aplicación de forma incremental.

1.2. Contribuciones de este TFG

Este proyecto añade soporte para programación dinámica al modelo de programación OmpSs. Se presenta una herramienta que permite combinar técnicas de paralelismo y programación dinámica. Por si no fuera suficiente, OmpSs es un modelo basado en directivas de compilador, es

decir, pequeñas anotaciones de código. De este modo, conseguimos, tanto el paralelismo como la memoización (técnica de programación dinámica) con un impacto mínimo sobre el código de las aplicaciones.

En otras palabras, este proyecto presenta una herramienta que consigue memoización automática combinada con paralelismo mediante directivas de compilador. Por tanto, el esfuerzo del desarrollador para optimizar sus aplicaciones y el impacto sobre el código son mínimos.

1.3. Estructura del documento

Este documento contiene dos partes diferenciadas. La primera parte versa sobre la gestión del proyecto. En otras palabras, planificación, costes, etcétera. La segunda parte del documento, más técnica, se centra en los detalles de desarrollo e implementación. Después se concluye y se discute brevemente sobre trabajo futuro. Finalmente se hace una revisión del proyecto para adaptar las previsiones de planificación y costes a la realidad.

2. Contexto y estado del arte

En esta sección se detalla el contexto y los agentes involucrados, directa o indirectamente, en el proyecto. Además, se presentan las distintas aproximaciones que han trabajado sobre programación dinámica automática y/o la combinación de programación dinámica con paralelismo.

2.1. OmpSs

El modelo de programación paralela OmpSs es producto de la integración de diversas características de la familia de modelos de programación StarSs, desarrollado en el Barcelona Supercomputing Center - Centro Nacional de Supercomputación (de ahora en adelante BSC), en un único modelo de programación. Concretamente, OmpSs persigue extender OpenMP con nuevas directivas que permitan soportar paralelismo asíncrono basado en dependencias de datos y heterogeneidad (por ejemplo, GPUs) [oB14][Lab12][BM13].

Una de las características más destacadas de OmpSs es que extiende la descomposición en tareas de OpenMP 3.0 permitiendo añadir dependencias entre tareas. Mediante las dependencias el usuario puede definir el flujo de datos de su aplicación. De este modo, en tiempo de ejecución se pueden identificar condiciones de carrera entre tareas [oB13d].

Tanto OpenMP como OmpSs son modelos portables y escalables que ofrecen una interfaz sencilla y flexible para desarrollar aplicaciones paralelas en plataformas que van desde los sistemas empotrados hasta los sistemas de memoria compartida, pasando por sistemas multi-procesador o dispositivos aceleradores. Especifican un conjunto de directivas de compilador, rutinas de librerías y variables de entorno que permiten generar paralelismo de alto nivel en Fortran y C/C++ [ARB14][ARB13][dIdlUCIdM12].

El paralelismo se consigue indicando directivas al compilador, que son unas pequeñas anotaciones en el código, como las que se muestran en la Figura 1.

Un entorno válido de OmpSs sería el formado por el compilador Mercurium y el sistema de *runtime* Nanos++.

Listing 1: Directivas OpenMP/OmpSs

```
// Fragmento de codigo donde se resaltan las directivas OpenMP/OmpSs en rojo.
#pragma omp task
void foo ( int Y[size], int size ) {
  int j;
  for( j = 0; j < size; ++j )
    Y[j] = j;
}

int main () {
  int X[100];
  foo( X, 100 );
  #pragma omp taskwait
}
```

Figura 1: Fragmento de código donde se resaltan las directivas OpenMP/OmpSs.

2.1.1. Filosofía

Los principios de diseño de OpenMP y StarSs forman las ideas básicas usadas para concebir OmpSs, he ahí su nombre. El objetivo principal de OmpSs es ofrecer un entorno productivo en el que se puedan desarrollar aplicaciones para sistemas modernos de computación de altas prestaciones. Para hacer de OmpSs un entorno productivo se necesitan dos conceptos clave: **rendimiento** y **sencillez**. Por lo que respecta al rendimiento, OmpSs debe conseguir un rendimiento razonablemente comparable a otros modelos que usen las mismas arquitecturas. En cuanto a la sencillez, a pesar de ser un concepto difícil de valorar por no ser cuantificable, OmpSs ha sido diseñado siguiendo unos principios elogiados por su efectividad en esa área [oB13e][DAB+11].

De OpenMP, OmpSs toma la forma de realizar versiones paralelas de una aplicación introduciendo anotaciones en el código de la versión secuencial. La ventaja de estas anotaciones es que no tienen un efecto explícito en la semántica del programa. Sin embargo, el compilador es capaz de interpretarlas generando una versión paralela. Esta característica permite paralelizar aplicaciones de forma incremental: empezando desde la versión secuencial se van añadiendo directivas para paralelizar diversas partes del programa. Esta forma de implementar concurrencia tiene un gran impacto en la productividad, pues usando otros modelos más explícitos las aplicaciones deben ser rediseñadas para ser paralelizadas. Una consecuencia directa de estos modelos más explícitos es el esfuerzo creciente que se necesita para mantener, debugar o probar las aplicaciones [oB13e][DAB+11].

De StarSs, o Star SuperScalar toma el modelo de ejecución. StarSs es una familia de modelos de programación que, aunque también funcionan con anotaciones de código, difiere de OpenMP en ciertas áreas importantes. Mientras que OpenMP usa un modelo *fork-join*, StarSs usa un modelo *data-flow*. En el primero, la ejecución se hace con un sólo hilo hasta llegar a una región paralela: entonces se crean el resto de hilos que se vuelven a eliminar al término de la región paralela. En el segundo, los hilos se crean al principio de la ejecución y se destruyen al final. Durante la ejecución de regiones no paralelas simplemente esperan sin realizar trabajo [oB13e][DAB+11].

Además, StarSs también incluye características que permiten usar arquitecturas heterogéneas en contraposición con OpenMP que solo permite sistemas de memoria compartida hasta la versión 4.0, donde ya introduce el uso de aceleradores. Por último, StarSs también dispone de paralelismo asíncrono como mecanismo principal para expresar paralelismo. Mientras tanto, OpenMP empezó a añadir esta característica en la versión 3.0 [oB13e][DAB+11].

StarSs destaca en la cantidad de lo implícito que ofrece el modelo. Por ejemplo, al usar OpenMP el usuario debe encargarse, primero, de definir las regiones paralelas; después debe expresar el código de la región paralela y por último añadir directivas para sincronizar los distintos hilos. En comparación, StarSs simplifica este proceso: el paralelismo se crea implícitamente al principio de la ejecución. El código paralelo se define usando el concepto de tareas, que son trozos de código que pueden ejecutarse asíncronamente en paralelo. Por último, para la sincronización ofrece un mecanismo de dependencias que permite definir un orden para asegurar una ejecución correcta. Por todo ello, StarSs hace posible expresar el paralelismo de una forma más rica que la conseguida con OpenMP, explotando los recursos disponibles de una forma más eficiente [oB13e].

Por todo lo mencionado anteriormente, OmpSs pretende ser la evolución que OpenMP necesita para apuntar a nuevas arquitecturas. Es por eso que toma características clave de OpenMP combinándolas con ideas innovadoras ofrecidas por la familia StarSs.

Cabe destacar que parte de las ideas que se exploran en OmpSs acaban siendo añadidas al estándar OpenMP.

2.1.2. Expresando el paralelismo

La diferencia más notable entre OmpSs y OpenMP es que en el primero no es necesario el uso de la cláusula *parallel* para especificar una región paralela, en contraste con el segundo. Esto es debido a que OpenMP usa el modelo *fork-join* en el que es necesario definir el inicio y final del paralelismo, mientras OmpSs usa el modelo *thread-pool*. Los recursos paralelos pueden compararse con un grupo de hilos -he ahí su nombre, *thread-pool*- que el sistema de *runtime* puede usar durante la ejecución. Puesto que el usuario no tiene ningún control sobre este grupo de hilos, algunos métodos de OpenMP como, por ejemplo, *omp_get_num_threads()*, no están disponibles en OmpSs [oB13c][DAB+11].

El modelo OmpSs permite expresar el paralelismo mediante tareas. Las tareas son pequeñas partes de código, independientes, que pueden ser ejecutadas concurrentemente. Durante la ejecución del programa, si el flujo llega hasta una sección declarada como tarea, se creará una instancia de la tarea y se delegará la ejecución al *runtime* de OmpSs. La tarea podrá ser ejecutada en un recurso paralelo [oB13c].

Otra forma de expresar paralelismo en este modelo es usando la cláusula *for*. Esta cláusula existe tanto en OmpSs como en OpenMP y se comportan idénticamente. Se debe usar en un bucle for. Lo que hace es encapsular las iteraciones del bucle en tareas. Puesto que el número de tareas creadas está controlado por el modelo, se ofrece al usuario la cláusula *schedule* para especificar el tipo de planificación que quiere usar [oB13c].

Este modelo también permite definir múltiples niveles de paralelismos, ya que permite tener tareas dentro de otras tareas. Esta característica puede ser clave para mejorar el rendimiento de algunas aplicaciones debido a que el *runtime* de OmpSs puede explotar ciertos factores como la localidad temporal y espacial entre tareas. Cabe destacar, también, que esta funcionalidad permite implementar algoritmos recursivos [oB13c][DAB+11].

Por último, es necesario hablar de la sincronización. A menudo, las tareas usan datos calculados por otras tareas para realizar su trabajo. Consecuentemente, es necesario sincronizarlas para conseguir una ejecución correcta. OmpSs ofrece dos formas distintas de sincronización [oB13c][DAB+11]:

- **Dependencias de datos**. Se expresan con las cláusulas *in, out* o *inout*. Permite especificar qué datos necesita una tarea para empezar su ejecución, y qué datos están listos para ser usados al finalizar.
- **Directivas explícitas para introducir puntos de sincronización**. En OmpSs la directiva de sincronización es *taskwait*.

2.2. Programación dinámica

La programación dinámica es una extensión del esquema divide y vencerás (*divide&conquer*) usado en diversas áreas como, por ejemplo, las matemáticas, la informática o la economía. Se trata de una técnica que permite resolver problemas complejos dividiéndolos en problemas más pequeños y simples, pero evitando resolver el mismo subproblema múltiples veces.

Cuando se puede usar este método, los problemas se resuelven en un tiempo bastante más corto, puesto que se aprovecha de la superposición de los subproblemas. En general, para resolver un problema, es necesario resolver las diferentes partes del problema, para luego combinarlas y alcanzar una solución total. A menudo, los subproblemas se generan y resuelven muchas veces, como en el caso de la sucesión de Fibonnaci, de modo que malgastamos tiempo resolviendo una y otra vez lo mismo. La programación dinámica persigue resolver cada subproblema una

única vez, reduciendo notablemente el número de cálculos. Esto es: una vez se ha computado la solución a un subproblema, se almacena. De esta forma, la próxima vez que se necesite, sólo es necesario leerlo donde está guardado. Cabe destacar que cuanto mayor es el número de repeticiones de los subproblemas, obviamente, más útil es este método. En la actualidad, la programación dinámica es una tarea que el desarrollador debe realizar a mano para cada aplicación. En consecuencia, es una tarea tediosa y costosa en tiempo. A través de este proyecto se pretende conseguir que la memoización, una técnica de programación dinámica, se convierta en un trabajo mucho más ligero y corto, facilitando así la ardua tarea del desarrollador. Esto es debido a que la memoización pasará a hacerse de manera automática mediante unas pocas anotaciones en el código, en las que el programador deberá indicar algunos parámetros, cómo las mostradas en la Figura 1.

2.2.1. Historia

El término *programación dinámica* se introduce en los años 40, de la mano de Richard Bellman. Al nacer, se describe este término como el proceso de resolver un problema tomando la decisión óptima una vez tras otra. En sus inicios, nada tenía que ver con lo que ahora conocemos como programación. El autor le dio ese nombre en el sentido de planificación. Por otro lado, la palabra diñamica fue usada para enfatizar la variabilidad que los problemas podían tener en el tiempo. Pero no sólo eso, Bellman quería una palabra capaz de impresionar a un congresista, algo sobre lo que no fuera posible hacer objeciones [Edd04]. Él mismo explica este razonamiento en este fragmento extraído de su biografía publicada en 1984: *Eye of the Hurricane: An Autobiography* [Dre02].

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."

Su autor siguió trabajando en el concepto y en 1952 publica *The theory of dynamic programming*, donde refina el término. En esta primera publicación cuenta que este método fue concebido para tratar problemas matemáticos derivados del estudio de procesos de decisión multi etapa. Describía estos problemas de la siguiente forma:

Existe un sistema cuyo estado en un tiempo T está determinado por unos parámetros o variables de estado. En ciertos momentos, que se decidirán más adelante o que serán determinados por el propio proceso, es necesario tomar decisiones que afectarán al estado del sistema. Estas decisiones

son transformaciones de las variables de estado. El resultado de las decisiones anteriores se usará para tomar las siguientes, con el objetivo de maximizar alguna función de los parámetros, describiendo el estado final.

Algunos ejemplos de problemas de este tipo son la planificación de líneas de producción o la planificación de visitas de pacientes en una clínica [Bel54].

Hay dos términos que son importantes para discutir este tipo de problemas [Bel54]:

- **Política**: secuencia de decisiones.
- **Política óptima**: política que presenta la situación más ventajosa respecto a un criterio preasignado.

Para afrontar un problema de este tipo usando una aproximación tradicional es necesario tener en cuenta todo el abanico de posibles políticas, calcular los resultados de aquellas que sean factibles y maximizar el resultado de estas últimas. A pesar de ser un proceso sencillo y razonable, no suele ser práctico. Incluso para aquellos problemas que no presentan un número de etapas elevado ni un rango muy grande de decisiones en cada etapa, la maximización resultante acaba siendo impracticablemente grande. Sin embargo, esta excesividad en el tamaño se debe a una sobreinformación. No es necesario tener conocimiento de toda la secuencia de decisiones en todo momento, simplemente necesitamos una receta global que permita determinar la siguiente decisión en términos del estado actual del sistema. Es decir, si sabemos lo que tenemos que hacer en cualquier instante particular, no es necesario conocer las decisiones requeridas posteriormente, sólo la decisión del instante actual. Por tanto, esto reduce notablemente las dimensiones del problema [Bel54].

Este nuevo enfoque suponía un gran avance para procesos estocásticos, puesto que resolver este tipo de problemas usando el enfoque clásico es virtualmente imposible dado su enorme tamaño que requeriría demasiando tiempo de computación.

En la misma publicación, introduce el ***Principio de Optimalidad***. Una política óptima cumple la siguiente propiedad: sea cual sea el estado y las decisiones iniciales, las decisiones restantes constituirán una política óptima con respecto al estado resultante de las primeras decisiones [Bel54].

Ya con las bases puestas, sigue investigando y realiza un informe e incluso publica un libro sobre la teoría de la programación dinámica. A partir de ahí, continúa su trabajo, poniendo más énfasis en aplicar sus descubrimientos teóricos en problemas reales. De hecho, en 1962 publica un libro, junto con Stuart Dreyfus, llamado *Applied Dynamic Programming* en el que discuten como aplicar su método a problemas reales relacionados con satélites y viajes espaciales, la determinación de trayectorias o la planificación de procesos, entre otros.

Por otro lado, ***memoization*** es un terminado acuñado por Donald Michie en 1968 en su publicación *'Memo' functions and machine learning*. Suele confundirse con memorización, pero memoización tiene un significado específico en informática. En esta primera publicación sobre el tema, Donald Michie versa sobre lo bueno que sería que los ordenadores pudieran aprender de la experiencia de modo que su eficiencia aumentara automáticamente tras cada ejecución. Su idea surge del estudio de Arthur Samuel sobre el juego de las damas, en el que diseñó un jugador que aprendía de cada partida que jugaba para jugar cada vez mejor [Sam59].

Michie quiso trasladarlo a funciones matemáticas. Propone cambiar la forma de ver estas funciones. En lugar de verlas sólo como una operación, piensa en representarlas también usando una tabla. Entonces, afirma que

1. una función puede ser evaluada mediante su regla (calcular la operación) o mediante una tabla;

2. esa evaluación procederá mediante regla o tabla o una combinación de ambas según conveniencia en ese instante;
3. la decisión de proceder mediante una u otra forma la realizará la máquina de forma opaca al usuario y
4. que se permitirán varios tipos de interacción entre la regla y la tabla.

Tras cada evaluación usando la regla, se añade una entrada a la tabla [Mic68].

Para justificar esta necesidad de mejorar con la experiencia, utiliza una parábola en la que muestra como el comportamiento humano se corresponde exactamente con este método. Es la siguiente:

Imagine that I am hired by an old-fashioned commercial company which does not possess, or believe in, calculating machines. 'We need someone to sit under the stairs during board meetings', explains the company secretary, 'someone with his wits about him. We sometimes need to know the hcf (Highest Common Factor) of two numbers: when I shout a pair of numbers down the stairs, you will shout the answer up as soon as you've worked it out. Our chairman is an impatien man, and you will be paid a speed bonus.

'Well', I might say, 'I don't know about hcf's, but I need the money and I'll do the best I can'.

So I am issued with Euclid's rule for computing the hcf; I also have the sense to demand a supply of index cards and a pencil. Whenever I apply my rule to a given pair of numbers, I enter this pair together with the result on an index card and add it to a growing pile of tabulated answers. Whenever I am asked for the hcf of a new pair, I first look through this pile, so that I can find it a little faster next time. Through this promotion process, rarely-ocurring problems will gravitate towards the bottom and frequently ocurring problems towards the top. If I do not find the required number-pair in the pile, then and only then do I have recourse to the rule. Having found the answer, in this case by calculation, I enter the result on a new card to be added to the pile. Supposing there is a limit to the size of the pile of cards which I can handle, I discard the bottom card whenever there is need to shorten the pile. [Mic68]

Extrapolando esta parábola a la informática, se trata de tener una estructura de datos en la que guardar los resultados de una función. La primera vez que se calcula, se guarda en la tabla y las siguientes se consulta directamente allí, ahorrando tiempo de cálculo. El autor añade la ordenación de los elementos según el uso, sin embargo, esto no es necesario actualmente, puesto que, usando la estructura correcta, nos cuesta lo mismo acceder a cualquier elemento.

2.2.2. Concepto

La programación dinámica, en la actualidad, es un método de optimización tanto matemático como informático. Se refiere al hecho de

I. dividir un problema grande y complicado en pequeños problemas más simples;
II. resolver los subproblemas, almacenando las soluciones y
III. combinar sus soluciones para conseguir resolver el problema principal.

2.2.3. Programación dinámica en informática

Para poder aplicar este método de optimización a un problema informático, éste debe cumplir dos propiedades [DPV06]:

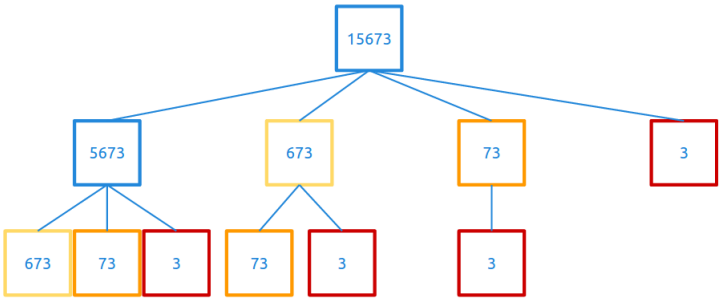


Figura 2: Grafo de tareas de un problema que tiene una subestructura óptima y se descompone en subproblemas superpuestos.

1. **Subestructura óptima:** Un problema tiene una subestructura óptima si se puede conseguir una solución óptima combinando las soluciones óptimas de sus subproblemas.
2. **Subproblemas superpuestos:** Significa que un problema se puede dividir en subproblemas que se repiten varias veces en lugar de generar nuevos subproblemas.

En la Figura 2 se observa un grafo de tareas de un problema que contiene ambas propiedades.

La programación dinámica se puede implementar de dos formas distintas [Pic11]:

- **Top-down (De arriba a abajo):** Con este enfoque, el problema sigue la dirección natural de la recursividad. Cuando un subproblema es calculado por primera vez, se guarda en una tabla. En las sucesivas llamadas a ese subproblema, el resultado se irá a buscar a la tabla en lugar de recalcularlo. A este método se le llama **memoización**.
- **Bottom-up (De abajo a arriba):** Usando este enfoque, lo que se pretende es reformular la recursividad en la dirección contraria. Primero se resuelven los problemas más pequeños y con sus soluciones, que se almacenan en una tabla, se resuelven los problemas superiores. Se suele hacer de forma iterativa con el objetivo de no repetir cálculos. Por ejemplo, en el caso de Fibonacci, si ya conocemos fib(4) y fib(5) podemos calcular fib(6) directamente.

2.3. Implicados

Hay varios agentes implicados en este proyecto que se detallan a continuación.

- **Desarrollador**: el desarrollador del proyecto soy yo. Puesto que es un proyecto complicado, con objetivos ambiciosos y sobre una herramienta ya existente con una complejidad elevada, contaré con el soporte de un Ingeniero de soporte a la investigación.
- **Ingeniero de soporte a la investigación**: será el encargado de ayudar al desarrollador en los puntos críticos. También protagonizará una pequeña formación sobre las herramientas al desarrollador, puesto que al empezar el proyecto no las conoce.
- **Director de proyecto**: su tarea es supervisar el avance del proyecto. Debe comprobar que se cumplen los objetivos marcados en los plazos previstos.
- **Barcelona Supercomputing Center**: dado que el modelo OmpSs ha sido desarrollado por el BSC, este centro se verá beneficiado por este proyecto, pues añadirá una nueva funcionalidad a su modelo, haciéndolo más completo y funcional.
- **Usuarios**: los usuarios verán reducido su esfuerzo para implementar memoización. Además, podrán combinar paralelismo y programación dinámica de una forma sencilla y rápida.

2.4. Trabajos relacionados

Revisando la literatura es posible encontrar diversas aproximaciones a, por un lado, memoización automática y, por otro lado, la combinación de programación dinámica y paralelismo.

2.4.1. Memoización automática

En 1991, Peter Norvig discute sobre la memoización automática en el lenguaje *Common Lisp* [Nor91]. Detalla posibles implementaciones en este lenguaje, pero también da unas pinceladas sobre como podría hacerse en otros lenguajes usando macros. Comenta ciertos puntos importantes, en común con este proyecto. Los detalles de implementación de la memoización deben esconderse al usuario e incluso la propia existencia de la tabla no es conocida por el usuario. Por tanto, el usuario puede olvidarse de declarar, alocatar e inicializar la tabla, e incluso de interactuar con ella para obtener/guardar los resultados y, sin embargo, seguir obteniendo los beneficios de este método. Sin embargo, como sugirió Michie, proponen descartar las entradas de la tabla más viejas, quedando sólo las más nuevas; contrastando con el enfoque de este proyecto que prevee no descartar ninguna de las entradas.

Más adelante, en 1995, se discute el uso de una herramienta de memoización automática en sistemas de Inteligencia Artificial (IA) [MFH95]. Los autores de la publicación en cuestión describen como hacer viable este tipo de herramientas para problemas de gran escala. Justifican su uso enumerando las ventajas que puede ofrecer y presentan un paquete público con este método para el lenguaje *Common Lisp*. Por otro lado, también identifican los potenciales problemas que pueden surgir en su herramienta. Al igual que en este proyecto, la responsabilidad de decidir qué funciones deben ser memoizadas recae sobre el usuario. A diferencia de este proyecto, la tabla de memoización permanece viva hasta el final del programa mientras que nuestra aproximación es destruir la tabla cuando la función que inició la memoización termina.

Los autores dividen las ventajas de la memoización automática en tres grupos, de los cuales dos son extensibles a este proyecto: calidad de la solución y facilidad de uso.

- **Calidad de la solución**: Por lo que respecta a calidad, la memoización automática consigue soluciones más cortas y limpias. En otras palabras, conseguimos un código más legible y mantenible que si usáramos una memoización artesanal. Otro punto clave es evitar introducir errores en códigos ya probados. Si se usara una memoización manual, al tener que escribir código nuevo, la posibilidad de introducir nuevos errores es considerable. Este aspecto es especialmente importante en códigos grandes y complejos, puesto que suele haber cierta reticencia a reescribir este tipo de rutinas una vez han sido probadas y verificadas. Por último y no menos importante, usando la memoización automática es sencillo y rápido cambiar de la versión optimizada a la versión normal y, en consecuencia, comparar sus rendimientos.
- **Facilidad de uso**: Usando una herramienta de memoización automática es muy sencillo y rápido mejorar la eficiencia de una aplicación. Ninguna de las alternativas para mejorar la eficiencia, como podrían ser realizar un nuevo algoritmo o usar un enfoque de programación dinámica *bottom-up*, requieren tan poco esfuerzo como esta. Además, vuelven a aparecer ciertos aspectos descritos anteriormente, el usuario no debe preocuparse de mantener nuevo código. Es especialmente importante este aspecto, la facilidad de uso, en aquellas aplicaciones que tienden a cambiar frecuentemente.

Por otro lado, de los problemas que describe, hay uno extrapolable a este trabajo. La memoización se realiza mediante una búsqueda exacta de los parámetros de entrada. Por ejemplo, si usamos números decimales, 2 y 2.0, no son lo mismo, pero el resultado de la función sí sería

igual. Otro caso parecido sería el Máximo Común Divisor de dos números. Si tenemos guardado el MCD de 2 y 4 y nos consultan el de 4 y 2, no coincide y se recalculará. Esto es solucionable mediante un *wrapper* que sea capaz de identificar estos casos.

2.4.2. Programación dinámica y paralelismo

Existen varios trabajos en los que se combina programación dinámica y paralelismo. Sin embargo, todos ellos usan el método *Bottom-up*, a diferencia de la herramienta desarrollada en este proyecto, que usa el método *Top-down* también conocido como memoización. Además, todos ellos investigan un problema o un tipo de problema específico. Por ejemplo, el más repetido es el problema de la mochila (en inglés, Knapsack) [BE05][ARQ93], aunque también se tratan comparaciones de secuencias [MCU+01]. Cabe destacar, también, la discusión sobre *nonserial polyadic dynamic programming*. Se trata de una familia de algoritmos de programación dinámica donde las dependencias de datos no son uniformes [GNS07]. En contraste, en este trabajo se intenta ofrecer una solución genérica que pueda resolver cualquier problema.

Además, la herramienta desarrollada en este trabajo funciona en arquitecturas de memoria compartida, mientras que algunos de los experimentos mencionados antes se realizan sobre arquitecturas de memoria distribuida [MCU+01].

3. Alcance y objetivos del proyecto

En esta sección de definen los objetivos y el alcance del proyecto. También se enumeran los posibles obstáculos que se prevee que puedan ocurrir, presentando, a su vez, las soluciones para solventarlos.

3.1. Formulación del problema

En la actualidad, a pesar de la gran potencia de cálculo de la que se dispone, sigue habiendo muchos problemas que requieren técnicas de optimización para poder ejecutarlos en tiempos asumibles. La **memoización** es una de las técnicas de optimización que nos permite reducir los tiempos de ejecución de ciertos tipos de problemas. Estos problemas son aquellos que tienen una subestructura óptima y que se dividen en subproblemas superpuestos (ver 2.2.3).

Sin embargo, la memoización es una técnica difícil de implementar, con la que es fácil introducir errores en el código y, además, reduce la mantenibilidad y legibilidad del código. En la Figura 3 se observa un código. En la Figura 4 se observa un código que realiza la misma función pero memoizado. El tamaño de la función ha crecido considerablemente, a pesar de ser muy sencilla. Todo esto produce cierta reticencia a implementar memoización en algunos desarrolladores.

Listing 2: Código de ejemplo para calcular la secuencia de Fibonacci

```
// Codigo de ejemplo para calcular la secuencia de Fibonacci
int fib( int n ) {
  if( n < 2 ) return n;
  else return fib( n - 1 ) + fib( n - 2 );
}
```

Figura 3: Código de ejemplo para calcular la secuencia de Fibonacci.

Listing 3: Código de ejemplo para calcular la secuencia de Fibonacci con memoización

```
// Codigo de ejemplo para calcular la secuencia de Fibonacci con memoizacion
int memo_table[n]; //all values initialized to -1

int fib( int n ) {
  if( n < 2 ) return n;
  else {
    if( memo_table[n] != -1 ) return memo_table[n];
    else {
      memo_table[n] = fib( n - 1 ) + fib( n - 2 );
      return memo_table[n];
    }
  }
}
```

Figura 4: Código de ejemplo para calcular la secuencia de Fibonacci con memoización.

Por otra parte, la combinación de paralelismo y memoización es un tema sin tratar en la literatura. Por un lado, las máquinas de hoy en día disponen, en la mayoría de ocasiones, de procesadores

multihilo. Por otro lado, aún utilizando técnicas de optimización como la memoización, algunos problemas siguen siendo muy costosos en tiempo. En consecuencia, podríamos combinar técnicas de paralelismo con la memoización para intentar reducir todavía más los tiempos de nuestras aplicaciones.

En resumen, a pesar de ser una gran técnica de optimización, la memoización presenta ciertas desventajas que hay que resolver:

- Peor mantenibilidad y legibilidad del código.
- Propenso a introducir errores en códigos ya verificados y probados.
- Complejidad.

Además, con el objetivo de optimizar todavía más nuestras aplicaciones hay otro problema que resolver:

- Combinación de paralelismo y memoización.

3.2. Alcance

Para estudiar el impacto de la memoización sobre aquellos problemas que usen algoritmos basados en recursividad, se implementará un amplio conjunto de éstos, que servirán como códigos de evaluación. Es necesario, pues, analizar qué tipos de problema se adecúan a las necesidades del proyecto, para más tarde poder buscar problemas que cumplan con las propiedades requeridas. Por último, se deben implementar de distintas formas para comprobar las diferencias en el rendimiento.

Además, se harán las respectivas modificaciones sobre el compilador Mercurium y el *runtime* Nanos++, para proveer de soporte para programación dinámica al modelo de programación OmpSs, puesto que son estas las herramientas que usa para compilar y ejecutarse. Esto requiere:

a. Añadir nueva sintaxis al modelo, para soportar la nueva funcionalidad. Mediante ésta, el usuario demandará la memoización. Será necesario, también, que le indique algunos parámetros para el correcto funcionamiento de la herramienta.

b. Extender el *runtime* Nanos++ con el fin de guardar los resultados la primera vez que se calculan, para, más tarde, poder consultarlos sin necesidad de computar de nuevo.

c. Extender el compilador Mercurium para que sea capaz de reconocer la nueva cláusula que se añada al modelo. Además, deberá generar cierta información y algunas llamadas a Nanos++ para realizar la memoización correctamente.

Una vez implementado el soporte para memoización de OmpSs, se procederá a ejecutar los códigos de evaluación. Se usarán cuatro versiones:

- Versión serie.
- Versión usando OmpSs.
- Versión con memoización manual.
- Versión utilizando el soporte para memoización de OmpSs.

Más tarde, se efectuará una comparación de rendimiento entre las distintas versiones. Esto es comparar los tiempos de ejecución de las versiones.

De la comparación se concluirá, por una parte, si la mejora de rendimiento es la esperada y, por otra parte, para qué tipo de problemas funciona mejor. Se tendrá en cuenta la ganancia de rendimiento sobre los problemas probados.

Además, se instrumentará el código para generar trazas, mediante la librería Extrae. Usando Paraver, podremos visualizar las trazas, que nos permitirán extraer un conocimiento mayor de las ejecuciones.

Para finalizar, se aprovechará el conocimiento extraído del análisis anterior para intentar mejorar el soporte para memoización. Se repetirá el proceso varias veces para afinar todo lo posible.

3.3. Objetivos

Los objetivos que persigue este proyecto son:

1. Añadir soporte para programación dinámica al modelo de programación paralela OmpSs. Para ello es necesario:
    i. Extender la sintaxis del modelo.
    ii. Extender el compilador Mercurium.
    iii. Extender el *runtime* Nanos++.
2. Crear un marco de evaluación de rendimiento. Esto consiste en:
    i. Analizar las propiedades que deben tener los problemas que formen parte de este marco.
    ii. Buscar problemas que cumplan con las propiedades necesarias e implementarlos.
    iii. Ejecutar las distintas versiones de los problemas para comparar los resultados.

3.4. Posibles obstáculos y soluciones

Antes de nada, es importante entender que un diseño erróneo del proyecto podría derivar en resultados erróneos o defectuosos. Por ejemplo, ejecutando problemas que no cumplan las propiedades necesarias o utilizando técnicas de programación dinámica anticuadas o desacertadas. Para evitar esto, el tutor del proyecto supervisará semanalmente, en una reunión, el trabajo realizado. Además, puesto que se desarrollará en un grupo de investigación especialista en este ámbito, también contará con la supervisión y apoyo del resto de miembros del grupo.

Cabe añadir, también, que el tiempo juega en contra: el proyecto debe terminarse en poco más de tres meses. En consecuencia, y como se comentó en el párrafo anterior, se seguirá un exhaustivo plan de supervisión, con reuniones semanales, para asegurar que se cumplen los plazos establecidos. Asimismo, es necesario apuntar que las modificaciones necesarias deben hacerse sobre el compilador y el *runtime*. Son partes críticas y complicadas, que además son difíciles de depurar en caso de introducir errores en el código. Por lo tanto, con el fin de asegurar que todo funciona de forma correcta, se ejecutará un robusto conjunto de juegos de pruebas.

Respecto a los resultados, pueden no ser de trivial interpretación. En este proyecto, el objetivo esencial es mejorar el rendimiento de las aplicaciones ejecutadas. El rendimiento puede verse afectado negativamente, no sólo porque la herramienta desarrollada funcione de forma incorrecta, sino por una mala implementación del algoritmo o por una mala gestión de los recursos ofrecidos por el modelo de programación. Por ello, es importante usar las herramientas adecuadas de modo

que se tenga claro en todo momento dónde estamos consumiendo el tiempo, y, en consecuencia, perdiendo rendimiento para saber por dónde atacar el problema.

Por último, y no menos importante, hay que hacer hincapié en que OmpSs es una plataforma de programación paralela. Debido a esto, es importante disponer de máquinas potentes para maximizar el rendimiento.

4. Metodología y rigor

En esta sección se describen los métodos de trabajo, así como las herramientas de seguimiento y los métodos de validación usados en el desarrollo del proyecto.

4.1. Métodos de trabajo

Teniendo en cuenta el corto periodo de tiempo del que se dispone para realizar el proyecto, es lógico pensar que una metodología ágil es lo óptimo para cumplir los plazos. Sin embargo, las metodologías ágiles más conocidas, como por ejemplo SCRUM o XP (*eXtreme Programming*) están orientadas a trabajos en grupo, de modo que seguirlas a rajatabla no sería una buena opción. No obstante, ciertos conceptos de estas metodologías sí son útiles para un proyecto individual.

4.1.1. Desarrollo de ciclos cortos

Una metodología de desarrollo de ciclos cortos se ajusta bastante bien a las necesidades de este trabajo. Se trata de proponer objetivos en ciclos cortos (en este caso semanales). De esta forma, el seguimiento es más preciso, asegurando así que se cumple la planificación y se tiene consciencia real del estado del proyecto. Además, si surgen errores, se atajan rápido, minimizando los riesgos.

4.1.2. *Feedback* del cliente

Por otra parte, es importante tener *feedback* del cliente. Sin embargo, en este proyecto no hay un cliente real, aunque sí un director de proyecto, que actuará de cliente, de modo que deberá comprobar el avance del proyecto con la máxima frecuencia posible. Así se evitarán malentendidos o se corregirán lo antes posible.

4.1.3. Desarrollo guiado por pruebas

Se seguirá, también, el desarrollo guiado por pruebas (*Test Driven Development* en inglés), ya que para cada modificación que se haga sobre el compilador o *runtime* debemos asegurar que un código correcto pueda compilar y ejecutarse adecuadamente.

4.1.4. Prototipado

Este método permite desarrollar de una forma incremental. Los desarrolladores van añadiendo funcionalidades, o mejorando las ya existentes, a la vez que el cliente (director en este caso) evalúa el producto para determinar si cumple con las expectativas, como se muestra en la Figura 5. Si hay algún punto que no satisface lo acordado, o que puede mejorar, se itera sobre el estado actual del proyecto y se vuelve a evaluar. Se repite el proceso hasta conseguir el producto esperado.

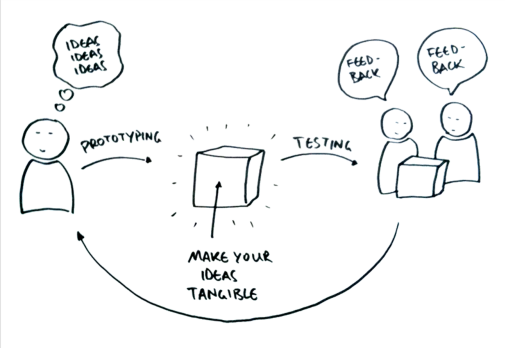


Figura 5: Dibujo en el que se muestra la metodología de prototipado.

4.2. Herramientas de seguimiento

Para este proyecto se usará el sistema de control de versiones GIT. Esta herramienta es ideal para forzar ciclos cortos de desarrollo. Además, también obliga a documentar cada cambio que se pretenda realizar sobre el servidor.

También se añade una herramienta de *ticketing* en la que se podrán reportar todos los errores que se vayan encontrando. Además, esta herramienta permite un seguimiento del error, pasando por diferentes estados, desde detectado hasta solucionado, pasando por "bajo análisis" o "en progreso de reparación".

Por otra parte, el BSC ofrece herramientas para medir el rendimiento de las aplicaciones y conocer con precisión en qué se está gastando el tiempo. En este proyecto se usarán Extrae y Paraver. Gracias a ellas se podrá saber si las modificaciones que se van realizando sobre el modelo son eficientes y, por tanto, mejoran el rendimiento.

4.3. Métodos de validación

Un grupo de investigación especialista en el ámbito ofrecerá supervisión y soporte. Esto, combinado con reuniones semanales con el director del proyecto en el que se revisa el estado, a mi parecer, son suficiente para dar validez a los avances que se vayan realizando y comprobar que se van asumiendo los objetivos establecidos.

Sin embargo, se cuenta también con la herramienta de *ticketing*, en la que los usuarios podrán reportar todos los fallos que encuentren y el desarrollador podrá comprobar el estado de todos. Una vez el estado de cada uno de ellos sea "corregido" o "error aceptado", el trabajo será válido.

Asimismo, puesto que las modificaciones se hacen sobre compilador y *runtime*, un error podría afectar a las ejecuciones de las aplicaciones. Para comprobar la corrección de estas, también han sido implementadas y compiladas con un compilador estándar, de manera que los resultados de las ejecuciones deberán coincidir.

5. Planificación del proyecto

Esta sección está dedicada a definir y describir las tareas en las que se dividirá el proyecto. A su vez, se describirán los recursos usados para realizarlas. Se añaden los diagramas de Gantt y Pert correspondientes. Para finalizar, se realiza un pequeño estudio de las desviaciones que puede sufrir el proyecto y cómo solventarlas.

5.1. Descripción de las tareas

Este proyecto se divide en cuatro grandes tareas:

I. Planificación del proyecto y viabilidad

II. Análisis y diseño del proyecto

III. Desarrollo

IV. Etapa final

A continuación, se describe cada una de ellas detalladamente.

5.1.1. Planificación del proyecto y viabilidad

Esta parte corresponde directamente a la asignatura GEP (Gestión de Proyectos). Consta de seis etapas:

I. Definición del alcance del proyecto

II. Planificación del proyecto

III. Estimación del presupuesto del proyecto

IV. Estado del arte y contextualización

V. Pliego de condiciones

VI. Documento final

Además, entre la etapa 3 y 4 es necesario realizar una presentación para la asignatura de GEP.

5.1.2. Análisis y diseño del proyecto

En esta etapa del proyecto lo más importante es realizar un análisis exhaustivo del proyecto para desarrollar un diseño apropiado.

En primer lugar, para el análisis, es esencial definir los objetivos y los requisitos del proyecto. Por otro lado, el diseño del proyecto consiste en:

a. Crear un conjunto de problemas con los requisitos que se obtienen del análisis para comprobar, una vez se haya terminado el proyecto, que los objetivos se cumplen.

b. Diseñar el soporte para programación dinámica en el modelo de programación OmpSs de manera que se cumplan los objetivos determinados en el análisis.

5.1.3. Desarrollo

Esta fase se divide en tres subetapas necesarias para realizar el desarrollo de una forma apropiada. Son las siguientes:

1. **Configuración del entorno**. Como indica el nombre, esta etapa está destinada a instalar todas las aplicaciones y herramientas necesarias para el correcto desarrollo del proyecto. Además, una vez instaladas, será necesario configurarlas de la forma que más convenga.
2. **Adquisición de conocimiento**. Este proyecto pretende añadir una funcionalidad al modelo OmpSs. Como se mencionaba en la sección 2.1, el entorno de OmpSs está formado por el compilador Mercurium y el sistema de *runtime* Nanos++. Estas dos herramientas son códigos ya existentes sobre los que se pretende hacer una extensión para soportar memoización en el modelo. Dada la alta complejidad del código de estas herramientas, combinada con el desconocimiento sobre ellas, exige un tiempo de familiarización y adaptación para poder trabajar en la extensión.
3. **Desarrollo**. En esta tercera subetapa, se ejecutará el desarrollo de lo referido en la sección 3.2. Puesto que, como se detallaba en la sección 4, se utilizará, entre otros métodos, prototipado, esta subetapa se dividirá en iteraciones.

   En cada una de las iteraciones se añadirá una funcionalidad, o se mejorará alguna ya existente. Una vez hecho esto, se evaluará y se identificarán ciertos puntos de mejora. Estos puntos de mejora se intentarán implementar en la siguiente iteración. Cuando esté completado, se volverá a evaluar. Este proceso se repetirá tantas veces como sea necesario hasta alcanzar los objetivos.

5.1.4. Etapa final

Por último, en la etapa final, se procederá a escribir la memoria del proyecto y preparar la presentación para la defensa ante el tribunal.

5.1.5. Dedicación

En el Cuadro 1 se detallan las horas que se han dedicado a cada tarea.

Etapa	**Dedicación (horas)**
Planificación del proyecto y viabilidad	80
Análisis y diseño del proyecto	100
Desarrollo: Configuración del entorno	50
Desarrollo: Adquisición de conocimiento	80
Desarrollo: Desarrollo	200
Etapa final	50
Total	**560**

Cuadro 1: Previsión de horas dedicadas a cada tarea del proyecto.

5.2. Recursos

En esta sección se detallan los recursos usados para llevar a cabo el proyecto. Se dividen en:

Figura 6: Dell Latitude E7440

Figura 7: Asus Zenbook UX32A

Figura 8: Supercomputador Marenostrum

- Recursos humanos.
- Recursos hardware.
- Recursos software.

5.2.1. Recursos humanos

Los recursos humanos necesarios para el desarrollo de este proyecto han sido:

- Becario de investigación. El desarrollador del proyecto.
- Ingeniero de soporte a la investigación. Orientará al desarrollador cuando sea necesario.
- Director de proyecto. Supervisará el avance del proyecto, así como el cumplimiento del calendario y los objetivos establecidos.

5.2.2. Recursos hardware

- Dell Latitude E7440 (ver Figura 6). Es el equipo que me proporciona el BSC.
- Asus Zenbook UX32A (ver Figura 7). Mi equipo personal.
- Marenostrum (ver Figura 8)

5.2.3. Recursos software

- Microsoft Windows 7 Professional

- Ubuntu 14.04 LTS
- Microsoft Office 2010/2013
- Gantter
- Dropbox
- Google Drive
- Github
- Editor de texto (Geany/GVIM/Gedit)
- TeX Live
- Kile
- Compilador Mercurium
- *Runtime* Nanos++
- Extrae
- Paraver

Algunos de los recursos software mencionados son muy conocidos, concretamente, los ocho primeros. Los otros se detallan a continuación:

- TeX Live: Es la distribución de LaTeX por defecto para Ubuntu.
- Kile: El entorno de desarrollo de documentos LaTeX usado para realizar la memoria.
- Compilador Mercurium: Se trata del compilador que usa el modelo de programación paralela OmpSs.
- *Runtime* Nanos++: Es el entorno de *runtime* sobre el que se ejecutan las aplicaciones del modelo de programación paralela OmpSs.
- Extrae: Se trata de una librería que, mediante la instrumentación del código, genera unas trazas con información y estadísticas sobre las ejecuciones de nuestras aplicaciones.
- Paraver: Es una herramienta con la que se pueden visualizar las trazas creadas con Extrae.

5.3. Diagramas

A continuación se presentan los dos diagramas utilizados para la planificación del proyecto.

5.3.1. Diagrama de Gantt

Con el objetivo de exponer el tiempo de dedicación previsto para cada tarea, se incluye el diagrama de Gantt presentado en la Figura 9.

5.3.2. Diagrama de Pert

Con el objetivo de exponer la relación entre tareas, se incluye el diagrama de Pert presentado en la Figura 10.

5.4. Valoración de alternativas y plan de acción

En la sección Metodología y rigor se estipulaba que se pretendía usar ciertos conceptos de metodologías ágiles para desarrollar este proyecto. Entre ellos se mencionaba el desarrollo de ciclos cortos, con un seguimiento exhaustivo. Una de las ventajas que se observaba era que de esta forma se tiene consciencia del proyecto casi a tiempo real, de modo que si surgen errores o malentendidos se pueden corregir o adaptar antes de que sea demasiado tarde e implique un esfuerzo inasumible. Por ello, la planificación inicial puede ir variando, tanto para bien: una tarea se acaba antes de lo esperado y se empieza la siguiente; como para mal: una tarea se demora más de lo que debería y retrasa las próximas.

Las posibles desviaciones o errores pueden aparecer, principalmente, en la etapa de desarrollo, puesto que la adquisición de conocimiento y la configuración del entorno no parecen tareas complicadas que puedan llevar a retrasos, al igual que el análisis y el diseño. Asimismo, la parte de planificación del proyecto y viabilidad tampoco es susceptible de retrasarse ya que las fechas están estipuladas por la asignatura GEP, y no cumplir los plazos significaría fracasar en ella. En la etapa de desarrollo se añadirá la funcionalidad para memoización en el modelo OmpSs. Para ello habrá que extender:

- El compilador, que se encarga de generar el ejecutable a partir del código fuente.
- El sistema de *runtime*, una serie de recursos (tanto hardware como software) que permiten ejecutar una aplicación. Se trata de un mecanismo diseñado para proveer servicios sin importar el lenguaje de programación.

Los errores que pueden surgir al modificar estas partes pueden ser, por ejemplo:

- Introducir errores en el compilador, de forma que un código fuente correcto no genere un archivo ejecutable.
- Introducir errores en el compilador que desemboquen en un archivo ejecutable que no realice el trabajo deseado.
- Introducir errores en el sistema de *runtime* cuyas consecuencias eviten que el programa pueda ser ejecutado.
- Introducir errores en el sistema de *runtime* que resulten en ejecuciones incorrectas del programa. Es decir, la ejecución es posible pero el resultado es incorrecto.
- Problemas en el sistema de *runtime* que ralenticen la ejecución del programa, de modo que el rendimiento empeore.

A pesar de tener que lidiar con todos estos posibles errores, no debería afectar negativamente a la planificación debido a que se destina un elevado número de horas a esta fase. Sin embargo, en caso de que no fuera suficiente, sería posible reclamar la ayuda de los compañeros de grupo, o incluso del director del proyecto, para solucionar los errores lo antes posible, evitando grandes retrasos. En el peor de los casos, cualquiera de estas posibles situaciones, o una combinación de ellas, no debería demorarse más de una semana.

Por otra parte, semanalmente, en una reunión con el director del proyecto, se comprobará que se esté siguiendo el camino adecuado. Puesto que el proyecto tendrá trece semanas de vida, habrá trece reuniones. La dedicación semanal estimada es 45 horas, en consecuencia, el proyecto es viable. Por último, mencionar que las posibles desviaciones no tendrían ningún efecto sobre los recursos, más allá de unas horas extra de uso.

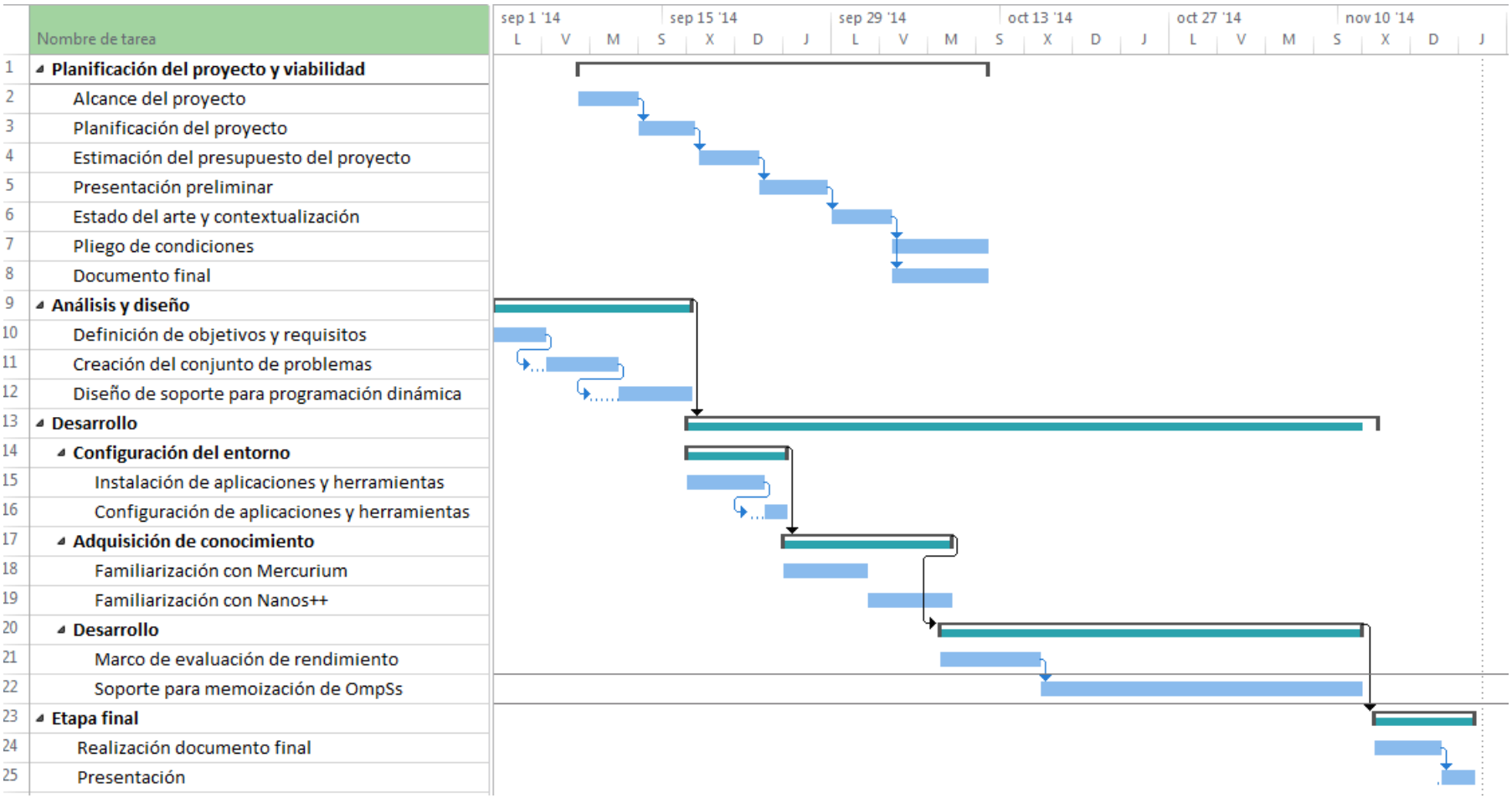


Figura 9: Diagrama de Gantt

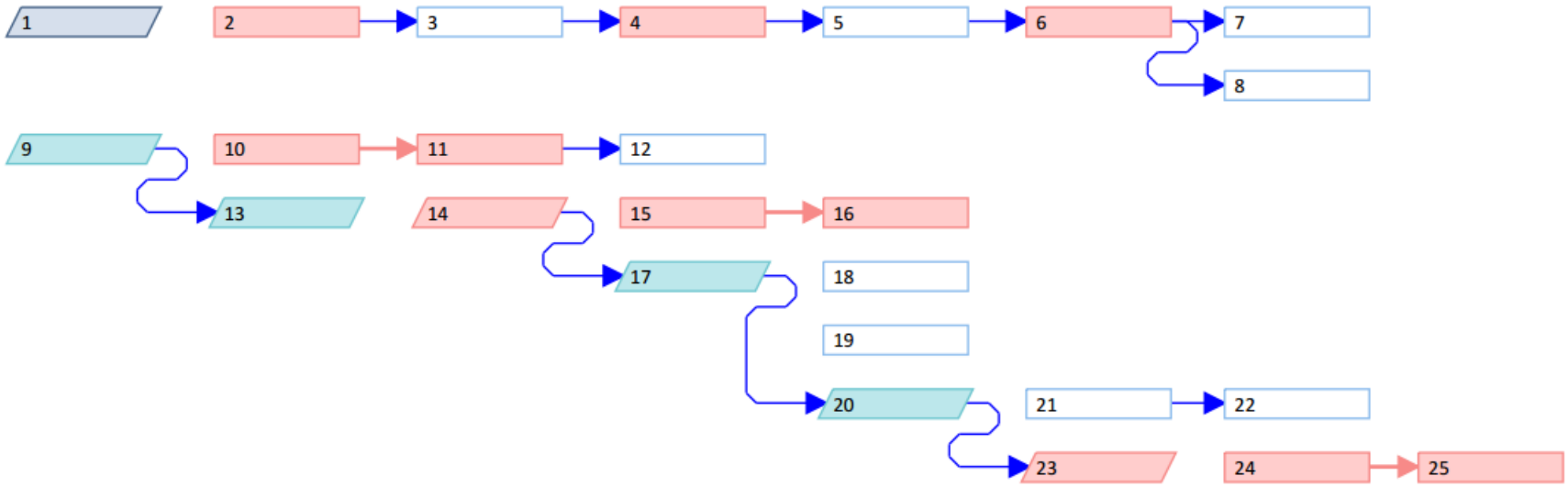


Figura 10: Diagrama de Pert

6. Costes del proyecto

En esta sección se detallan los costes del proyecto. Estos costes están desglosados en:

- Recursos humanos
- Recursos hardware
- Recursos software
- Contingencias
- Imprevistos

6.1. Costes de recursos humanos

En el Cuadro 2 se detalla la dedicación de cada uno de los roles que han participado en el proyecto así como el coste que supone.

Rol	**Dedicación (horas)**	**Precio por hora (€/h)**	**Coste total (€)**
Director de proyecto	30	50	1500
Ingeniero de soporte a la investigación	70	30	2100
Becario de investigación	560	10	5600
Total	**660**		**9200**

Cuadro 2: Previsión de costes de los recursos humanos.

6.2. Costes de recursos hardware

En el Cuadro 3 se explican los costes de los recursos hardware y su amortización. Cabe destacar que la hora de computación en Marenostrum no es gratuita, sin embargo, no se ha podido averiguar el coste.

Recurso	**Precio (€)**	**Unidades**	**Tiempo de vida**	**Amortización (€/h)**
Dell Latidude E7440	1600	1	4 años	0.21
Asus Zenbook UX32A	910	1	4 años	0.12
Marenostrum	-	1	-	-
Total	**2510**			**0.33**

Cuadro 3: Previsión de costes de los recursos hardware.

6.3. Costes de recursos software

En el Cuadro 4 se desglosan los costes de los recursos software y su amortización.

Recurso	Precio (€)	Unidades	Tiempo de vida	Amortización (€/h)
Microsoft Windows 7 Professional	89.99	2	3 años	(Incluido en los PC)
Ubuntu 14.04 LTS	0	2	-	-
Microsoft Office 2010/2013	119	2	3 años	0.06
Gantter	0	1	-	-
Dropbox	0	1	-	-
Google Drive	0	1	-	-
Github	0	1	-	-
Editor de texto (Geany/Vim/Gedit)	0	2	-	-
Compilador Mercurium	0	2	-	-
Runtime Nanos++	0	2	-	-
Librería de instrumentación Extra	0	2	-	-
Paraver	0	2	-	-
Total	**417.98**			**0.06**

Cuadro 4: Previsión de costes de los recursos software.

6.4. Costes directos por actividad

En el Cuadro 5 se observa qué recursos materiales se prevee que se usen en cada actividad y la duración de cada actividad, de modo que se puedan extraer el número de horas que se ha usado cada recurso con el fin de calcular el coste total. Los recursos estarán identificados por un número tal como se muestra a continuación:

1. Dell Latitude E7440
2. Asus Zenbook UX32A
3. Marenostrum
4. Microsoft Windows 7 Professional
5. Ubuntu 14.04 LTS
6. Microsoft Office 2010/2013
7. Gantter
8. Dropbox
9. Google Drive
10. Github
11. Editor de texto (Geany/GVIM/Gedit)
12. TeX Live
13. Kile
14. Compilador Mercurium
15. *Runtime* Nanos++
16. Librería de instrumentación Extrae
17. Paraver

6.5. Costes indirectos

Al realizar el proyecto en el BSC, no puedo acceder a los costes indirectos como podrían ser consumo eléctrico, coste del edificio o impuestos, ya que es información privada.

Actividad	Recursos	Duración (horas)
Planificación del proyecto y viabilidad	1, 2, 4, 5, 6, 7, 8, 9, 11	80
Análisis y diseño	1, 2, 4, 5, 6, 8, 11	100
Desarrollo: Configuración del entorno	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17	50
Desarrollo: Adquisición de conocimiento	1, 5, 11, 14, 15	80
Desarrollo: Desarrollo	1, 2, 3, 5, 10, 11, 14, 15, 16, 17	200
Etapa final	1, 2, 4, 5, 6, 7, 8, 9, 10, 12, 13	50

Cuadro 5: Utilización de los recursos por actividades.

6.6. Contingencias

Los posibles riesgos de este proyecto son, como se relataba en la sección 5.4:

- Introducir errores en el compilador, de forma que un código fuente correcto no genere un archivo ejecutable.
- Introducir errores en el compilador que desemboquen en un archivo ejecutable que no realice el trabajo deseado.
- Introducir errores en el sistema de *runtime* cuyas consecuencias eviten que el programa pueda ser ejecutado.
- Introducir errores en el sistema de *runtime* que resulten en ejecuciones incorrectas del programa. Es decir, la ejecución es posible pero el resultado es incorrecto.
- Problemas en el sistema de *runtime* que ralenticen la ejecución del programa, de modo que el rendimiento empeore.

Cualquiera de ellos sólo afectaría en un posible aumento de las horas dedicadas. En el peor de los casos, se estima que se puedan necesitar unas 5 horas extra por parte del jefe de proyecto y unas 10 horas extra por parte del Ingeniero de soporte a la investigación. Por parte del becario, no habrá añadido, puesto que si no surgen errores, se dedicará a eventuales mejoras sobre la herramienta para optimizar el rendimiento, de modo que las horas se invertirán igualmente en el proyecto. En conclusión, el coste de contingencias es (5*50)+(10*30)=550€.

6.7. Imprevistos

Los imprevistos que podrían surgir son:

- Desconfiguración del entorno: Reconfigurar el entorno, desde cero, no supondría más de 8-10 horas.
- Avería o pérdida del equipo: Se dispone de dos equipos, uno personal y otro proporcionado por el BSC, de modo que esto no afectaría más de 2-4 horas a la planificación.

- Pérdida de datos o de avances: aunque es muy difícil que suceda, ya que todos los datos y avances deben ir subiéndose a la nube o a GIT, siempre existe la posibilidad. En cualquier caso, como usamos un desarrollo de ciclos cortos y lo recomendado es subir los datos al servidor al menos una vez al día, como máximo se perdería un día de trabajo, es decir, unas 8-10 horas.

En cualquier caso, como se menciona en el apartado de contingencias, este tiempo se invertiría en el proyecto de todas formas, ya sea por imprevistos, o por eventuales mejoras de la herramienta una vez conseguidos los objetivos básicos, así que no aumentaría los costes.

6.8. Estimación total

El coste total previsto del proyecto es **9942€** y se puede observar, debidamente desglosado, en el Cuadro 6. Para calcular el coste de los recursos materiales se ha multiplicado el coste por hora de cada recurso, por el número de horas que se usa.

Recursos	**Dedicación (horas)**	**Coste por hora (€/h)**	**Coste (€)**
Dell Latitude E7440	560	0.21	117.6
Asus Zenbook UX32A	480	0.12	57.6
Microsoft Office 2010/2013	280	0.06	16.8
Becario de investigación	560	10	5600
Ingeniero de soporte a la investigación	70	30	2100
Director de proyecto	30	50	1500
Contingencias	-	-	550
Total	-	-	**9942**

Cuadro 6: Estimación del coste total del proyecto (sólo se muestran los recursos que suponen un coste).

6.9. Control de gestión

Para monitorizar adecuadamente los recursos que se destinan a cada parte del proyecto, antes de cada reunión semanal se hace un pequeño reporte para situar el avance del proyecto. Además, en este documento se contabilizan las horas dedicadas a cada tarea, junto con los recursos hardware y software que han sido necesarios.

Asimismo, al finalizar cada tarea o subtarea se modifica el documento de planificación, sustituyendo los datos esperados por los datos reales. De esta manera, todos los recursos, tanto humanos como materiales, se tienen controlados. En consecuencia, se pueden detectar desviaciones respecto a la planificación y se puede ajustar el presupuesto conforme se vaya necesitando.

7. Sostenibilidad y compromiso social

En esta sección se evalúa la sostenibilidad y el compromiso social del proyecto. La evaluación se hará considerando por separado la dimensión económica, social y ambiental.

7.1. Dimensión económica

En el documento actual existe una evaluación de costes detallada, en el que se distingue, además, entre recursos materiales y recursos humanos. Asimismo, se tienen en cuenta las contingencias del proyecto y los posibles imprevistos que puedan surgir para realizar un presupuesto final. Dado que es un proyecto de investigación, orientado a la comunidad científica, no debe ser competitivo, de modo que no aplica valorar su viabilidad económica en términos de competitividad. Cabe destacar que no se ha mantenido el presupuesto previsto, sin embargo, esto es debido a una decisión del desarrollador y el director de alargar el proyecto un mes y medio para enriquecerlo.

Por otra parte, el proyecto podría ser desarrollado en menos tiempo, pero no con menos recursos. Es decir, quizá alguien con más experiencia y que cuente con conocimiento sobre el modelo podría llevar a cabo el trabajo más rápido. Sin embargo, ello implicaría un mayor coste en recursos humanos, ya que su posición va ligada a una remuneración mayor. En consecuencia, el coste, más o menos, acabaría siendo el mismo. Cabe añadir, además, que el tiempo está ajustado al máximo y que se ha intentado hacer un reparto de tiempo consecuente con la importancia de cada tarea.

Por supuesto, al añadir una funcionalidad al modelo de programación OmpSs desarrollado por el BSC, este proyecto es una colaboración al modelo.

7.2. Dimensión social

En la actualidad, España inicia un crecimiento económico tras varios años en una crisis profunda. Esta crisis, sumada a la división en Cataluña entre independentismo y unionismo, se traduce en una crispación social y política en todo el territorio catalán. No obstante, esto parece no haber afectado al sector informático toda vez que es uno de los pocos que ha seguido creciendo y ofreciendo empleo. Probablemente, este proyecto no tenga ninguna repercusión en la situación actual.

Actualmente, la memoización, es una tarea tediosa que debe ser realizada manualmente y a medida para cada aplicación. En consecuencia, surge la necesidad de facilitarla. Este proyecto pretende acabar con esta situación, convirtiendo el engorroso proceso actual en unas pocas anotaciones, que, además, no alterarán el código original, puesto que sólo se interpretarán si se decide usar el modelo de programación OmpSs. Esto mejora notablemente la calidad de vida del usuario, que verá su trabajo reducido considerablemente. Además, nadie se ve perjudicado por esto.

7.3. Dimensión ambiental

En el Cuadro 5 se detallan los recursos utilizados para cada fase del proyecto. Principalmente, consumen electricidad. Además, el consumo no será elevado toda vez que los dos equipos con los que se realiza el proyecto son de bajo consumo. Sin embargo, como ya se ha comentado, una vez desarrollada la nueva funcionalidad, se analizará el rendimiento. Para ello, se lanzarán

ejecuciones de los programas en Marenostrum. En este caso, el consumo sí es bastante elevado, pero al disponer de muchos nodos ejecutando muchas aplicaciones simultáneas, es difícil precisar el gasto que se destina a este proyecto.

Sin este proyecto, la memoización es más costosa en tiempo, es decir, se tarda más en implementarla. Por tanto, también es más costosa en consumo eléctrico. En conclusión, este proyecto permite ahorrar energía, aunque es imposible cuantificarla. Asimismo, la huella ecológica también se ve reducida por lo argumentado anteriormente.

Por último, la contaminación que se pudiera generar depende del método de producción eléctrica que use el proveedor.

7.4. Matriz de sostenibilidad

¿Sostenible?	**Económica**	**Social**	**Ambiental**
Planificación	Viabilidad económica	Mejora en calidad de vida	Análisis de recursos
Valoración	0	10	10
Resultados	Coste final vs previsión	Impacto en entorno social	Consumo de recursos
Valoración	10	10	10
Riesgos	Adaptación a cambios de escenario	Daños sociales	Daños ambientales
Valoración	0	0	0
Valoración total	**50**		

Cuadro 7: Matriz de sostenibilidad del TFG.

8. Entorno de trabajo y herramientas

Esta sección presenta las herramientas usadas para realizar este proyecto. En primer lugar está el modelo OmpSs, junto con el compilador Mercurium y el sistema de *runtime* Nanos++, que forman una implementación válida del modelo. También se presentarán Extrae y Paraver, herramientas que permiten analizar las ejecuciones de las aplicaciones. Finalmente se presentarán las herramientas de apoyo, como pueden ser Git, para el control de versiones, o GDB para depurar el código.

8.1. OmpSs

En esta sección, a diferencia de la sección 2.1, en la que se introdujo el modelo, se describirá el modelo. Esto implica describir el modelo de ejecución, las dependencias de datos y las herramientas Mercurium y Nanos++ que forman una implementación válida del modelo.

8.1.1. Modelo de ejecución

El sistema de *runtime* de OmpSs crea un grupo de hilos al principio de la ejecución. Este grupo será el *equipo inicial* y estará formado por un único hilo maestro y varios hilos de trabajo. El número de hilos que formará parte de este equipo inicial será determinado por el usuario. Por ejemplo, si el usuario pide cuatro hilos, entonces habrá un hilo maestro y tres hilos de trabajo en el equipo inicial [oB13b][DAB+11].

El hilo maestro, también conocido como hilo inicial, ejecuta el código de usuario de forma secuencial en un contexto implícito conocido como la *tarea inicial*. Esta tarea inicial es circundante a todo el programa. El resto de hilos esperarán hasta que alguna tarea concurrente esté preparada para ser ejecutada [oB13b][DAB+11].

Sólo se usarán múltiples hilos para ejecutar tareas definidas de forma implícita (e.g. #pragma omp for) o explícita (e.g. #pragma omp task), en caso contrario, aunque existan múltiples hilos, el programa se ejecutará de manera secuencial [oB13b][DAB+11].

Por un lado, en el momento en que un hilo encuentra un *#pragma omp for*, el espacio de iteraciones se reparte de acuerdo con la política de planificación. Cada parte del bucle se convierte en una tarea implícita independiente y el equipo cooperará para ejecutar todas las tareas. Cabe destacar la presencia implícita de un *taskwait* al final de un bucle paralelo a menos que esté presente la cláusula *nowait* [oB13b].

Por otro lado, en el momento en que un hilo encuentra *#pragma omp task*, se genera una nueva tarea explícita. Las tareas explícitas son ejecutadas por uno de los hilos del equipo inicial. En consecuencia, la tarea puede ser ejecutada de forma inmediata, es decir, por el mismo hilo que la ha creado, o su ejecución puede ser pospuesta debido a múltiples causas como, por ejemplo, la política de planificación. Además, los hilos pueden elegir suspender la tarea actual para ejecutar una tarea distinta aunque sólo en un *task scheduling point*. Dependiendo del tipo de tarea que sea la tarea suspendida, la terminará el mismo hilo (*tied tasks*) o cualquier otro (*untied tasks*). Se definen los siguientes *task scheduling points* [oB13b]:

- El punto en el que se crea una tarea.
- El punto en el que se encuentra un *taskwait*.
- El punto en el que se encuentra un *taskyield*.
- El punto en el que se completa una tarea.

8.1.2. Modelo de dependencias de datos

Como se refería en la sección 2.1, OmpSs consigue paralelismo asíncrono de flujo de datos mediante el uso de dependencias de datos entre tareas. A menudo, las tareas necesitan datos para realizar cálculos. Es común que una tarea use los datos de entrada para hacer ciertos cálculos que produzcan resultados que más tarde usará otra tarea u otras partes del programa. Por esta razón, el modelo de dependencias de datos que nos proporciona OmpSs es muy útil [oB13a].

La directiva *task* ha sido extendida para soportar las cláusulas *in*, *out* e *inout*, que permiten definir las dependencias de datos entre tareas. Con este mecanismo es posible especificar qué datos necesita una tarea para empezar así como avisar cuando su resultado está listo. Es importante destacar que si la tarea usa o no los datos especificados en las dependencias es responsabilidad del usuario, el modelo esperará hasta que estén listos aunque no se usen [oB13a][DAB+11].

Antes de seguir es necesario introducir el concepto *lvalue*. Un valor L o *lvalue* hace referencia a un objeto que se conserva más allá de una sola expresión. Puede considerar un valor L como un objeto que tiene nombre. Todas las variables, incluidas las variables no modificables (const), son valores L [Mic15]. Veamos con más detalle las tres cláusulas para definir dependencias entre tareas [oB13a][DAB+11]:

- **in(memory-reference-list)**: Si una tarea tiene una cláusula *in* que evalúa a un determinado *lvalue*, entonces la tarea no podrá ser ejecutada hasta que una tarea creada previamente con una cláusula *out* sobre el mismo *lvalue* termine.
- **out(memory-reference-list)**: Si una tarea tiene una cláusula *out* que evalúa a un determinado *lvalue*, entonces la tarea no podrá ser ejecutada hasta que una tarea creada previamente con una cláusula *in* o *out* sobre el mismo *lvalue* termine.
- **inout(memory-reference-list)**: Si una tarea tiene una cláusula *inout* que evalúa a un determinado *lvalue*, se considera como si la tarea tuviera una cláusula *in* y una cláusula *out* sobre el mismo *lvalue*.

Cuando se ejecuta una aplicación OmpSs el sistema de *runtime* usa la información de las dependencias de datos y el orden de creación de cada tarea para realizar un análisis de dependencias. Este análisis produce restricciones sobre el orden de ejecución de las tareas para conseguir un orden correcto de las tareas de la aplicación. Estas restricciones están formadas por parejas. Por ejemplo, hay una dependencia entre T1 y T2 si se cumple lo siguiente [oB13a]:

- T1 ha sido creada antes que T2 y ambas han sido creadas por la misma tarea, es decir, comparten el mismo dominio de dependencias.
- T2 tiene una referencia a memoria que se solapa con una referencia a memoria de T1 y, en ambas tareas, al menos, la referencia solapada se ha especificado en una cláusula de dependencias (in, out o inout) y, al menos, una de esas dependencias es *out*.

Cuando se crea una nueva tarea, las dependencias de entrada y salida se emparejan con las de las tareas ya existentes que correspondan al mismo dominio de dependencias. Si existe una dependencia con otra tarea, ya sea RaW (*Read after Write*), WaW (*Write after Write*) o WaR (*Write after Read*), la nueva tarea se convierte en sucesora de las tareas correspondientes. Este proceso, además, crea un grafo de dependencias entre tareas en tiempo de ejecución. Las tareas se marcan como listas para ejecutar tan pronto como todas sus predecesoras en el grafo han finalizado. En caso de que no tengan predecesoras, se marcan como listas para ejecutar al ser creadas [oB13a].

El ejemplo mostrado en la Figura 11 ilustra cómo funcionan las cláusulas de dependencias de datos.

Listing 4: Dependencias de datos en OmpSs

```
void foo ( int *a, int *b ) {
  for ( i = 1; i < N; i++ ) {
    #pragma omp task in(a[i-1]) inout(a[i]) out(b[i])
    propagate(&a[i-1],&a[i],&b[i]);

    #pragma omp task in(b[i-1]) inout(b[i])
    correct(&b[i-1],&b[i]);
  }
  #pragma omp taskwait
}
```

Figura 11: Fragmento donde se muestra el uso de las cláusulas de dependencias de datos.

Gracias al uso de las cláusulas de dependencias de datos se pueden ejecutar tareas de distintas iteraciones concurrentemente. Todas estas cláusulas permiten *lvalues* extendidos de los permitidos por C/C++. Las dos extensiones permitidas son [oB13a][DAB+11]:

- Secciones de un *array* para referirse a múltiples elementos del *array* en una única expresión. Se pueden expresar las secciones de dos formas:
  a. ***[lower:upper]**. En este caso, se referencian todos los elementos de la sección entre el límite inferior y superior (ambos incluidos). Si el límite inferior no se especifica, se asume que es 0. Si el límite superior no se especifica, se asume que es el último elemento.
  b. ***[lower;size]**. En este caso se referencian todos los incluidos en el rango [lower:lower+(size-1)] (ambos incluidos).
- Las expresiones de *shaping* permiten hacer *recast* de punteros para recuperar el tamaño de las dimensiones que podrían haberse perdido entre las distintas llamadas. Este tipo de expresiones son uno o más **[size]** antes de un puntero.

Se puede ver un ejemplo de estas extensiones en la Figura 12.

Asimismo, también existen las siguientes cláusulas [DAB+11]:

- **copy_in**. Puede ser necesario que el conjunto de datos especificado en esta cláusula deba ser transferido a otro dispositivo, por ejemplo GPU, antes de ejecutar el código asociado.
- **copy_out**. Puede ser necesario que el conjunto de datos especificado en esta cláusula deba ser transferido desde otro dispositivo, por ejemplo GPU, tras ejecutar el código asociado.
- **copy_inout**. Es una combinación de las dos anteriores.
- **copy_deps**. Si se han especificado dependencias, esta cláusula provoca que las dependencias también sean interpretadas como copias. Es decir, *in* también se interpretará como *copy_in*.

Estas cláusulas son importantes no sólo para facilitar la heterogeneidad del modelo, sino porque el mecanismo que se encarga de gestionar las copias será el que se usará para ayudar a implementar la memoización. En consecuencia, por defecto se activa *copy_deps*, de modo que todas las dependencias se interpretarán también como copias.

Listing 5: Expresiones extendidas permitidas en las cláusulas de dependencias de datos

```
void sort (int n, int *a) {
if (n < small) seq_sort(n, a);

#pragma omp task inout(a[0:n/2-1])
     sort(n/2, a);
#pragma omp task inout(a[n/2:n-1])
     sort(n/2, a[n/2]);
#pragma omp task inout(a[0:n/2-1], a[n/2:n-1])
     merge(n/2, a, a, a[n/2]);
#pragma omp taskwait
}
```

Figura 12: Expresiones extendidas permitidas en las cláusulas de dependencias de datos.

Por último, además de las dependencias de datos, OmpSs también ofrece una directiva para establecer puntos de sincronización, la directiva *taskwait*. Cuando el flujo de control llega a un punto de sincronización, espera hasta que todas las tareas creadas anteriormente, en el mismo dominio de dependencias, se completen. OmpSs también ofrece un punto de sincronización sólo sobre ciertas tareas mediante la cláusula *on* de la directiva *taskwait*. Ésta recibe una referencia de memoria y esperará sólo a las tareas que usen la referencia especificada [oB13a].

8.1.3. Mercurium

Mercurium es una infraestructura de compilación *source-to-source*. Actualmente soporta C/C++ y Fortran. Se usa principalmente junto con Nanos++ para implementar OpenMP/OmpSs pero, como es muy extensible, también ha sido usado para implementar otros modelos de programación o transformaciones de compilador como pueden ser *Cell SuperScalar*, *Software Transactional Memory* o *Distributed Shared Memory* [oBa].

Para extender Mercurium sólo es necesario usar un *plugin* de arquitectura. Los *plugins* representan varias fases del compilador. Estos *plugins* se han desarrollado en C++ y los carga dinámicamente el compilador según la configuración [oBa].

Mercurium es capaz de reconocer las directivas de compilador y transformarlas en llamadas al sistema de *runtime*. Además, también es capaz de reestructurar código para diferentes dispositivos como CPUs o GPUs. Para ello contiene un administrador específico para cada dispositivo. Si es necesario, genera código en archivos separados para los distintos dispositivos. Además, puede invocar distintos compiladores dependiendo del dispositivo, como podría ser *nvcc* para compilar código CUDA [oBa].

En la Figura 13 se muestra la estructura de Mercurium.

8.1.4. Nanos++

Nanos++ es una librería de *runtime* diseñada para funcionar en entornos paralelos. Principalmente se usa para OmpSs aunque también tiene módulos para dar soporte a OpenMP y Chapel [oBb].

Nanos++ ofrece varios servicios para dar soporte a paralelismo basado en tareas usando sincronización basada en dependencias de datos. Las tareas se implementan como si fueran hilos a nivel

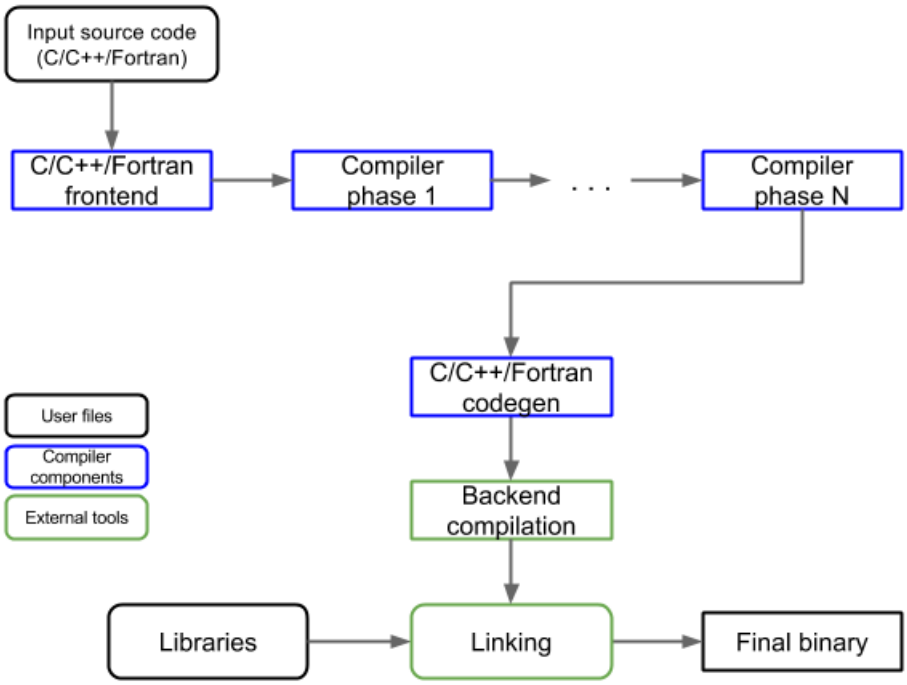


Figura 13: Estructura del compilador Mercurium

de usuario, cuando es posible. También ofrece soporte para mantener la coherencia de memoria entre varios espacios de direcciones distintos (como las máquinas con GPUs) [oBb].

El objetivo principal de Nanos++ es ser usado para investigación de programación paralela. Se ha puesto énfasis en permitir un desarrollo sencillo de distintas partes de la librería para que los investigadores tengan una plataforma donde poder probar distintos mecanismos. Ha sido diseñado para poder ser extendido usando *plugins*. Mediante ellos se puede cambiar la política de planificación, el comportamiento de las dependencias o añadir instrumentación, entre otras cosas. Sin embargo, la extensibilidad y la simplicidad a la hora de programar tienen un coste: el *overhead* del *runtime* ha crecido ligeramente [oBb], pero sólo debería ser un problema en tareas de granularidad muy fina.

En la Figura 14 se muestra la estructura de Nanos++.

8.2. Extrae

Extrae es el paquete dedicado a generar trazas para un análisis posterior. Ésta es una herramienta que utiliza diferentes mecanismos de interposición para inyectar elementos en el código de la aplicación con el fin de recolectar información sobre el rendimiento. Tiene soporte para distintos modelos de programación como, por ejemplo, MPI, OmpSs o CUDA [BSC14a].

Con la instrumentación que nos ofrece esta herramienta, en OmpSs, podemos saber muchas cosas de gran interés. Por ejemplo, cuándo empieza y termina una tarea, qué ha estado haciendo esa tarea en cada momento, comunicaciones entre tareas, etcétera. Esto es especialmente útil cuando una aplicación no tiene el rendimiento esperado. Usando Extrae se pueden identificar cuáles son las causas de la ralentización del programa y atajar los problemas.

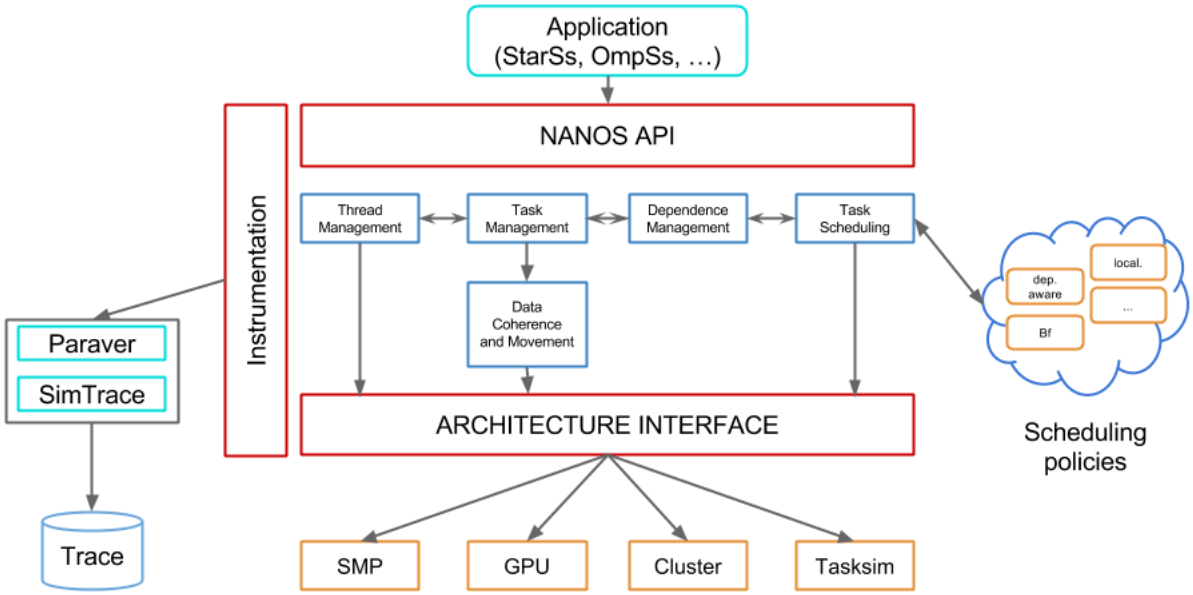


Figura 14: Estructura del sistema de *runtime* Nanos++

8.3. Paraver

Paraver es un analizador de rendimiento basado en trazas con gran flexibilidad para explorar la información recogida por Extrae. Con esta herramienta no sólo es posible detectar los problemas de rendimiento que puedan tener las aplicaciones sino que, además, es posible comprender el comportamiento de las aplicaciones [BSC14b].

En la Figura 15 se puede observa todo el entorno del modelo OmpSs, incluyendo Mercurium, Nanos++, Extrae y Paraver.

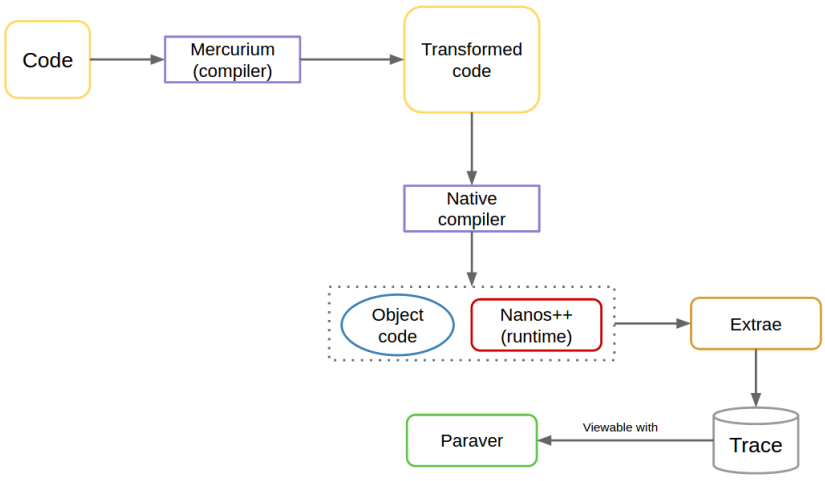


Figura 15: Entorno del modelo OmpSs

8.4. Herramientas de apoyo

Además de las herramientas principales, para llevar a cabo el proyecto han sido necesarias otras herramientas de soporte. En este caso han sido Git y GDB. Se detallan a continuación.

8.4.1. Git

Git es un software de control de versiones diseñado pensando en la eficiencia sobretodo para aquellos proyectos de gran tamaño que tienen un gran número de archivos de código fuente. Ayuda a tener un control más preciso del desarrollo.

8.4.2. GDB

Es un software de depuración que incorporan varias plataformas Unix. Esta herramienta permite trazar y modificar la ejecución del programa, así como alterar y controlar las variables internas del programa para detectar y depurar errores en sus programas.

9. Soporte para memoización de OmpSs

Con el fin de implementar el soporte para memoización de OmpSs hay que hacer extensiones en tres partes:

I. **Modelo OmpSs.** Es necesario ofrecer una nueva sintaxis mediante la cual el usuario pueda demandar la nueva funcionalidad.

II. **Mercurium.** El compilador debe ser capaz de reconocer la nueva sintaxis, procesar los datos y generar el código de memoización usando algunos servicios del *runtime* adecuadamente.

III. **Nanos++.** El sistema de *runtime* debe interactuar con el compilador e implementar las funcionalidades necesarias para llevar a cabo la memoización.

A continuación estas extensiones se explican detalladamente.

9.1. Extensión del modelo OmpSs

Para que el usuario pueda escoger qué métodos quiere memoizar y cuáles no, era necesario introducir una cláusula que permitiera identificar cuándo se usa la memoización y cuándo no. Además, también se requiere que el usuario informe del tamaño de la tabla así como los índices para saber dónde colocar cada resultado en la tabla. Por último, el usuario también debe indicar cuál es el resultado que se debe almacenar.

Teniendo en cuenta todo esto, se introduce en el modelo la sintaxis descrita en la Figura 16. La cláusula *memo* recibirá N parámetros, donde N tiene que ser siempre un número par y como mínimo igual a 2. Los primeros N/2 argumentos serán los tamaños de las distintas dimensiones de la tabla. Los segundos N/2 argumentos son las posiciones de cada dimensión de la tabla donde se almacenará el resultado. Finalmente, usando la cláusula *out* se indica qué se almacenará en la posición indicada de la tabla. Por ejemplo, en el código que se muestra en la Figura 17, *n*

Listing 6: Nueva sintaxis para memoización

```
#pragma omp task memo( dim_size_0, ..., dim_size_N, index_0, ..., index_N )
```

Figura 16: Nueva sintaxis para memoización

será el tamaño de la dimensión 0 mientras que *capacity+1* será el tamaño de la dimensión 1. En otras palabras, tendremos una tabla de *n*capacity+1*. Por otra parte, el resultado, en este caso **sol*, de cada tarea se guardará en la posición (*idx*, *weight*).

9.2. Extensión de Mercurium

Una vez extendido el modelo, al tener una nueva cláusula, lo más importante es que el compilador sea capaz de reconocerla y saber qué hacer con ella. Primeramente, cuando Mercurium lee la cláusula procesa los argumentos que la acompañan y los almacena en unas estructuras de datos. Además, el compilador situará en el código algunas llamadas a ciertos servicios del sistema de *runtime*.

- La primera llamada está destinada a conseguir cierta información para calcular correctamente los índices de la posición en la que se encuentra, o encontrará, el resultado de la tarea actual.

Listing 7: Código de ejemplo correspondiente al problema de la mochila usando el soporte para memoización de OmpSs

```
#pragma omp task memo( n, capacity+1, idx, weight ) out( *sol )
void knapsack(int weight, int idx, int *sol, const item_t item[]) {
  if (idx < 0) {
    *sol=0;
    return;
  }

  if (weight < item[idx].weight) {
    if(idx - 1 >= 0) {
      knapsack(weight, idx - 1, sol, item);
      #pragma omp taskwait
    }
    return;
  }

  int v1=0, v2=0;
  if(idx - 1 >= 0) {
    knapsack(weight, idx - 1, &v1, item);
    if(weight-item[idx].weight >= 0) {
      knapsack(weight - item[idx].weight, idx - 1, &v2, item);
    }
  }
  #pragma omp taskwait
  v2 += item[idx].value;
  *sol = (v1>=v2) ? v1 : v2;
}
```

Figura 17: Código de ejemplo correspondiente al problema de la mochila usando el soporte para memoización de OmpSs

- Una vez disponemos de esa información, el compilador realiza otra llamada para consultar si el resultado deseado ya se encuentra almacenado en la tabla. Si es así, la tarea termina; en caso contrario, se calcula la tarea y se realiza otra llamada.
- Esta última llamada se encarga de almacenar el resultado recién calculado en la tabla.

Este proceso también se muestra en la Figura 18.

Respecto a la información guardada en las estructuras de datos, se envía al sistema de *runtime*. Éste, la leerá y realizará las acciones necesarias, descritas en la sección 9.3.

9.3. Extensión de Nanos++

Para dar soporte a la memoización, Nanos++ debe almacenar mucha información. Por esta razón, una nueva clase se ha añadido al sistema de *runtime*. Además, también se han extendido algunas estructuras de datos ya existentes.

Al crear una tarea, Nanos++ lee la información ofrecida por Mercurium. Si detecta que la memoización está activada, debe comprobar si la infraestructura necesaria está creada. Si todavía no lo está, debe indicar las dimensiones de la tabla para poder crearla más adelante.

Cuando la tarea comienza su ejecución, si todavía no está creada la infraestructura, la crea.

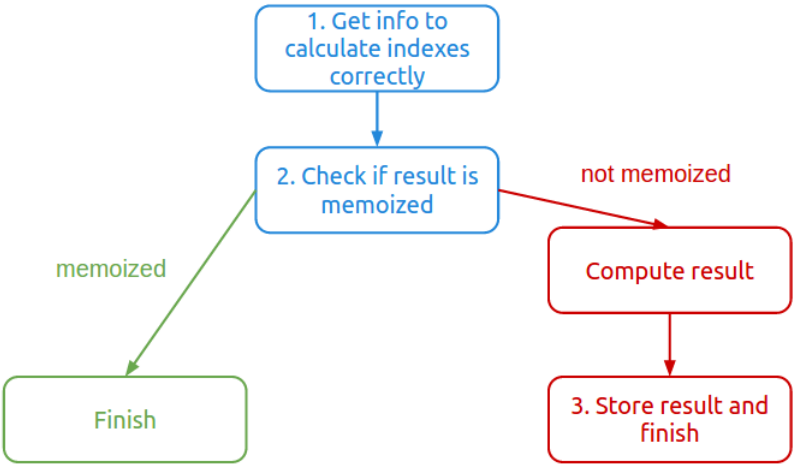


Figura 18: Llamadas adicionales añadidas por Mercurium

Consta de diversas partes:

- **Tabla de memoización.** En esta tabla es donde se almacenarán los resultados. También habrá un byte de presencia, para indicar si un resultado está ya guardado en la tabla o todavía no y otro para saber si ese resultado ya está siendo calculado.
- ***Locks.*** Dado que OmpSs es un modelo de programación paralela, debemos combatir las condiciones de carrera. En consecuencia, debemos asegurar que las operaciones sobre la tabla de memoización son atómicas. Es decir, debemos garantizar que mientras un hilo está escribiendo, ningún otro hilo accede a la misma posición. Para este propósito usaremos *locks*. Éstos hacen que sólo pueda haber un hilo ejecutando simultáneamente una sección de código.
- **Condiciones de sincronización.** Esto es un mecanismo que permite detener una tarea hasta que se cumple una condición. En este caso, si un resultado ya está siendo calculado por un hilo y otro hilo encuentra una tarea que debe calcular lo mismo, esta última se detiene. Cuando el primer hilo ha terminado, avisa a todas las tareas que estuvieran esperando.
- **Tabla de reuso.** Esta tabla sólo se usa en modo depuración. Sirve para ofrecer estadísticas sobre la memoización. Concretamente, almacena las veces que se ha reusado cada posición de la tabla de memoización.

Crear la infraestructura significa:

I. Obtener el tamaño, en bytes, de cada resultado que debe ser almacenado en la tabla.

II. Calcular el tamaño del bloque de memoria que debemos reservar, como se muestra en la Figura 19.

III. Reservar memoria e inicializar toda la tabla a cero. Puede verse en la Figura 20. Se declara *char ** para poder acceder a nivel de byte.

IV. Crear los *locks* y las condiciones de sincronización. Cabe destacar que ambos son mecanismos ofrecidos por Nanos++. Una vez creados, se guardan en sus respectivas estructuras para que puedan ser usados. Habrá un *lock* y una condición de sincronización por cada posición de la tabla. Es especialmente importante la creación de las condiciones de sincronización. En el momento de creación se indica cuál es la condición sobre la que deben esperar. Exactamente se indica cuál es la condición que debe cumplirse para no detener la ejecución. Por ejemplo,

Listing 8: Cálculo del tamaño en memoria de la tabla de memoización

```
//Tamano del resultado que se debe almacenar. Se obtiene usando mecanismos de
    Nanos++.
size_t res_size;
size_t element_size = /*bit de presencia*/ 1 + /*se esta calculando*/ 1 + res_size;
//Numero de elementos de la tabla. Lo recibimos desde Mercurium.
int num_elements;
//Tamano total de la tabla
size_t table_size = element_size * num_elements;
```

Figura 19: Cálculo del tamaño en memoria de la tabla de memoización

Listing 9: Alocatación e iniciliazión de la tabla de memoización

```
//Alocatar tabla.
char * memoization_table = ( char * ) malloc( table_size );
//Inicializar tabla a cero.
memset( memoization_table, 0, table_size );
```

Figura 20: Alocatación e iniciliazión de la tabla de memoización

en la Figura 21, la ejecución se detendrá si *memoization_ table[i*element_ size+1]* es distinto de 0 hasta que la condición cambie y reciba un aviso para despertar.

Listing 10: Ejemplo condición de sincronización

```
//Verificador de condicion de igualdad
EqualConditionChecker<char> ecc( &memoization_table[i*_element_size+1], 0 );
//Condicion de sincronizacion con multiples tareas potencialmente esperando.
//Recibe como argumentos el verificador de condicion y el numero maximo de tareas
    que puede haber esperando.
MultipleSyncCond<EqualConditionChecker<char> > msc( ecc, _num_elements );
```

Figura 21: Ejemplo condición de sincronización

v. Sólo en caso de que esté en modo depuración se reservará memoria para la tabla de reuso y se inicializará toda a 0. Se muestra en la Figura 22

Una vez creada la infraestructura, la tarea empieza a realizar su trabajo. Se ofrecen varios servicios para ser llamados desde Mercurium, detallados a continuación.

- ***nanos_ memo_ get_ dimensions***. Este método es un consultor de las dimensiones de la tabla de memoización, que se usan para calcular la posición de la tabla que corresponde al resultado de la tarea actual. Recibe como argumento la tarea actual.
- ***nanos_ memo_ task_ memoized***. Este servicio consulta si el resultado de la tarea ya está almacenado en la tabla. En caso afirmativo, lee el resultado y lo devuelve. Si no está almacenado, comprueba si algún otro hilo está calculando el resultado. Si es así, queda esperando hasta que el otro hilo termina. Si nadie lo está calculando, asume el cálculo. Recibe como parámetros la tarea actual y el índice donde debe estar el resultado.
- ***nanos_ memo_ store_ task***. Con este método se almacena el resultado en la tabla. Tam-

Listing 11: Alocatación e iniciliazión de la tabla de reuso

```
#ifdef NANOS_DEBUG_ENABLED
//Alocatar tabla
//Como es una tabla para contar los accesos a una posicion, usamos enteros.
int * reuse_table = ( int * ) malloc( num_elements );
//Inicializar tabla a cero.
memset( reuse_table, 0, num_elements*( sizeof ( int ) );
#endif
```

Figura 22: Alocatación e iniciliazión de la tabla de reuso

bién se despierta a las tareas que estuvieran esperando este resultado. Recibe como parámetros la tarea actual y el índice donde debe estar el resultado.

Primero, usando el método *nanos_memo_get_dimensions* obtiene las dimensiones de la tabla. Las usará para calcular la posición de la tabla que corresponde al resultado.

Con el índice, lanza una consulta mediante el servicio *nanos_memo_task_memoized*. Si el resultado ya se encuentra la tabla, se copia automáticamente en la variable estipulada en la cláusula *out*. Si todavía no se encuentra, se comprueba si ya hay alguien calculando ese resultado. De ser así, esta tarea quedará esperando hasta que el resultado esté disponible. En cuanto reciba el aviso para despertarse, copiará el resultado en la variable de la cláusula *out*. Además, si se ejecuta en modo depuración, este servicio también incrementará la posición correspondiente de la tabla de reuso.

Si el resultado no estuviera siendo calculado por nadie, asume el cálculo el hilo actual. Debe indicar que lo va a calcular para que el resto de tareas que requieran ese resultado queden a la espera. Tras calcular el resultado, lo almacena en la tabla llamando a *nanos_memo_store_task*. Lo que hace esta función es copiar el resultado de la variable estipulada en la cláusula *out* a la posición correspondiente de la tabla. Además, marca la tarea como memoizada y despierta a las tareas que estuvieran esperando.

Finalmente, cuando el padre termina, destruye todo lo correspondiente a la memoización y libera toda la memoria. Cabe remarcar que es el padre quien debe hacerlo puesto que es el último en terminar, ya que él no terminará hasta que todos sus hijos hayan acabado.

9.4. Versiones

Como se detallaba en la sección 4.1.4, este proyecto se ha desarrollado de una forma incremental. Se realizaban avances, se evaluaban, se identificaban los campos de mejora y se intentaban solventar. Por esta razón, se han obtenido distintas versiones de la herramienta hasta llegar a la versión final. A continuación se detallan las más reseñables.

9.4.1. Versión 1. Memoización global

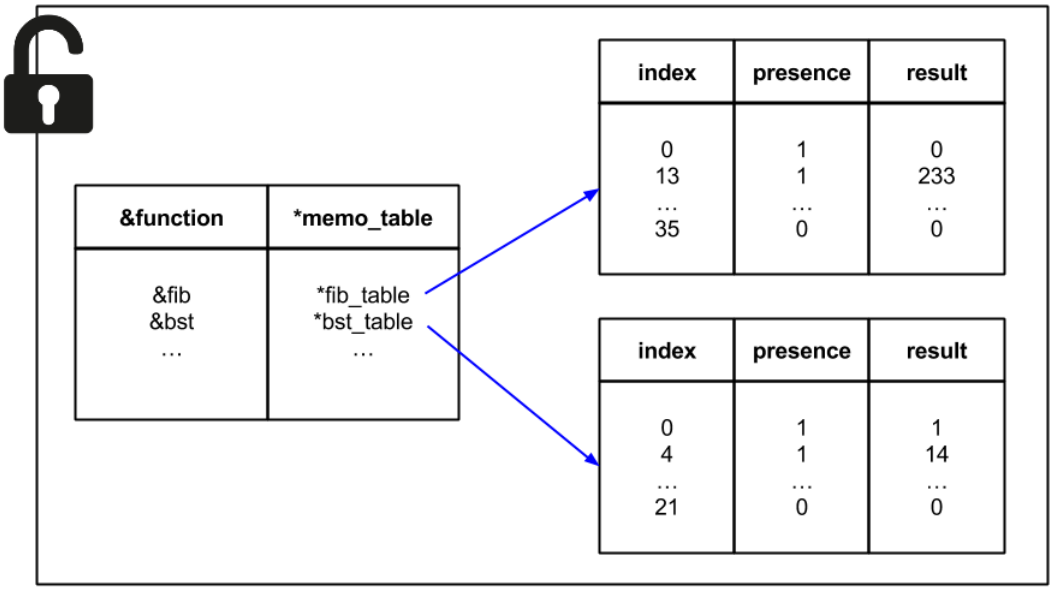


Figura 23: Memoización global

La primera versión que fue implementada contemplaba que la tabla de memoización se mantuviera viva durante toda la ejecución del programa. En esta primera versión en realidad hay varias tablas. La primera tabla es clave-valor: la clave es la dirección de la función que debe ser memoizada y el valor es el puntero a la tabla de memoización de esa función. En esta primera tabla se almacenan los punteros a las tablas de memoización de las distintas funciones que puede haber en una misma aplicación. El resto de tablas son las propias tablas de memoización donde se almacenan los resultados y los respectivos bytes de presencia. Se puede observar en la Figura 23.

En esta primera versión no hay más que un único *lock* para todas las tablas. En consecuencia, hay un gran *overhead*, ya que sólo puede haber un hilo accediendo a la infraestructura concurrentemente; aunque estén accediendo a posiciones distintas de la misma tabla o incluso a tablas de distintas funciones.

Además, tampoco hay condiciones de sincronización. En otras palabras, si un hilo está calculando el resultado de una tarea y llega otro hilo que debe calcular el mismo resultado, como todavía no está almacenado en la tabla, el hilo que llegó más tarde volverá a calcularlo de nuevo. Esto significa que algún resultado se sigue calculando múltiples veces, que es exactamente lo que pretende evitar la memoización.

Por último, tras ejecutar algunas aplicaciones de ejemplo, nos dimos cuenta de que, en algunos casos, el resultado de la herramienta podría ser incorrecto. Por ejemplo, el código que se muestra en la Figura 24. La herramienta almacenará los resultados para el conjunto *itemsA*. Si durante la ejecución de la función para el conjunto *itemsB* coinciden los índices con alguna posición que ya hubiera sido calculada para *itemsA*, ese resultado será incorrecto y potencialmente podría fallar toda la aplicación.

Este último problema podría resolverse permitiendo que el usuario vaciara la tabla de memoización mediante alguna directiva. Sin embargo, pensamos que eso sería dar demasiada responsabilidad al usuario y queremos que la herramienta sea sencilla y fácil de usar, de modo que

el usuario debe tener la mínima responsabilidad. También podría solucionarse este problema almacenando también todos los parámetros de la función, de modo que no habría confusión. No obstante, esta solución significaría usar demasiada memoria y realizar demasiadas copias que desembocarían en un gran *overhead*.

Por todas estas razones, decidimos mejorar la herramienta e implementamos una segunda versión.

9.4.2. Versión 2. Memoización local en tarea con un único *lock*

Listing 12: Código con resultado erróneo

```
#pragma omp task memo( n, capacity+1, idx, weight ) out( *sol )
void knapsack(int weight, int idx, int *sol, const item_t item[]) {
  if (idx < 0) {
    *sol=0;
    return;
  }

  if (weight < item[idx].weight) {
    if(idx - 1 >= 0) {
      knapsack(weight, idx - 1, sol, item);
      #pragma omp taskwait
    }
    return;
  }

  int v1=0, v2=0;
  if(idx - 1 >= 0) {
    knapsack(weight, idx - 1, &v1, item);
    if(weight-item[idx].weight >= 0) {
      knapsack(weight - item[idx].weight, idx - 1, &v2, item);
    }
  }
  #pragma omp taskwait
  v2 += item[idx].value;
  *sol = (v1>=v2) ? v1 : v2;
}

int main () {
  item_t itemsA[10];
  item_t itemsB[10];
  int sol1 = 0;
  int sol2 = 0;

  //Se realizara la memoizacion correctamente.
  knapsack( 50, 9, &sol1, itemsA );
  //El resultado correspondera al conjunto de itemsA, de modo que sera incorrecto.
  knapsack( 50, 9, &sol2, itemsB );
}
```

Figura 24: Código con resultado erróneo

Para intentar corregir los problemas detectados en la primera versión se implementó esta segunda. Principalmente corrige el problema que puede desembocar en un resultado erróneo. Para ello se cambia la estructura y, consecuentemente, también cambia el tiempo de vida de la tabla de memoización.

En esta segunda versión, el objeto que representa la memoización se almacena dentro de la tarea.

Esta herramienta está pensada para memoizar funciones con estructura recursiva. Esto significa que tenemos una primera tarea (Padre) que creará varias tareas (hijos) que crearán varias tareas (nietas), etcétera. Esto es importante y es algo que se toma en consideración en esta segunda versión y en las sucesivas. A partir de esta versión, la memoización sólo durará hasta que termine la tarea que la inició. Es decir, cuando termina el padre, destruye el objeto que representa la memoización. Se muestra en la Figura 25. De este modo, si cambian los argumentos de una función no importa porque serán dos tablas de memoización distintas.

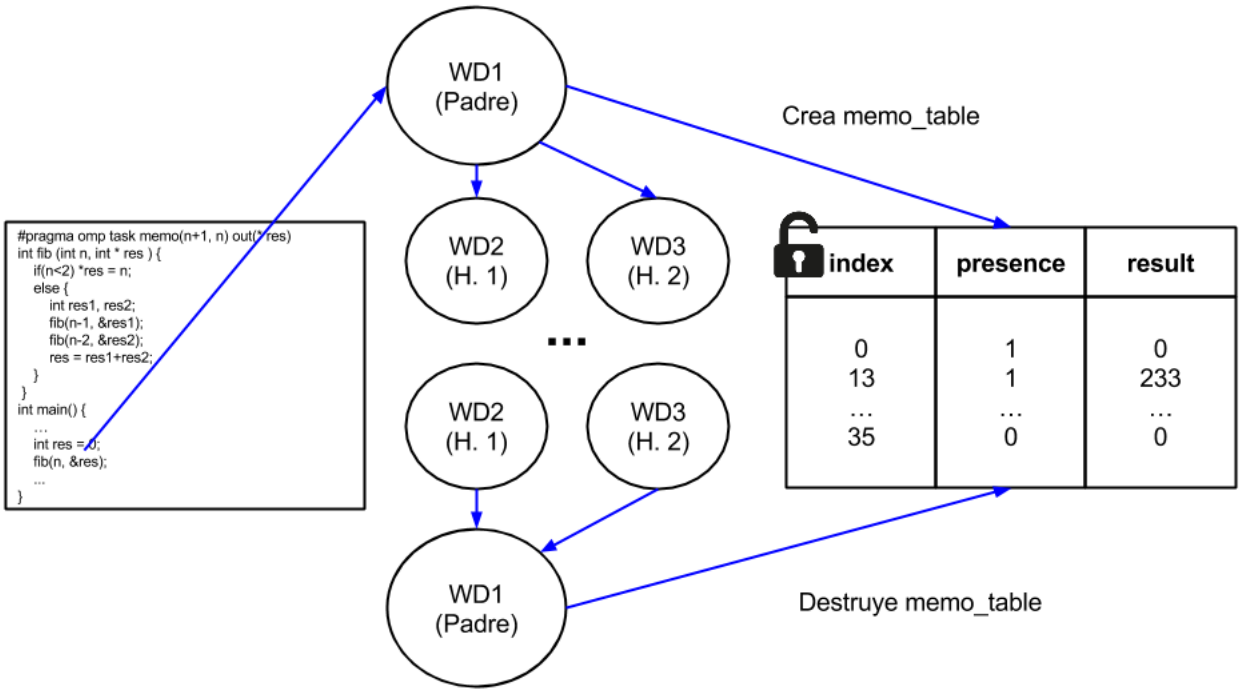


Figura 25: Memoización local en tarea no persistente con un único *lock*

Además, eliminamos la primera tabla. Ya no es necesaria porque cada tarea padre tendrá su tabla, y los descendientes la heredarán de su padre, de modo que no necesitamos una estructura que nos indique qué tabla corresponde a cada función. La parte más positiva de este cambio, es que ahora el *lock* sólo se aplicará sobre una tabla y no sobre varias, como sucedía en la versión anterior. Esto se traduce en una mejora de rendimiento. Sin embargo, no significa demasiado dado que seguimos teniendo un único *lock* para toda la tabla de la misma función que es lo realmente costoso.

Asimismo, no usaremos tanta memoria, puesto que sólo habrá una tabla en memoria y no varias como en la primera versión.

A pesar de haber solucionado el problema más grave, esta versión sigue teniendo problemas de rendimiento. Tener un único *lock* para toda la tabla es un gran problema, sobretodo cuando el número de hilos es elevado. Esto es debido a que todos los hilos estarán peleando por acceder a la tabla, aunque sea a posiciones distintas y no conseguiremos aprovechar el paralelismo.

Por otra parte, también sigue existiendo el problema que provoca que se tenga que repetir trabajo en algunas ocasiones, explicado detalladamente en la sección 9.4.1, que también se traduce en bajo rendimiento.

En conclusión, la segunda versión de esta herramienta funciona correctamente en todos los casos probados. Incluso en aquellos que cambian sus parámetros, como el de la Figura 24, a diferencia de la primera versión. Sin embargo, tiene ciertos problemas de rendimiento que deben ser resueltos.

9.4.3. Versión 3. Memoización local en tarea con un *lock* por elemento

Como se mencionaba en la sección 9.4.2, uno de los principales problemas de rendimiento que presentaban las versiones 1 y 2 era el hecho de tener un único *lock*. En la segunda versión, había una tabla entera a la que no se podía acceder concurrentemente porque el hecho de tener un sólo *lock* no lo permitía. Por tanto, en esta versión se pretende solucionar ese problema.

Primero es necesario analizar si la situación permite mejorar el problema. Las posiciones de la tabla son independientes entre ellas, de modo que si hay varios hilos accediendo a distintas posiciones no hay ningún problema. El problema sólo existe cuando se escribe en una posición y alguien más quiere acceder a esa misma posición.

Por tanto, tras analizar la situación, comprobamos que podemos dividir el *lock* que protege toda la tabla en *locks* individuales para cada una de las posiciones de la tabla, como se muestra en la Figura 26. De este modo, los hilos podrán acceder concurrentemente a la tabla, excepto cuando quieran acceder a la misma posición. En consecuencia, aumenta el paralelismo y disminuye el *overhead*.

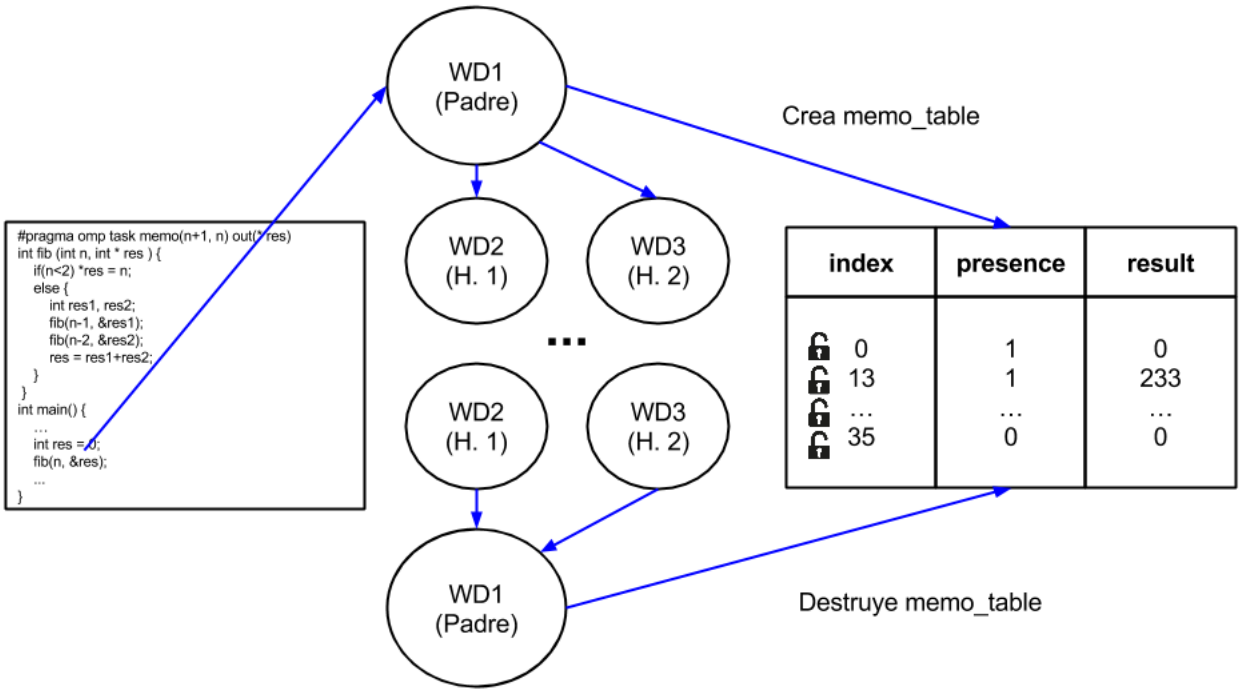


Figura 26: Memoización no persistente con un *lock* por elemento

Tras realizar varias ejecuciones de las aplicaciones de evaluación de rendimiento, realmente se notaba una gran mejora. Sin embargo, por otra parte, nos dimos cuenta de que en muchas

ocasiones se estaba malgastando una gran cantidad de memoria.

En estas tres primeras versiones, se alocata un gran bloque de memoria al principio de la ejecución de la función memoizada. Se crea toda la infraestructura necesaria para todos los elementos que pueden caber en la tabla a pesar de que, usando el modo depuración y mirando la tabla de reuso se puede comprobar que en muchas ocasiones son pocas las posiciones de la tabla que se usan. Puede observarse en la Figura 27, donde hay una gran cantidad de ceros, que indican que esa posición no se ha reusado nunca y por tanto, no hubiera sido necesario crear la infraestructura para ella.

5	3	4	0	0	0	0	0	1	7	0	0	6	0	0	0	0	0	4	0
0	1	4	0	0	0	2	0	3	2	0	0	8	3	0	1	3	0	0	0
0	0	0	5	0	0	0	1	0	2	3	0	0	0	0	0	6	4	0	0
0	0	2	4	0	0	0	0	6	0	2	0	0	0	0	0	0	1	0	3
0	5	2	1	3	0	0	0	0	0	0	0	0	0	0	0	0	3	4	0
3	3	5	0	0	0	0	1	0	0	0	2	3	0	0	0	0	0	5	6
6	0	0	0	0	0	0	8	9	0	2	0	3	0	0	0	3	0	1	0
0	0	0	0	0	2	4	0	3	3	0	1	0	0	0	0	3	6	0	0
4	2	0	0	0	0	0	0	0	0	4	0	0	0	0	6	2	0	0	1
0	0	0	0	4	3	0	0	0	2	0	0	6	9	0	3	0	0	0	0

Figura 27: Tabla de reuso de una aplicación con muchas posiciones sin usar

Concluyendo, tenemos una versión con rendimiento mejorado, que funciona para todos los casos, pero que no aprovecha bien la memoria. Es más, malgasta, en muchos casos, una gran cantidad de memoria que no llega a ser usada nunca.

9.4.4. Versión 4. Memoización local en tarea usando *std::unordered_ map*

En esta cuarta versión se pretende enfrentar el problema relatado en la sección 9.4.3: se malgasta memoria en muchas ocasiones. Esta herramienta no sólo pretende obtener buen rendimiento, sino también aprovechar de la mejor forma posible los recursos de las máquinas.

De nuevo, es necesario realizar un análisis de la situación para determinar si es posible mejorar el uso de los recursos. Con el diseño de las tres primeras versiones, no es posible reducir el uso de memoria. Sin embargo, se puede usar un enfoque distinto: en vez de usar una gran tabla que contenga toda la infraestructura para todos los elementos desde el principio, se puede usar un contenedor de la librería estándar de C++. Este contenedor será el *unordered_ map*.

Las razones que llevan a escoger este contenedor en lugar de otros son varias. En primer lugar, es un contenedor clave-valor, así que podemos hacerlo funcionar igual que un *array* convencional, simplemente es necesario que la clave sea el índice que le correspondería usando el *array*. Por otra parte, la gran ventaja que ofrece respecto a la tabla es que no hace falta crear la infraestructura para todos los elementos desde el principio. Este contenedor permite ir insertando sólo aquellos elementos que necesitemos, en el momento que sea necesario. Además, en promedio, el tiempo para acceder a un elemento es constante. En el peor caso el tiempo es lineal.

Otra ventaja que presenta usar este contenedor es la escalabilidad. La clave puede cambiar en cualquier momento y si es necesario dejar de usar índices numéricos para usar otro tipo de

índices, no requiere un cambio profundo. Además, la clave puede ser cualquier *Template*, de modo que nos permitiría incluso usar estructuras complejas creadas por nosotros mismos, de modo que se ajuste lo máximo posible a lo que buscamos.

Cabe destacar el hecho de que usamos un *unordered_map*, es decir, no está ordenado. Esto es porque siempre hacemos inserciones de un único elemento. Entonces, si usarámos un *map* normal, como en cada inserción debe ordenar los elementos, la complejidad de la operación de inserción es logarítmica. En contraste, usando un *unordered_map*, en promedio, la operación de inserción tiene complejidad constante, aunque en el peor caso la complejidad es lineal.

También es reseñable el hecho de que en esta implementación, los elementos ya no son contiguos en memoria. Ahora en cada inserción se reserva el espacio necesario para guardar el resultado correspondiente y en el *map* de memoización guardamos ese puntero. Es importante porque puede afectar a la caché. En las versiones anteriores, al estar todos los resultados contiguos en memoria, es posible que en una misma línea de caché hubiera más de un resultado, de modo que si escribías una nueva posición de la tabla, significaba invalidar toda la línea aunque sólo hubiera cambiado un resultado. En las versiones anteriores se intenta paliar este efecto usando *padding*. Sin embargo, la contrapartida es que si cargabas un resultado, al traer toda la línea, podía pasar que el siguiente resultado que buscaras ya estuviera en caché y ahorrar el acceso a memoria.

En esta implementación, pues, tal como se muestra en la Figura 28, tenemos un *unordered_map* que mantiene los índices numéricos que se calculaban en versiones anteriores como clave. Por otro lado, el valor es un puntero a la posición de memoria donde se encuentra el resultado que corresponde a ese índice.

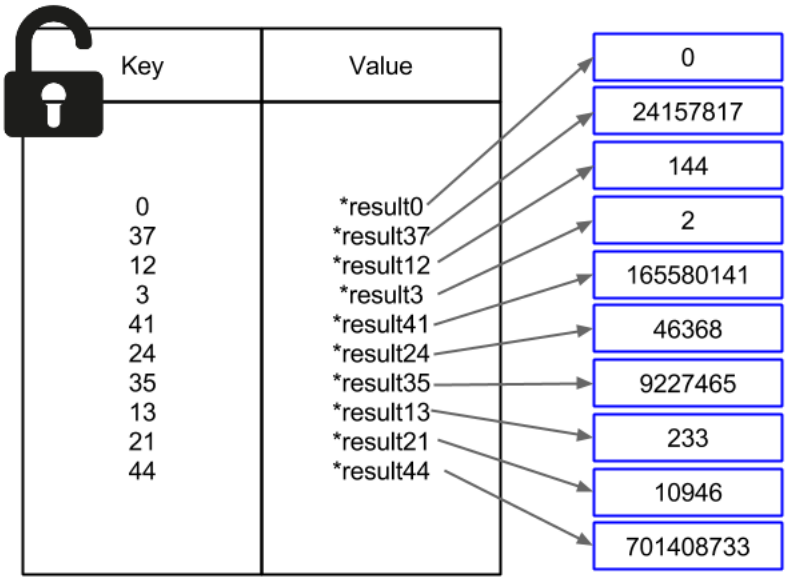


Figura 28: Memoización no persistente usando *std::unordered_map*

No obstante, el *map* presenta una gran desventaja. Debido a la forma en la que está implementado en la librería estándar de C++, no es un contenedor seguro respecto a las condiciones de carrera. Mientras en la tabla el único problema era que no se podía acceder a una posición que estaba siendo escrita, en este caso, si algún hilo está escribiendo, sea cual sea la posición, ningún otro

hilo puede acceder al contenedor. Por tanto, de nuevo, sólo habrá un único *lock* para todo el *map*. Esto presenta problemas de rendimiento, aunque se ven contrarrestados por el tiempo que ahorramos al no tener que crear toda la infraestructura para todos los elementos, aunque no vayan a ser usados.

Resumiendo, esta versión presenta un rendimiento similar a la versión 3, pero mejora el uso de la memoria. A pesar de esto, aun queda una vía de mejora que se explica en la sección 9.4.1 y que se vuelve a mencionar en la sección 9.4.2. Se trata de que en algunas ocasiones se repite trabajo porque mientras una tarea está calculando un resultado, otra tarea mira si ese resultado ya está en la tabla, pero como todavía se está calculando, la tarea que llegó más tarde lo vuelve a calcular.

9.4.5. Versión 5. Memoización local en tarea usando condiciones de sincronización

En esta última versión se enfrenta el problema detallado en la sección 9.4.1 y mencionado en las secciones 9.4.2 y 9.4.4. Este es un problema realmente importante tanto por rendimiento como por concepto.

En primer lugar, la memoización trata de evitar que se repita trabajo, de modo que si lo estamos haciendo, aunque sea dentro del *runtime*, por el motivo que sea, no estamos consiguiendo el objetivo.

Por otra parte, esto afecta en gran medida al rendimiento. Primero por el simple hecho de tener que recalcular de nuevo el trabajo, que implica un tiempo. Pero además, la estructura de los problemas y de Nanos++ hace que la penalización sea aún mayor. En el tipo de problemas que buscamos resolver, recursivos, no encontrar un resultado puede significar también no encontrar los resultados de las recursiones y realizar un árbol entero de recursión que otro hilo está realizando a la vez. Es decir, no es sólo una simple tarea la que se recalcula, si no que puede ser incluso una gran parte del árbol de recursión, con todas las tareas que ello implique. Además, Nanos++ debe crear una tarea cada vez que encuentra un *#pragma omp task*, que tampoco es gratuito, ni en tiempo, ni en memoria. En otras palabras, si creamos demasiadas tareas, podemos estar gastando mucho tiempo simplemente en la creación de tareas en vez de en el cálculo de éstas. Máxime cuando las tareas que suelen ejecutar estos problemas tienen poco trabajo y, en algunos casos, puede tardar más la creación que la propia tarea.

Por todo lo explicado, es esencial evitar este problema. Para ello se van a añadir dos cosas a la infraestructura presentada en las versiones anteriores:

- Para cada posición de la tabla, además del byte de presencia, habrá otro byte que indique si hay alguna tarea calculando el resultado que corresponde a esa posición.
- Para cada posición de la tabla, habrá una condición de sincronización. Se encargará de comprobar si hay alguna tarea calculando el resultado. Si es así, esperará hasta que la tarea que estuviera calculando el resultado avise de que ya está disponible. En caso contrario, marcará que ese resultado se está calculando y asumirá el cálculo.

Con estos dos nuevos elementos corregimos el problema. Cuando una tarea consulta si el resultado deseado está en la tabla y recibe una negativa, consulta si alguna otra tarea lo está calculando. Si es así, la condición de sincronización pone la tarea en espera hasta recibir el aviso de que la condición para despertar se cumple. Este aviso lo dará la tarea que estaba calculando al resultado, una vez ya esté guardado en la tabla y marcado como presente. Este flujo se muestra en la Figura 29.

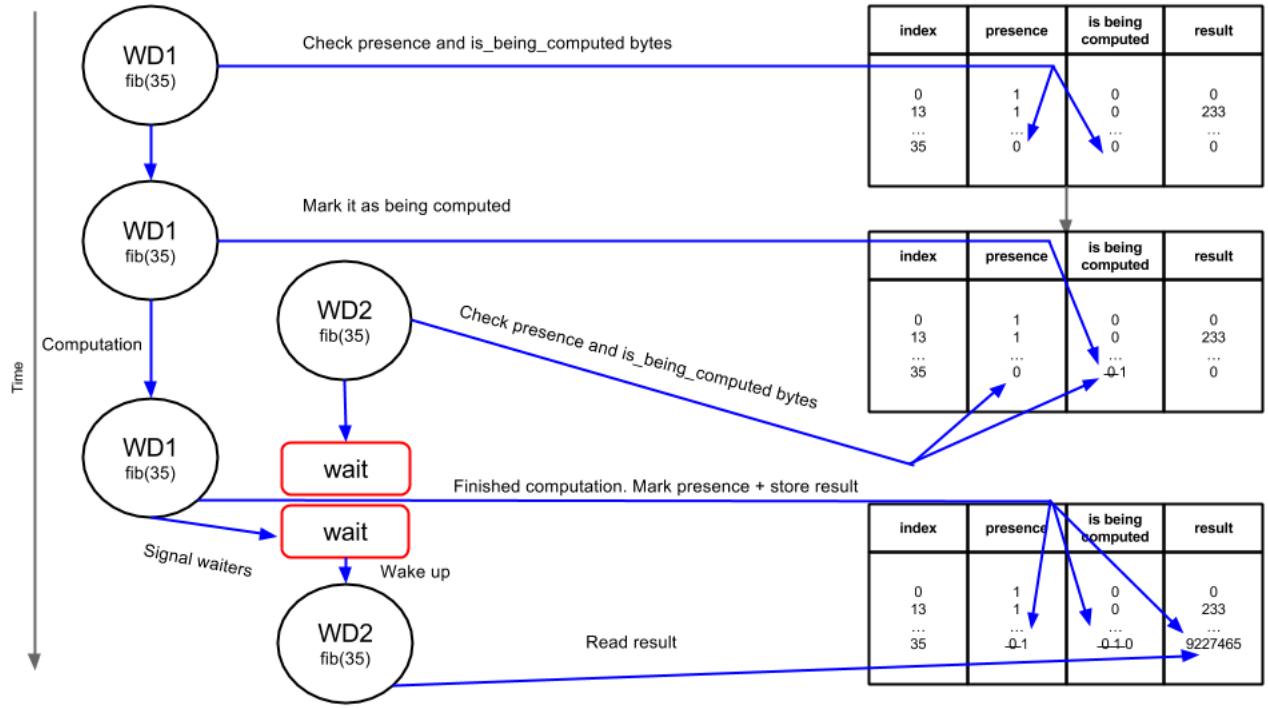


Figura 29: Flujo de memoización usando condiciones de sincronización

Así pues, ya no se crean más tareas que aquellas necesarias y no se repite trabajo. De modo que corregimos el problema tanto en concepto como en rendimiento.

Esta optimización se ha realizado tanto para la versión usando *std::unordered_map* como para la versión que usa una tabla.

10. Diseño e implementación de un marco de evaluación de rendimiento

Una vez tenemos desarrollado el soporte para memoización debemos evaluar su rendimiento para determinar si se cumplen los objetivos. En esta sección se detalla el proceso que se ha seguido para diseñar e implementar un marco de evaluación de rendimiento que permita decidir si las metas se han cumplido.

10.1. Análisis de propiedades

Lo primero que hay que hacer es un análisis sobre la herramienta y las propiedades que deben satisfacer los problemas para evaluar el rendimiento. Las dos propiedades básicas que deben cumplir los problemas para formar parte de este marco son las detalladas en la sección 2.2.3:

- **Subestructura óptima**
- **Subproblemas superpuestos**

Sin estas dos propiedades, no se pueden aplicar técnicas de programación dinámica, de modo que los problemas que no cumplan ambas propiedades quedan directamente descartados.

Además, el diseño de la herramienta añade más requerimientos. Los problemas deben ser recursivos. Existen muchos problemas que se pueden resolver aplicando programación dinámica de forma iterativa, pero esos también quedan descartados. Esto es debido a que en nuestra implementación, la tabla de memoización se va heredando del padre, pero si la implementación no es recursiva, no hay padre y por tanto cada tarea tendría una tabla propia en vez de compartirla entre todos.

Asimismo, para hacer más rica la comparación, es importante tener problemas con distintas estructuras. Algunos como el cálculo de la secuencia de Fibonacci, donde el número de tareas reusadas es muy alto pero el paralelismo muy pequeño. Otras como encontrar la subsecuencia creciente más larga de una secuencia, donde el número de tareas reusadas es más reducido, pero el paralelismo posible es más grande.

Viendo la Figura 30 es sencillo observar como el árbol es siempre muy estrecho, no gana amplitud para abrir paralelismo y enseguida unas tareas y otras llaman a las mismas subtareas.

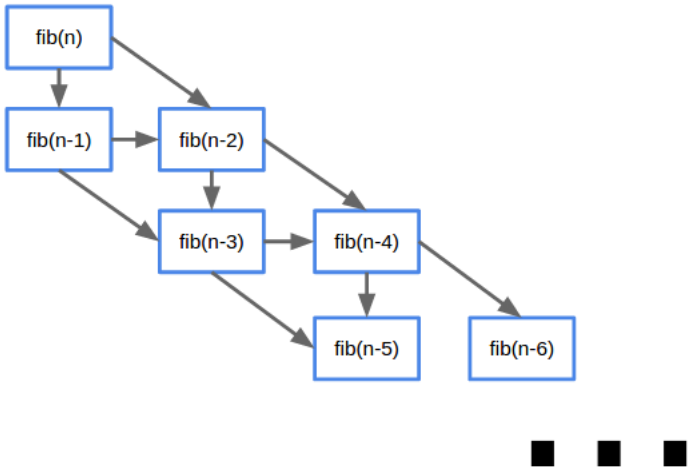


Figura 30: Árbol de recursión para el cálculo de la secuencia de Fibonacci

En contraste, en la Figura 31 es muy distinto. A cada nivel de profundidad el árbol se hace más ancho, y hay pocas tareas que se repiten, de modo que se saca más provecho del paralelismo.

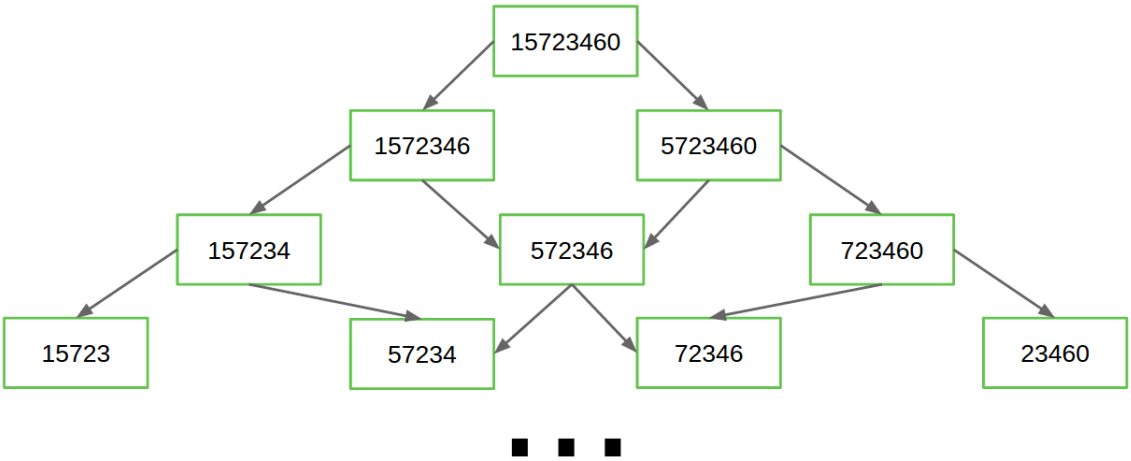


Figura 31: Árbol de recursión para encontrar la subsecuencia creciente más larga de una secuencia

Teniendo varios problemas con estructuras distintas podremos ver cómo se comportan tanto la memoización como el paralelismo y ver de qué técnicas se obtiene más beneficio y en qué problemas se obtiene una ganancia de rendimiento más alta.

10.2. Selección e implementación de problemas

En esta sección se procederá a explicar los problemas que formarán parte del marco de evaluación de rendimiento. Además, se darán algunos detalles de la implementación.

10.2.1. Selección

Teniendo en cuenta los requerimientos descritos en la sección 10.1 realizamos una búsqueda de problemas que pudieran encajar. La intención era tener en consideración los problemas clásicos de programación dinámica para tener referencias con las que compararnos. Finalmente los problemas seleccionados para formar parte del marco fueron los siguientes:

1. ***Binary Search Tree***. Suponiendo que estás construyendo un *Binary Search Tree* de N nodos con valores 1..N: ¿cuántos árboles estructuralmente distintos puede haber que contengan esos valores? Dado el número de valores distintos, se debe computar el número de árboles estructuralmente distintos que contengan esos valores [Rol11].

   Por ejemplo: bst(3) debe ser 5 porque hay 5 árboles estructuralmente únicos que contienen 1, 2 y 3, como se muestra en la Figura 32.

2. ***Coin change***. Dado un conjunto de monedas $m_1, m_2, ..., m_n$ y una cantidad de dinero d, ¿cuántas formas distintas hay de dar el cambio? Es decir, de cuántas formas distintas podemos conseguir la cantidad d con las monedas disponibles. Cabe decir que se supone que tenemos cantidad infinita de cada tipo de moneda. Por ejemplo: coin_change para $d = 5$, con el conjunto $1, 2, 3$ es 5, como se muestra en la Figura 33.

3. ***Cutting a rod***. Dada una vara de longitud N y un conjunto de precios que contiene precios para piezas de todos los tamaños menores que N, hay que determinar el máximo

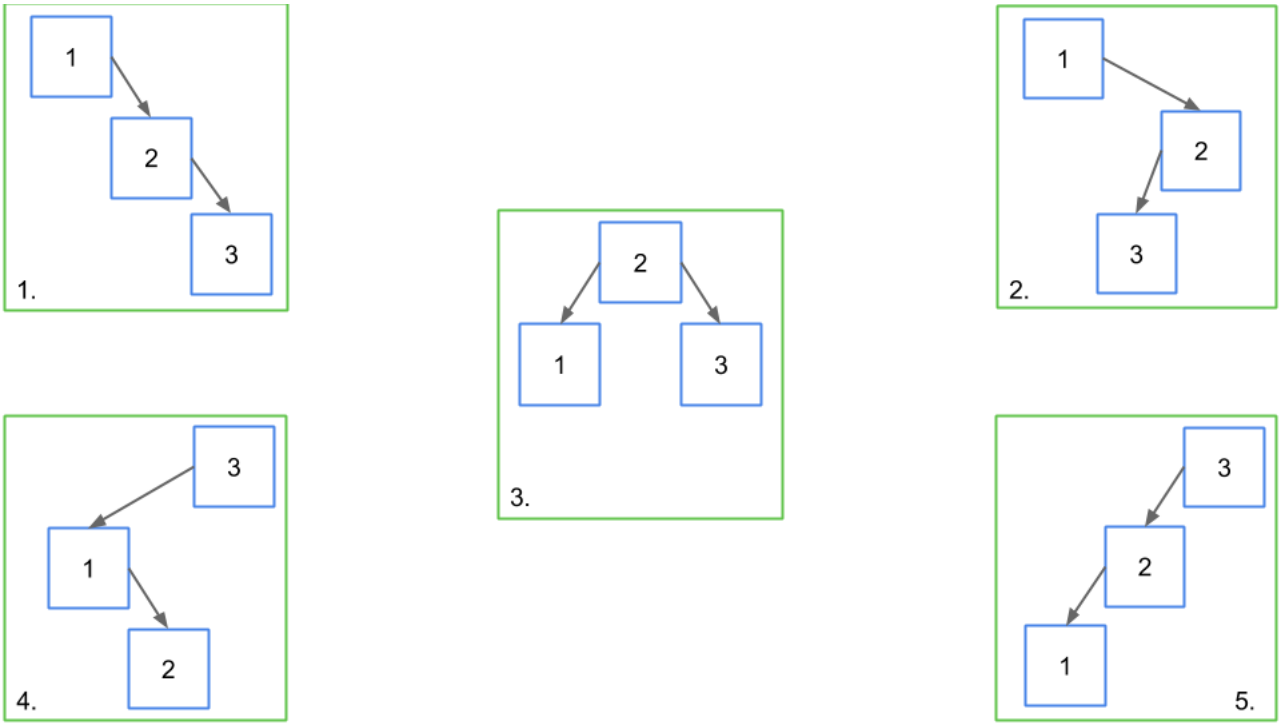


Figura 32: Solución de *Binary Search Tree* para N=3

valor que se puede obtener cortando la vara y vendiendo las piezas. Por ejemplo: si $N = 8$ y el conjunto de precios es el del Cuadro 8, la solución es 22, cortando la vara en dos piezas, una de 2 y otra de 6.

Longitud	1	2	3	4	5	6	7	8
Precio	1	5	8	9	10	17	17	20

Cuadro 8: Conjunto de precios de ejemplo para el problema *Cutting a rod*.

4. ***Egg Dropping Puzzle***. Supongamos que deseamos saber qué plantas de un edificio de K plantas son seguras para dejar caer un huevo y desde cuáles se romperá el huevo al dejarlo caer. Haremos algunas suposiciones:

   - Un huevo que sobrevive a una caída se puede usar otra vez.
   - Un huevo que se rompe se descarta.
   - El efecto de una caída es el mismo para cualquier huevo.
   - Si un huevo sobrevive desde una planta X sobrevivirá desde cualquier planta menor que X.
   - No se descarta que un huevo pueda romperse al dejarlo caer desde la primera planta, al igual que no se descarta que no se rompa al dejarlo caer desde la planta más alta.

   Si sólo dispusiéramos de un huevo, sólo hay una forma de realizar el experimento: empezando desde la planta más baja e ir subiendo hasta que se rompa. En el peor de los casos, se necesitarían K lanzamientos. Sin embargo, si dispusiéramos de dos huevos, cuál es el mínimo número de lanzamientos con el que obtendríamos el resultado correcto? Este problema trata de calcular cuál es el menor número de lanzamientos con el que se conseguiría encontrar la planta crítica (la primera desde donde el huevo se rompe) en el peor caso.

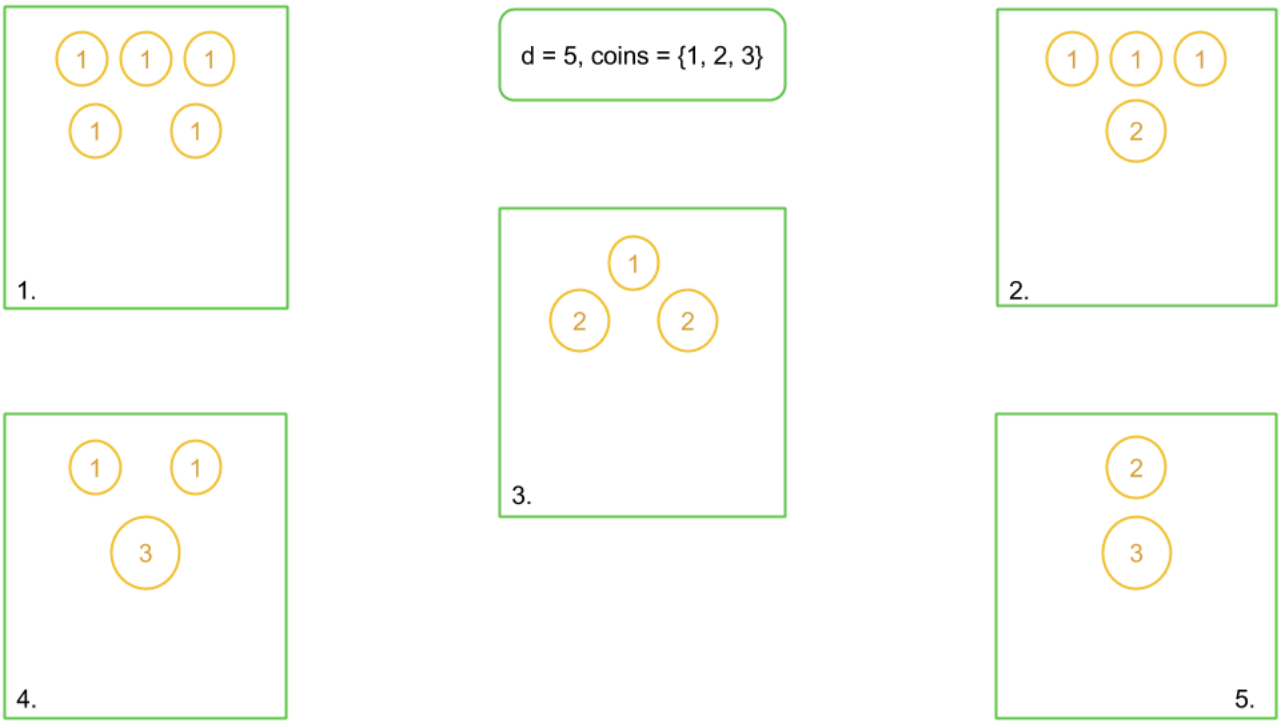


Figura 33: Solución de *Coin change* para $d = 5$, con el conjunto $1, 2, 3$

Por ejemplo, si disponemos de 2 huevos y un edificio de 10 plantas, el mínimo número de intentos en el peor caso es 4.

5. **Sucesión de Fibonacci**. Este es un problema muy conocido. Cada término de la sucesión de Fibonacci se calcula sumando sus dos términos anteriores. Empieza con los términos fib(1)=1, fib(2)=1. Particularmente, este es un problema muy duro para OmpSs puesto que sólo realiza una suma, entonces es difícil compensar el tiempo de creción de tarea, máxime cuando no se puede conseguir aprovechar mucho el paralelismo, como se explicaba en la sección 10.1.

   Por ejemplo, los 10 primeros términos de la sucesión de Fibonacci son:

   $fib(1) = 1, fib(2) = 1$
   $fib(3) = fib(1) + fib(2) = 1 + 1 = 2$
   $fib(4) = fib(3) + fib(2) = 2 + 1 = 3$
   $fib(5) = fib(4) + fib(3) = 3 + 2 = 5$
   $fib(6) = fib(5) + fib(4) = 5 + 3 = 8$
   $fib(7) = fib(6) + fib(5) = 8 + 5 = 13$
   $fib(8) = fib(7) + fib(6) = 13 + 8 = 21$
   $fib(9) = fib(8) + fib(7) = 21 + 13 = 34$
   $fib(10) = fib(9) + fib(8) = 34 + 21 = 55$

6. **Problema de la mochila**. También conocido como *Knapsack problem* por su nombre en inglés. Dado un conjunto de objetos $O_1, O_2, ..., O_n$, cada uno con un peso $p_1, p_2, ..., p_n$ y un valor $v_1, v_2, ..., v_n$; y una mochila con capacidad c: los objetos colocados en la mochila deben maximizar el valor sin exceder la capacidad máxima.

   Este problema es un clásico de la programación dinámica y ha sido muy discutido en la literatura del tema. Se puede resolver tanto de forma iterativa como de forma recursiva,

sin embargo, dados los requerimientos mencionados en la sección 10.1, usaremos la versión recursiva.

7. **Distancia entre palabras**. También conocido como *Levenshtein Distance* en honor a Vladimir Levenshtein, el primero en considerar este problema en 1965. Se trata de la distancia que hay entre dos palabras o secuencias de caracteres. En otras palabras, el número mínimo de inserciones, supresiones o sustituciones necesarias para que una palabra se convierta en otra.

   Cabe destacar que existen dos versiones del problema.

   a. Las sustituciones de letras cuestan el doble que las inserciones y las supresiones. Esto es porque interpretan que primero hay que eliminar una letra y después añadir otra.

   b. Todas las operaciones (inserciones, supresiones y sustituciones) cuestan lo mismo.

   Para este marco hemos seleccionado la versión en la que todas las operaciones cuestan lo mismo.

   Por ejemplo, la distancia entre *patata* y *pata* es 2, porque hay que suprimir *ta* en la primera palabra para que sea igual que la segunda. Otro ejemplo: la distancia entre *hola* y *bola* es 1. Hay que sustituir la *h* por la *b*.

8. **Subsecuencia más larga en común entre dos secuencias**. Como su nombre indica, este problema trata de encontrar la subsecuencia más larga que tengan en común dos secuencias dadas. También hay dos versiones:

   a. Se debe calcular tanto la distancia como la propia subsecuencia más larga que haya en común.

   b. Sólo es necesario calcular la distancia de la subsecuencia más larga que tengan en común ambas secuencias.

   En este marco se implementará la segunda versión, pues así se requiere menos memoria para la tabla de memoización.

   En la Figura 34 se muestra un ejemplo.

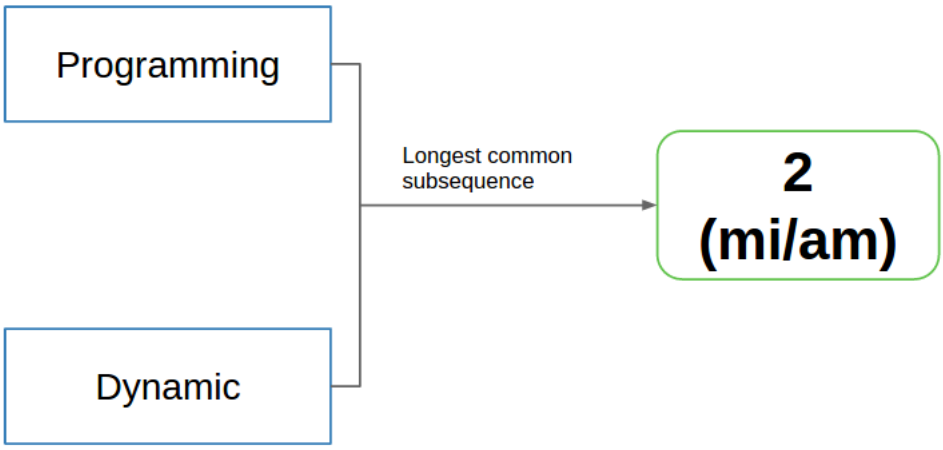


Figura 34: Subsecuencia más larga en común de *Programming* y *Dynamic*

9. **Subsecuencia creciente más larga de una secuencia**. Como su nombre indica, este problema trata de encontrar la subsecuencia creciente más larga que haya en una secuencia

dada. También hay dos versiones:

a. Se debe calcular tanto la distancia como la propia subsecuencia creciente más larga que haya.

b. Sólo es necesario calcular la distancia de la subsecuencia creciente más larga.

En este marco se implementará la segunda versión, pues así se requiere menos memoria para la tabla de memoización.

En la Figura 35 se muestra un ejemplo.

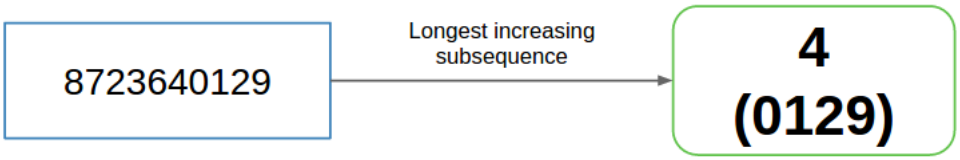


Figura 35: Subsecuencia creciente más larga 8723640129

10. **Multiplicación en cadena de matrices**. Este es un problema de optimización. Dada una secuencia de matrices, el objetivo es encontrar la manera más eficiente de multiplicarlas. Es decir, hay que encontrar el orden de la multiplicación que implique el menor número de operaciones y devolver el número mínimo de operaciones necesarias.

    Puesto que la multiplicación de matrices es asociativa, existen varias formas de multiplicar una cadena de matrices sin alterar el resultado. Sin embargo, el orden de las multiplicaciones afecta al número total de operaciones aritméticas necesarias para calcular el producto. En la Figura 36 se muestra cómo el orden afecta al número de operaciones.

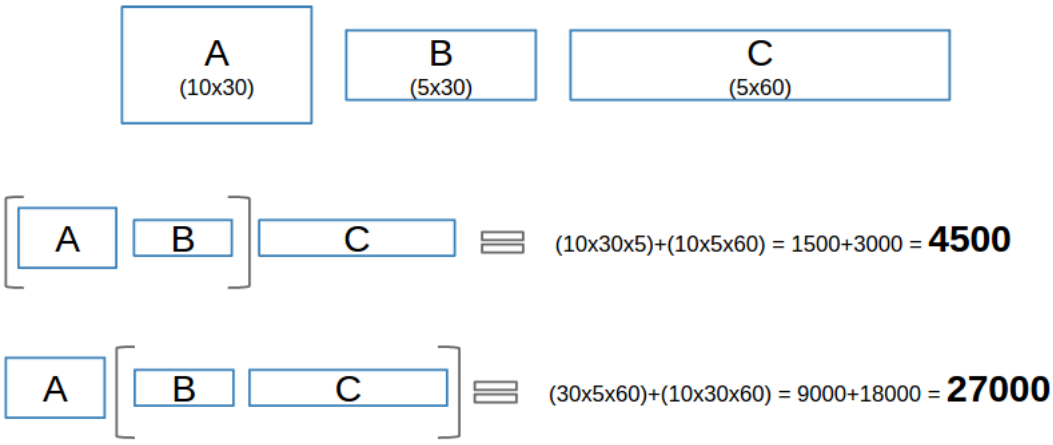


Figura 36: Número de peraciones según el orden de las multiplicaciones en el producto de matrices

11. ***Maximum product cutting***. Dada una cuerda de longitud N, se debe cortar la cuerda en varios trozos de modo que se maximice el producto de todas las partes. Las partes tienen que tener longitud entera. Es obligatorio hacer al menos un corte y la longitud mínima de la cuerda es 2.

    En la Figura 37 se muestra un ejemplo para calcular *Maximum product cutting* para $N = 4$.

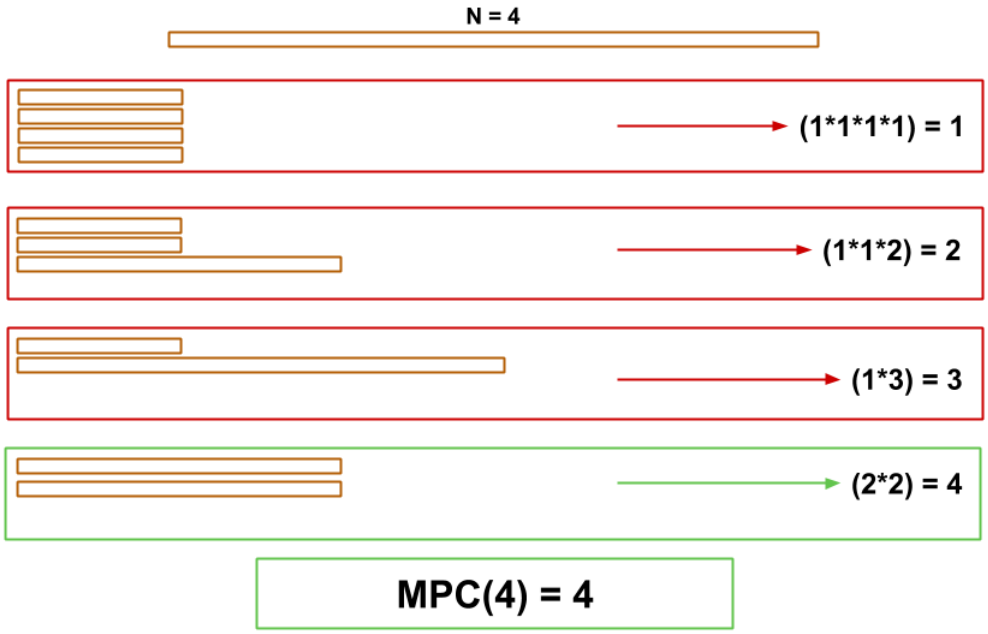


Figura 37: *Maximum product cutting* para $N = 4$

12. **Camino más corto**. Dada una matriz de coste MxN y una posición (x,y), se debe calcular el camino más corto desde la posición (0,0) hasta la posición (x,y). Cada posición de la matriz representa el coste para atravesar esa celda.

    El coste total de un camino es la suma de todos los costes del camino, incluyendo la posición inicial y la final. Sólo se puede pasar a la celda de la derecha, a la celda de abajo o, diagonalmente, abajo a la derecha. Todos los costes deben ser positivos.

    En la Figura 38 se puede observar un ejemplo.

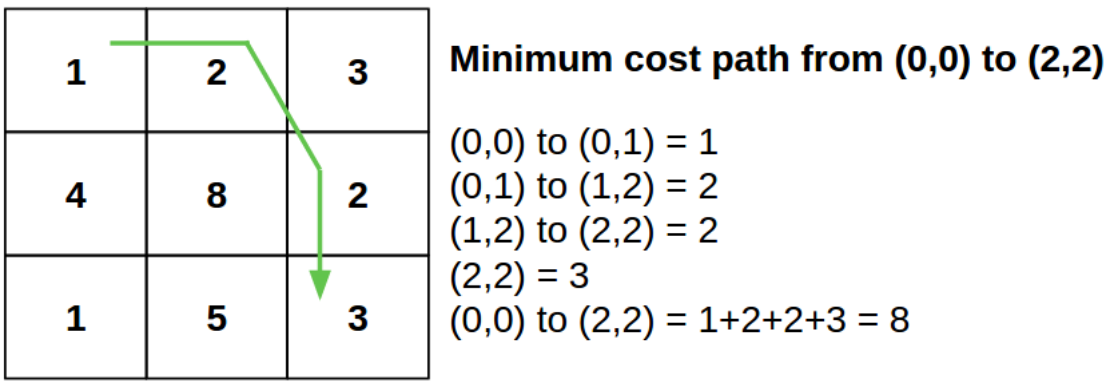


Figura 38: Ejemplo del problema *camino más corto*

10.2.2. Implementación

Una vez los problemas han sido seleccionados es necesario implementarlos. Todos los problemas se implementarán de cuatro formas distintas:

a. **Implementación serie**. Esta versión es una versión sencilla que no contiene ni técnicas de

paralelismo ni de programación dinámica. En otras palabras no contiene ninguna optimización.

b. **Implementación serie con memoización manual**. En esta versión se realiza una memoización manual sobre la implementación anterior. Sin embargo, no contiene ninguna técnica de paralelismo.

c. **Implementación usando OmpSs**. En esta implementación se paraleliza el código serie usando OmpSs. No obstante, no se usa ningún tipo de memoización.

d. **Implementación usando el soporte para programación dinámica de OmpSs**. Esta última versión de los problemas se implementará usando la herramienta desarrollada en este proyecto. Contendrá memoización y paralelismo.

Por otra parte, hay un aspecto digno de mención. Para que la evaluación de rendimiento sea fiable, los problemas deben tardar unos treinta segundos. Es decir, deben tener un tamaño de problema grande. Puesto que los problemas son todos recursivos, al aumentar el tamaño del problema crece mucho el número de tareas que se crean en OmpSs. En este tipo de problemas, las tareas suelen ser muy ligeras, de modo que cuando el número de tareas es demasiado elevado, el *overhead* de creación de las tareas ralentiza mucho la ejecución del problema, porque es más costoso crear la tarea que ejecutarla.

Para corregir el problema se usa la cláusula *final* de OmpSs. Esta cláusula recibe una condición. Cuando esta condición se cumple, se deja de crear tareas y se ejecuta el código de modo secuencial.

10.3. Descripción del entorno de ejecución

La experimentación de este proyecto se ha llevado a cabo en uno de los nodos del supercomputador Marenostrum III. El nodo que se ha usado se compone de:

- 2x E5-2760 SandyBridge-EP 2.6GHz cache 20MB 8-core.
- 1x 500GB 7200 rpm SATA II local HDD.
- 8x 4G DDR3-1600 DIMMs (2GB/core). En total 32GB de memoria.

El sistema operativo de Marenostrum III es Linux - SuSe Distribution 11 SP3.

La versión de Gcc utilizada para compilar ha sido gcc (SUSE Linux) 4.3.4.

La versión de Mercurium utilizada para compilar los códigos OmpSs ha sido mcxx 1.99.4.

La versión de Nanos++ usada ha sido nanox 0.8.

Todas las compilaciones se han realizado sin indicar ningún *flag* de compilación y todas las ejecuciones se han realizado mediante el sistema de colas de Marenostrum III.

10.4. Evaluación de rendimiento

En esta sección se detallan los resultados obtenidos tras ejecutar el marco de evaluación de rendimiento. Se han hecho tres experimentos distintos:

i. *Overhead*. El primer experimento sirve para comprobar si se ha añadido *overhead* a la aplicación. Se ejecutan todas las versiones usando un único hilo de modo que si hay *overhead* se muestra puesto que al no haber paralelismo no se puede compensar.

II. *Strong scaling*. En este experimento se fija un tamaño de problema y se va variando el número de hilos. Este experimento permite encontrar el número óptimo de hilos para un tamaño de problema concreto y muestra también como un aumento desmesurado del número de hilos puede ser contraproducente llegando a reducir el rendimiento.

III. *Weak scaling*. En este experimento se fija un tamaño de problema por hilo. Entonces, a medida que aumenta el tamaño del problema se aumenta el número de hilos de forma directamente proporcional. El comportamiento esperado es que la aplicación siempre tenga más o menos el mismo tiempo de ejecución ya que cada vez que se aumenta el tamaño de la entrada, también se aumentan el número de recursos de forma directamente proporcional. Si no es así, es probable que haya mucha comunicación entre los recursos y por tanto, al aumentarlos, se aumenta el *overhead*.

10.4.1. *Overhead*

Para este experimento se han seleccionado 5 tamaños de problema para cada uno de los problemas que forman parte del marco de evaluación de rendimiento. Se han lanzado 5 ejecuciones para cada tamaño de problema para calcular la media de los tiempos de ejecución. Esto es para evitar *outliers* que puedan desvirtuar la experimentación. Se han ejecutado las 4 versiones detalladas en la sección 10.2.2 con un único hilo.

En la Figura 39 se muestran los resultados obtenidos para el cálculo de la sucesión de Fibonacci.

En primer lugar, el *overhead* queda escondido por la gran mejora que supone la memoización. Otro detalle interesante es el crecimiento del *speedup* a medida que aumenta el tamaño de problema. Esto es debido a que la complejidad del problema para la versión secuencial es mayor que las de las versiones memoizadas, ya sea manualmente o con OmpSs. Particularmente, en este caso la complejidad de la versión secuencial es exponencial y la de las versiones memoizadas es lineal o muy cercana a lineal. En consecuencia, a medida que crece la entrada, el tiempo de ejecución de la versión secuencial crece mucho más rápido que el tiempo de ejecución de las otras versiones y el *speedup* crece cada vez más.

Cabe destacar el elevado rendimiento que consigue la memoización en este experimento, alcanzando casi 30.000x en la versión OmpSs memo y más de 30.000.000x con memoización manual. Esto está causado por la estructura del problema. En este caso, el reuso de las tareas es muy elevado como se puede observar en la Figura 30, y eso acelera mucho la ejecución.

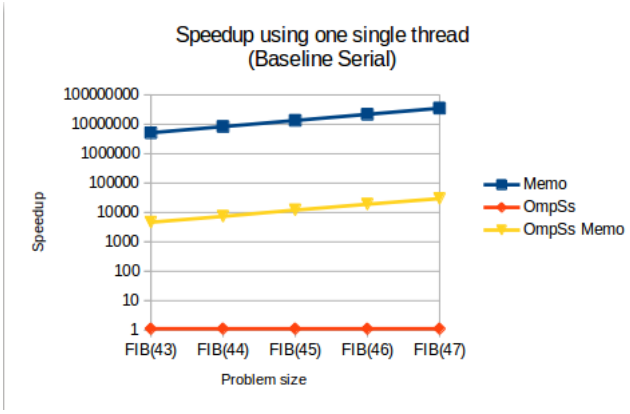


Figura 39: Resultados del experimento *Overhead* para el cálculo de la sucesión de Fibonacci

En contraste con el gran rendimiento que se consigue en el cálculo de la sucesión de Fibonacci

está el problema de la mochila. Los resultados para el problema de la mochila se muestran en la Figura 40. En este caso la estructura del problema es muy distinta. Dependiendo de la entrada

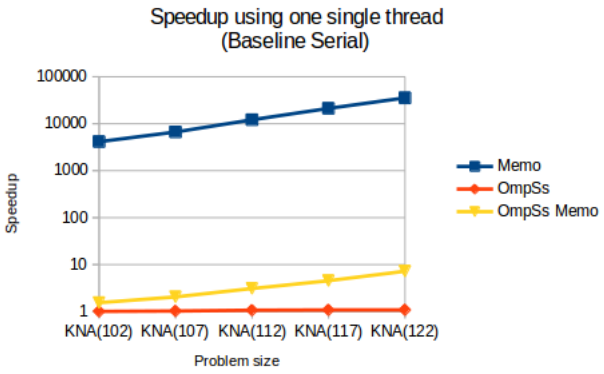


Figura 40: Resultados del experimento *Overhead* para el problema de la mochila

del problema el reuso de tareas puede ser mayor o menor, pero en cualquier caso el nivel de reuso es menor que el del cálculo de la sucesión de Fibonacci y esto se refleja en el rendimiento. En este caso, la versión OmpSs memo alcanza sólo poco más de 7x de *speedup*, mientras que la versión con memoización manual llega únicamente a algo más de 35.000x.

Sin embargo muestra similitudes con el cálculo de la sucesión de Fibonacci, como la tendencia a crecer el *speedup* al aumentar el tamaño del problema, por la misma razón que se explicaba con anterioridad.

El cálculo de la secuencia de Fibonacci es el mejor caso para este experimento, mientras que el problema de la mochila es el peor caso. El resto de problemas presentan comportamientos muy parecidos o al cálculo de la secuencia de Fibonacci o al problema de la mochila y sus rendimientos se sitúan entre los de estos dos problemas explicados, de modo que no se comentarán. Sin embargo, se añaden en el Anexo A.

10.4.2. *Strong scaling*

Para este experimento se ha seleccionado 1 único tamaño de problema para cada uno de los problemas que forman parte del marco de evaluación de rendimiento. Se han ejecutado las 4 versiones detalladas en la sección 10.2.2 variando el número de hilos entre 1 y 16, multiplicando por 2 en cada aumento, es decir: $1, 2, 4, 8, 16$. Se han lanzado 5 ejecuciones para cada número distinto de hilos para calcular la media de los tiempos de ejecución. Esto es para evitar *outliers* que puedan desvirtuar la experimentación.

Los resultados obtenidos para el cálculo de la sucesión de Fibonacci en este experimento se muestran en la Figura 41. En este caso se muestran dos gráficos distintos cambiando la escala, para poder ver algunos valores con más precisión.

De nuevo, para este problema observamos un gran rendimiento, que se mantiene igual para la memoización manual, puesto que no se puede aumentar el número de hilos, pero que crece hasta casi 35.000x en la versión OmpSs memo con 2 hilos.

Para este experimento también es importante la estructura del problema. En el caso del cálculo de la sucesión de Fibonacci, viendo la estructura que presenta este problema en la Figura 30 se justifica el escaso paralelismo que se puede abrir. En consecuencia, aumentar el número de hilos más allá de 2 no hace más que crear *overhead* en la ejecución. Por esta razón, en la Figura 41, en

el gráfico de la derecha, se puede ver, bien diferenciado, que el rendimiento más alto se consigue con 2 hilos y a partir de ahí el rendimiento cae en picado.

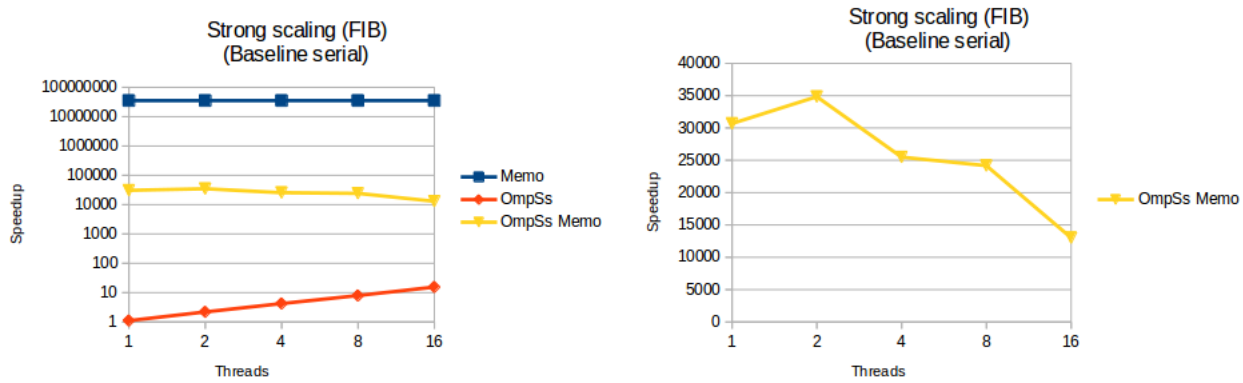


Figura 41: Resultados del experimento *Strong scaling* para el cálculo de la sucesión de Fibonacci

Por otra parte, el problema de la mochila contrasta con el gran rendimiento que se consigue en el cálculo de la sucesión de Fibonacci. Esto se puede observar en los resultados que se muestran en la Figura 42. Nuevamente, el rendimiento obtenido para este problema queda muy por debajo

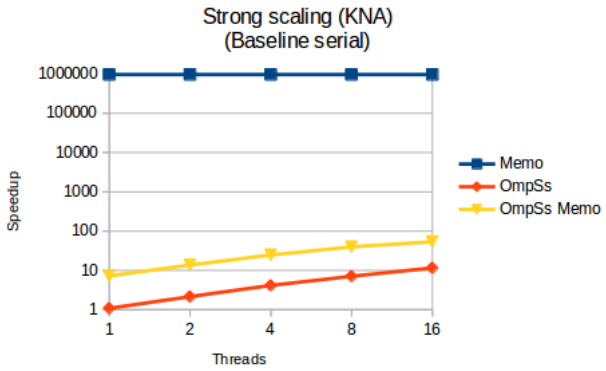


Figura 42: Resultados del experimento *Strong scaling* para el problema de la mochila

del rendimiento obtenido en el cálculo de la sucesión de Fibonacci. Al igual que sucedía en el experimento anterior, esto es debido a la estructura del problema que provoca poco reuso de tareas. Sin embargo, este experimento presenta un rendimiento mayor puesto que se aprovecha el paralelismo. La estructura de este problema, la misma que provoca un reuso reducido, es muy propicia para el paralelismo y esto se puede comprobar en la Figura 42 donde se muestra el crecimiento del *speedup* a medida que se aumenta el número de hilos. De hecho, en este experimento conseguimos pasar del poco más de 7x que nos ofrece un sólo hilo a más de un 50x con 16 hilos.

Al igual que en el experimento anterior, el cálculo de la secuencia de Fibonacci es el mejor caso para este experimento, mientras que el problema de la mochila es el peor caso. El resto de problemas presentan comportamientos muy parecidos o al cálculo de la secuencia de Fibonacci o al problema de la mochila y sus rendimientos se sitúan entre los de estos dos problemas explicados, de modo que no se comentarán. Sin embargo, se añaden en el Anexo A.

10.4.3. *Weak scaling*

Para este experimento se ha fijado un tamaño de problema por hilo para cada uno de los problemas que forman parte del marco de evaluación de rendimiento, excepto dos. Estos dos problemas restantes no han participado en este experimento puesto que no se podía ajustar el tamaño de la entrada correctamente. Se han lanzado 5 ejecuciones para cada tamaño de problema para calcular la media de los tiempos de ejecución. Esto es para evitar *outliers* que puedan desvirtuar la experimentación. Se han ejecutado las 4 versiones detalladas en la sección 10.2.2 variando el número de hilos entre 1 y 16, multiplicando por 2 en cada aumento, es decir: $1, 2, 4, 8, 16$.

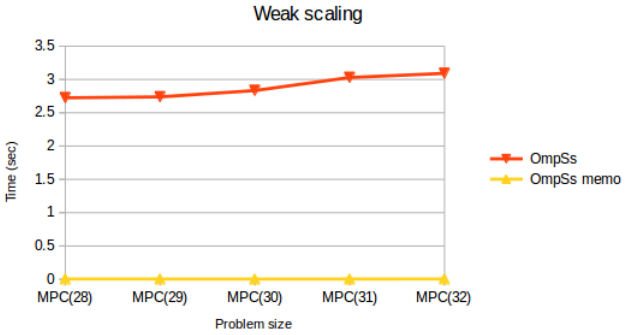


Figura 43: Resultados del experimento *Weak scaling* para el problema *Max product cutting*

Es posible observar los resultados obtenidos para el problema *Max product cutting* en la Figura 43.

En este caso, los gráficos muestran la evolución del tiempo de ejecución dependiendo del número de hilos y del tamaño del problema. Si la aplicación escala correctamente el tiempo de ejecución no debería variar demasiado entre las distintas muestras, y por tanto se debería presentar una línea lo más recta posible. Obviamente, está sujeta a pequeñas variaciones como sucede en la Figura 43.

Otro ejemplo es el mostrado en la Figura 44. Los resultados pertenecen al problema de la mochila. Para este problema, la línea no es todo lo recta que debería, puesto que hay bastante variación entre las distintas muestras. Una vez más, la explicación es la estructura del problema. Como se refería en la sección 10.4.2, la estructura de este problema es muy propicia para el paralelismo y a la vez provoca un reuso muy reducido. Esto, combinado con la naturaleza del experimento, que hace crecer el número de hilos a medida que crece el tamaño del problema, consigue mejor rendimiento con cada aumento del número de hilos, aprovechando el paralelismo.

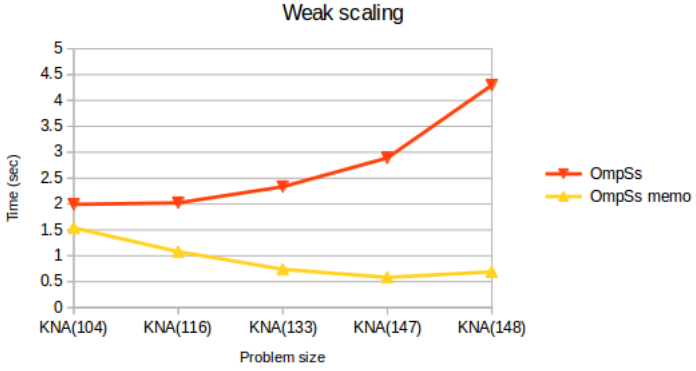


Figura 44: Resultados del experimento *Weak scaling* para el problema de la mochila

No obstante, el último punto del gráfico, el que equivale a 16 hilos no se corresponde a la explicación anterior, ya que está ligeramente por encima del tiempo con 8 hilos. Sin embargo, hay una explicación para esto, que también explica el gran aumento de tiempo que sufre la versión de OmpSs sin memoización. Cuando el número de tareas aumenta demasiado, la carga de trabajo no está bien balanceada y el tiempo de ejecución se ve afectado. Además, también puede estar relacionado con el hecho de que para usar 16 hilos es necesario usar los dos *sockets* del nodo, de modo que tiene que haber comunicación entre los *sockets* y esto puede afectar negativamente al rendimiento de la ejecución.

El resto de problemas presentan comportamientos muy parecidos o al problema *Max product cutting* o al problema de la mochila y sus rendimientos son cercanos a los de estos dos problemas explicados, de modo que no se comentarán. Sin embargo, se añaden en el Anexo A.

Tras presentar los resultados de la experimentación hay dos puntos en común en todos los experimentos.

- La gran mejora de rendimiento que supone una implementación usando el soporte para memoización de OmpSs respecto a una implementación usando OmpSs sin memoización. De hecho, si usamos un tamaño de problema suficientemente grande, habría problemas que tardarían días en ejecutarse usando OmpSs sin memoización mientras que se obtendrían en segundos usando el soporte para memoización de OmpSs. Por ejemplo, en un problema con una entrada pequeña, de unos 30 segundos de ejecución secuencial, la implementación usando el soporte para memoización puede mejorar en tres órdenes de magnitud a una implementación usando OmpSs sin memoización.
- La distancia que existe entre el rendimiento de una memoización manual y el rendimiento del soporte para memoización de OmpSs. Normalmente esta distancia se sitúa entre 2 y 3 órdenes de magnitud a favor de la memoización manual. Es decir, la memoización manual es entre 2 y 3 órdenes de magnitud mejor que la de OmpSs.

Este último punto tiene una explicación. Principalmente, esta distancia es debida al hecho de que cuando OmpSs entra en modo *final* porque la condición de la cláusula *final* se cumple, ya no se realiza la memoización. Es decir, hay una cantidad importante de tareas que no son memoizadas. En contraste, la memoización manual sí se realiza en todas las tareas.

Actualmente, este es un problema difícil de resolver, puesto que el modelo aplica una transformación al código de la función que debe entrar en modo *final* y es muy complicado introducir la memoización en esa transformación.

11. Conclusiones

El soporte para memoización del modelo OmpSs presentado en este proyecto es un esfuerzo para facilitar la ardua tarea del desarrollador para aplicar esta técnica de programación dinámica.

Gracias a la cláusula *memo* añadida al modelo, se consigue que esta técnica se realice de forma automática sólo indicando unos pocos parámetros. Esto reduce la complejidad de esta tarea enormemente.

Asimismo, el peligro de introducir errores en códigos verificados y probados al implementar esta técnica de forma manual desaparece, puesto que usando el soporte para memoización de OmpSs sólo es necesario añadir unas pocas anotaciones en el código, que el compilador es capaz de ignorar si no se quiere usar OmpSs. Esto enlaza con la mantenibilidad y la legibilidad del código. Puesto que las modificaciones son mínimas, es mucho más fácil mantener el código de las aplicaciones. Además, las modificaciones son directivas de compilador que no tienen un efecto sobre la semántica del programa, de modo que la legibilidad tampoco se ve afectada. De hecho, esto implica otra ventaja: se puede memoizar y paralelizar el código de forma incremental. Pueden irse añadiendo las directivas y las cláusulas necesarias poco a poco para ver cómo van afectando al rendimiento de la aplicación sin cambiar la propia aplicación.

Por otra parte, se consigue una combinación de paralelismo y memoización que no se había estudiado todavía, con un resultado bastante satisfactorio, ya que si la estructura del problema es propicia, el paralelismo permite acelerar más aún un código memoizado.

Con todo esto se consiguen resolver los problemas planteados en la sección 3.1.

Por otro lado, el proceso de creación del marco de evaluación de rendimiento ha sido exitoso. El análisis de las propiedades que se hizo para realizar la selección de los problemas fue correcta y, en consecuencia, en todos los problemas seleccionados se ha conseguido una mejora de rendimiento.

Además, el amplio abanico de problemas que forma parte de este marco permite estudiar el comportamiento de la memoización, del paralelismo y de la combinación de ambos de una forma más profunda. Al contar con problemas que presentan distintas estructuras, se ha podido observar cómo afecta la estructura del problema al rendimiento del problema, pero también a la memoización (nivel de reuso de tareas) y al paralelismo. Distintas estructuras obtienen más o menos beneficio de la memoización y más o menos beneficio del paralelismo y gracias a este marco de evaluación se han identificado qué estructuras son más propicias para cada técnica.

Igualmente, este marco de evaluación de rendimiento ha mostrado la gran mejora de rendimiento que supone el soporte para memoización en este tipo de problemas sobre el modelo OmpSs sin memoización. Sin embargo, también ha mostrado que el rendimiento de una memoización manual es mejor que el rendimiento del soporte para memoización de OmpSs.

Esta diferencia entre la memoización manual y la memoización de OmpSs es debida a que en la memoización manual se aplica esta técnica sobre todos los subproblemas, mientras que en la memoización de OmpSs deja de aplicarse cuando se cumple la condición de la cláusula *final*. Este problema se deja para trabajo futuro, puesto que implica cambios en el modelo para poder aplicar la memoización cuando se cumple la condición de la cláusula *final* que escapan del alcance de este TFG.

En conclusión, los problemas planteados en la sección 3.1 han sido resueltos y los objetivos propuestos en la sección 3.3 han sido alcanzados. Se ha conseguido una herramienta de memoización automática, que reduce notablemente el esfuerzo del desarrollador, minimizando los riesgos y

mejorando la legibilidad y la mantenibilidad del código. Por contra, queda trabajo por delante para conseguir colocar el rendimiento del soporte para memoización de OmpSs a la par que una memoización manual, aunque por el momento se entiende como un precio a pagar para contrarrestar el esfuerzo ahorrado y las otras ventajas ofrecidas.

12. Trabajo futuro

Aunque se ha conseguido un gran rendimiento y todos los objetivos, todavía se puede seguir trabajando en esta herramienta. Principalmente, se podría seguir trabajando en mejorar el rendimiento. Esto pasaría por conseguir que las tareas que se ejecuten cuando se cumple la condición de la cláusula *final* puedan realizar también memoización, como sucede cuando se implementa una memoización manual.

Otro aspecto en el que se puede trabajar es en facilitar todavía más al usuario la tarea. Lo que se puede hacer al respecto es cambiar el sistema de indexación de la tabla, de modo que no haga falta que el usuario indique las dimensiones, sólo los índices.

Puesto que OmpSs es un modelo que funciona para C/C++ y Fortran, otro aspecto en el que habría que trabajar es el funcionamiento de la memoización en Fortran.

Por último, para problemas extremadamente grandes que no permitan tener una tabla suficiente para guardar todos los subproblemas, se puede enfocar como una cache, guardando sólo los resultados de aquellos subproblemas más reusados.

13. Revisión del proyecto

Finalmente, el desarrollo del proyecto no ha cumplido con la planificación. Igualmente, el presupuesto también ha sufrido alteraciones. En esta sección se revisa tanto la planificación como el presupuesto inicial para adaptarlos al resultado final.

13.1. Desviaciones en la planificación

La principal desviación del proyecto es debida al siguiente motivo. Al empezar el proyecto, en septiembre, la Facultad de Informática de Barcelona sólo ofrecía, como fechas de defensa, principios de diciembre y después pasaba directamente a abril y junio. Debido a esto, se planificó para defender en diciembre, aunque el tiempo era un poco justo. Sin embargo, sobre noviembre nos dimos cuenta de que había un nuevo plazo de defensa de TFG: finales de enero.

Con más de un mes de tiempo con el que no se contaba, se decidió invertir más tiempo en el desarrollo del soporte para memoización de OmpSs para conseguir el mejor rendimiento posible de la herramienta.

Por otra parte, faltó por considerar una etapa. Tras escribir la memoria final, hace falta que sea revisado por el director y el ponente. Una vez recibidas sus sugerencias, es necesario hacer las correcciones necesarias para que vuelva a ser revisado. Este es un proceso iterativo hasta conseguir la versión definitiva que será la que se ofrezca al tribunal.

Con respecto a la etapa final, que incluía escribir la memoria y preparar la defensa también hay otra desviación. Se planificaron 50 horas para ambas cosas. No obstante, al final se estima que fueron unas 100 horas para esta etapa.

13.2. Planificación revisada

Finalmente, el proyecto ha constado de las siguientes fases:

I. Planificación del proyecto y viabilidad

II. Análisis y diseño del proyecto

III. Desarrollo

IV. Etapa final

V. Revisión

En el Cuadro 9 se muestra la dedicación final para cada tarea.

Etapa	**Dedicación (horas)**
Planificación del proyecto y viabilidad	80
Análisis y diseño del proyecto	100
Desarrollo: Configuración del entorno	50
Desarrollo: Adquisición de conocimiento	80
Desarrollo: Desarrollo	300
Etapa final	110
Revisión	80
Total	**800**

Cuadro 9: Detalle de horas dedicadas a cada tarea del proyecto.

13.3. Diagrama de Gantt revisado

Con el objetivo de exponer el tiempo de dedicación previsto para cada tarea, se incluye el diagrama de Gantt presentado en la Figura 45 con las tareas y los tiempos de cada tarea revisados y adaptados al resultado final.

13.4. Diagrama de Pert revisado

Con el objetivo de exponer la relación entre tareas, se incluye el diagrama de Pert presentado en la Figura 46 con las tareas y las dependencias entre tareas revisadas y adaptadas al resultado final.

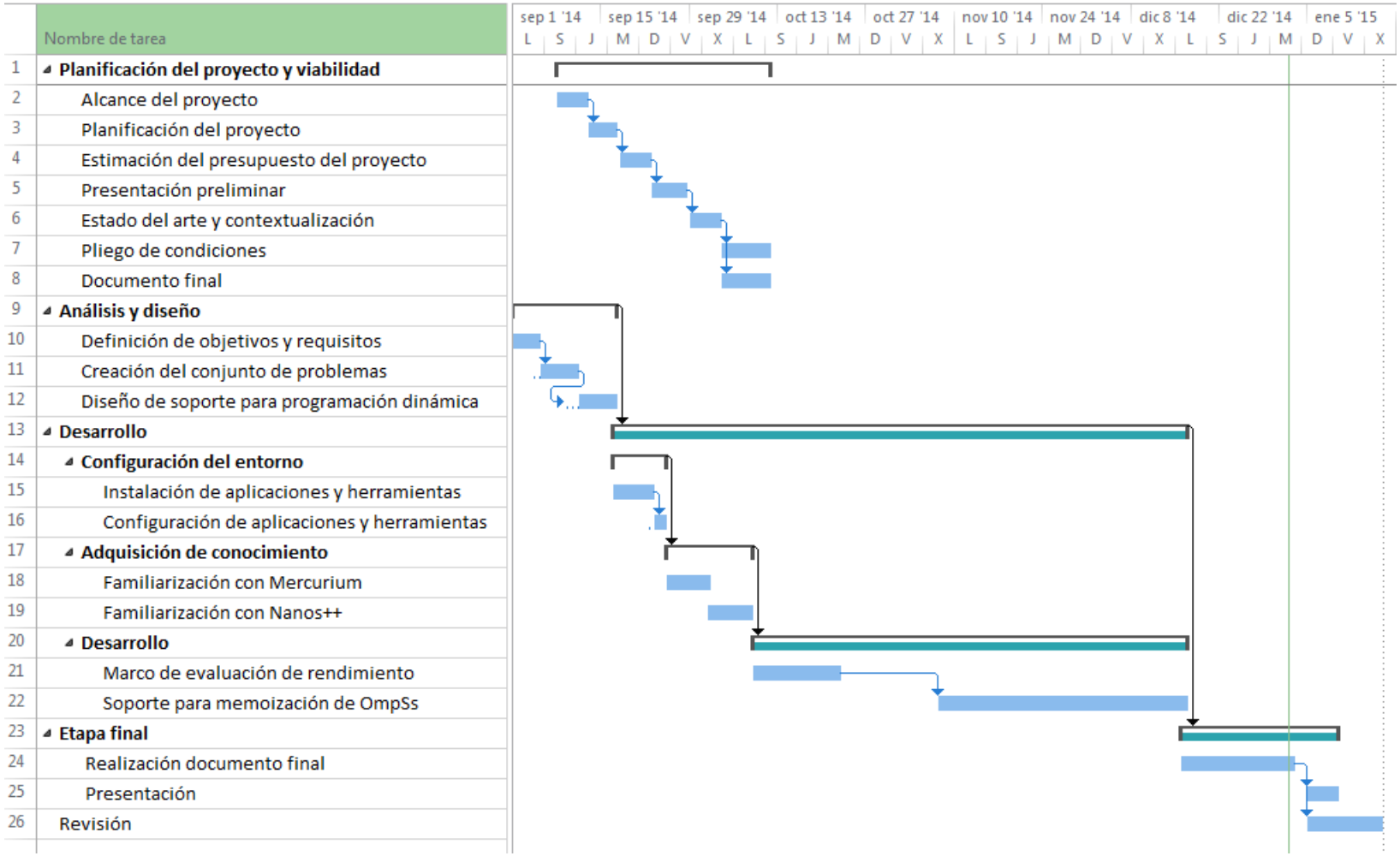


Figura 45: Diagrama de Gantt revisado

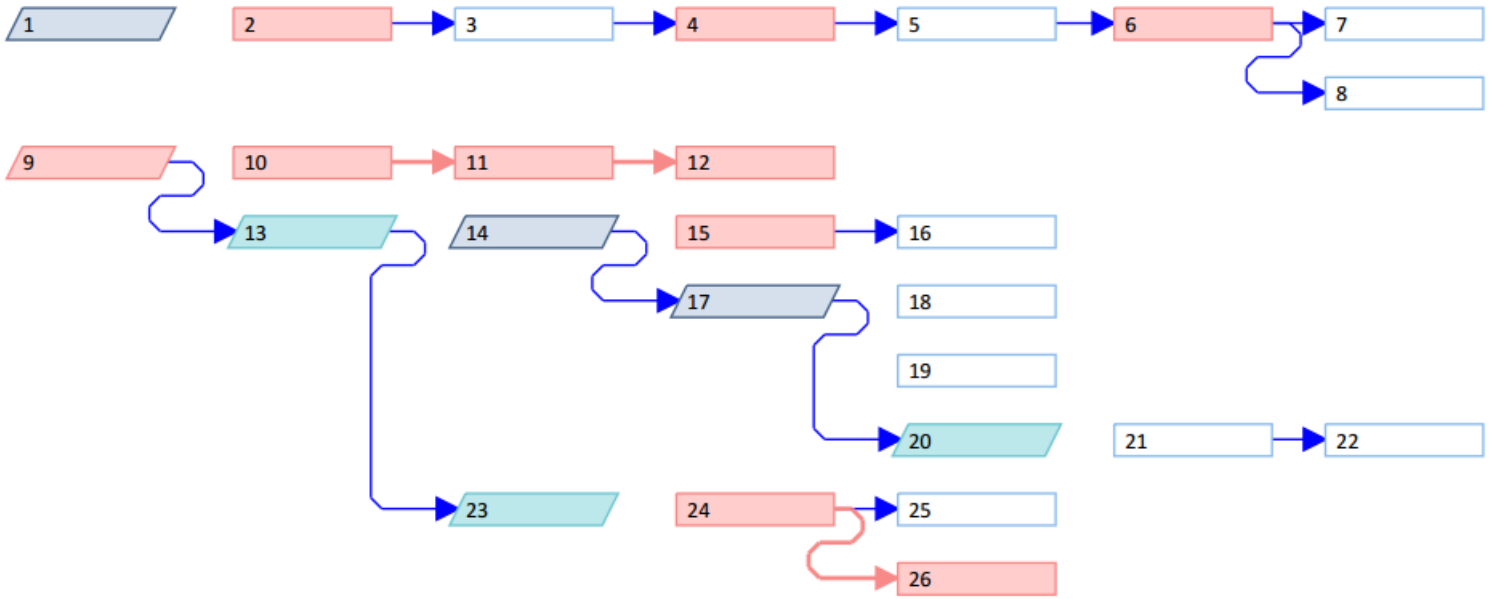


Figura 46: Diagrama de Pert revisado

13.5. Presupuesto revisado

El proyecto comenzó el día 1 de septiembre de 2014 y finaliza la semana del 19 al 23 de enero de 2015. Excluyendo festivos y fines de semana, el proyecto ha durado unos 100 días. La dedicación ha sido de 8 horas al día durante los 100 días de realización del proyecto.

Al igual que en la sección 6, los costes se dividen en:

- Recursos humanos
- Recursos hardware
- Recursos software
- Contingencias
- Imprevistos

13.5.1. Costes de recursos humanos revisados

Se había previsto que el proyecto tuviera unas 13 semanas de vida pero finalmente han sido 20 semanas. Esto ha aumentado la dedicación de todos los roles del proyecto. En el Cuadro 10 se detalla la dedicación revisada de cada uno de los roles que han participado en el proyecto así como el coste que supone.

Rol	**Dedicación (horas)**	**Precio por hora (€/h)**	**Coste total (€)**
Director de proyecto	45	50	2250
Ingeniero de soporte a la investigación	105	30	3150
Becario de investigación	800	10	8000
Total	**950**		**13400**

Cuadro 10: Costes de los recursos humanos.

13.5.2. Costes de recursos hardware revisados

En esta sección no ha habido ninguna alteración respecto a la sección 6.2. Sólo han aumentado las horas de uso y eso se detalla en la sección 13.5.7.

13.5.3. Costes de recursos software revisados

En esta sección no ha habido ninguna alteración respecto a la sección 6.3. Sólo han aumentado las horas de uso y eso se detalla en la sección 13.5.7.

13.5.4. Costes de contingencias revisados

En esta sección no ha habido ninguna alteración respecto a la sección 6.6.

13.5.5. Costes de imprevistos revisados

En esta sección no ha habido ninguna alteración respecto a la sección 6.7.

13.5.6. Costes directos por actividad

En el Cuadro 11 se observa qué recursos materiales se han usado en cada actividad y la duración de cada actividad, de modo que se puedan extraer el número de horas que se ha usado cada recurso con el fin de calcular el coste total. Los recursos estarán identificados por un número tal como se muestra a continuación:

1. Dell Latitude E7440
2. Asus Zenbook UX32A
3. Marenostrum
4. Microsoft Windows 7 Professional
5. Ubuntu 14.04 LTS
6. Microsoft Office 2010/2013
7. Gantter
8. Dropbox
9. Google Drive
10. Github
11. Editor de texto (Geany/GVIM/Gedit)
12. TeX Live
13. Kile
14. Compilador Mercurium
15. *Runtime* Nanos++
16. Librería de instrumentación Extrae
17. Paraver

Actividad	**Recursos**	**Duración (horas)**
Planificación del proyecto y viabilidad	1, 2, 4, 5, 6, 7, 8, 9, 11	80
Análisis y diseño	1, 2, 4, 5, 6, 8, 11	100
Desarrollo: Configuración del entorno	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17	50
Desarrollo: Adquisición de conocimiento	1, 5, 11, 14, 15	80
Desarrollo: Desarrollo	1, 2, 3, 5, 10, 11, 14, 15, 16, 17	300
Etapa final	1, 2, 5, 7, 8, 9, 10, 12, 13	110
Revisión	1, 2, 5, 7, 8, 9, 10, 12, 13	80

Cuadro 11: Utilización de los recursos por actividades.

13.5.7. Coste total

El coste total del proyecto es **14218.2€** y se puede observar, debidamente desglosado, en el Cuadro 12. Para calcular el coste de los recursos materiales se ha multiplicado el coste por hora de cada recurso, por el número de horas que se usa.

Recursos	**Dedicación (horas)**	**Coste por hora (€/h)**	**Coste (€)**
Dell Latitude E7440	800	0.21	168
Asus Zenbook UX32A	720	0.12	86.4
Microsoft Office 2010/2013	230	0.06	13.8
Becario de investigación	800	10	8000
Ingeniero de soporte a la investigación	105	30	3150
Director de proyecto	45	50	2250
Contingencias	-	-	550
Total	-	-	**14218.2**

Cuadro 12: Coste total del proyecto (sólo se muestran los recursos que suponen un coste).

Referencias

[ARB13] OpenMP ARB. Openmp 4.0 specifications released., 2013.

[ARB14] OpenMP ARB. About openmp, 2014.

[ARQ93] R. Andonov, F. Raimbault, and P. Quinton. Dynamic programming parallel implementations for the knapsack problem. *Journal of Parallel and Distributed Computing*, 1993.

[BE05] D. El Baz and M. Elkihel. Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0–1 knapsack problem. *Journal of Parallel and Distributed Computing*, 5(1):74–84, 2005.

[Bel54] R. Bellman. The theory of dynamic programming. In *Bulletin of the American Mathematical Society*, volume 60, pages 503–515, 1954.

[BM13] R. M. Badia and X. Martorell. Tutorial ompss: Single node programming, 2013.

[BSC14a] BSC. Extrae, 2014.

[BSC14b] BSC. Paraver, 2014.

[DAB+11] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173, 2011.

[dIdlUCIdM12] Departamento de Informática de la Universidad Carlos III de Madrid. Openmp, 2012.

[DPV06] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Memoization*, page 173. Algorithms. 2006.

[Dre02] S. Dreyfus. Richard bellman on the birth of dynamic programming, 2002.

[Edd04] S. R. Eddy. What is dynamic programming? 22(7):909–910, 2004.

[GNS07] T. Guangming, S. Ninghui, and Guang R. S. A parallel dynamic programming algorithm on a multi-core architecture. In *SPAA 07 Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 135–144, 2007.

[Lab12] J. Labarta. The ompss programming model., 2012.

[MCU+01] W. S. Martins, J. B. Del Cuvillo, F. J. Useche, K. B. Theobald, and G. R. Gao. A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In *Pacific Symposium on Biocomputing 6*, pages 311–322, 2001.

[MFH95] J. Mayfield, T. Finin, and M. Hall. Using automatic memoization as a software engineering tool in real-world ai systems. *Artificial Intelligence for Applications, 1995. Proceedings., 11th Conference on*, pages 87–93, 1995.

[Mic68] D. Michie. "memo"functions and machine learning. *Nature*, 218:19–22, 1968.

[Mic15] Microsoft. Lvalues y rvalues, 2015.

[Nor91] P. Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, 1991.

[oBa] Programming Models Group of BSC. Mercurium.

[oBb] Programming Models Group of BSC. Nanos++.

[oB13a] Programming Models Group of BSC. Data dependence model, 2013.

[oB13b] Programming Models Group of BSC. Execution model, 2013.

[oB13c] Programming Models Group of BSC. Expressing parallelism, 2013.

[oB13d] Programming Models Group of BSC. Introduction to ompss, 2013.

[oB13e] Programming Models Group of BSC. The ompss philosophy, 2013.

[oB14] Programming Models Group of BSC. The ompss programming model, 2014.

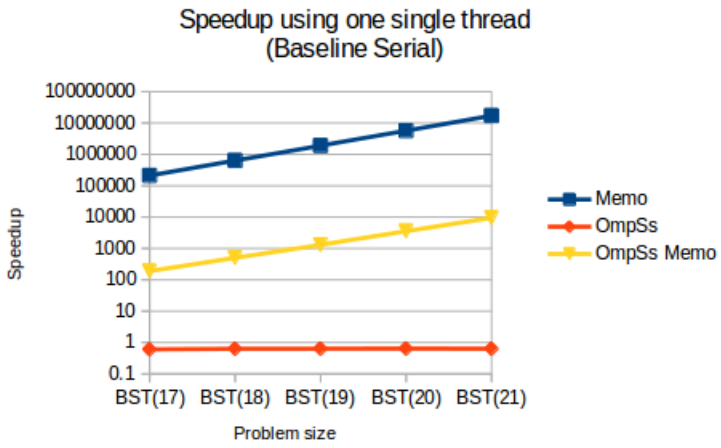
[Pic11] A. Pichler. Dynammic programming, 2011.

[Rol11] T. Rolfe. Exponential base change based on symmetry. *ACM Inroads*, 2(4):33–37, 2011.

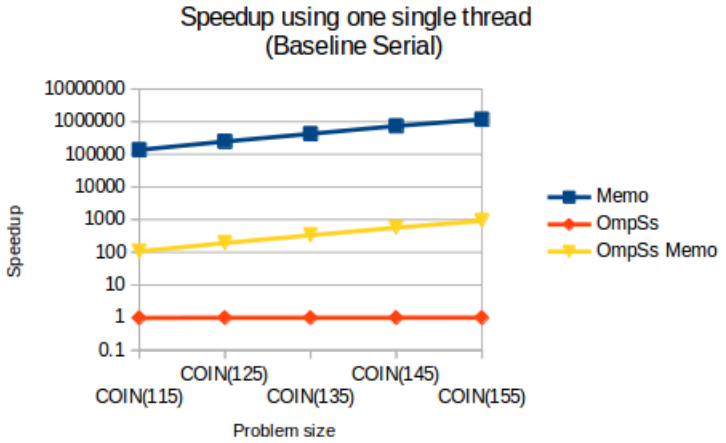
[Sam59] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

A. Resultados de la evaluación de rendimiento

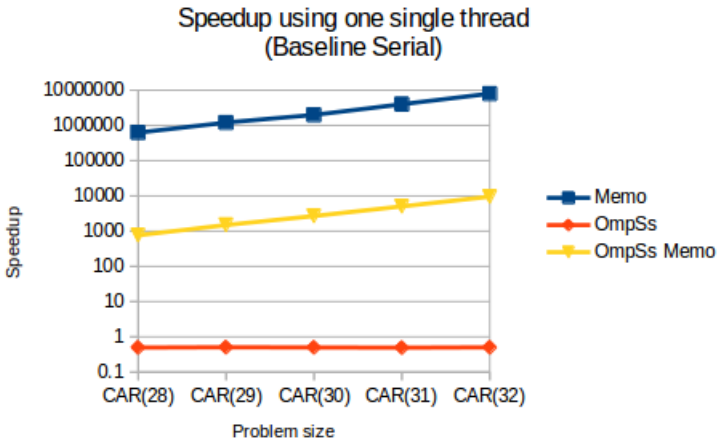
En este capítulo se adjuntan los gráficos resultantes de la evaluación de rendimiento obtenidos para los 12 problemas en cada uno de los tres experimentos.



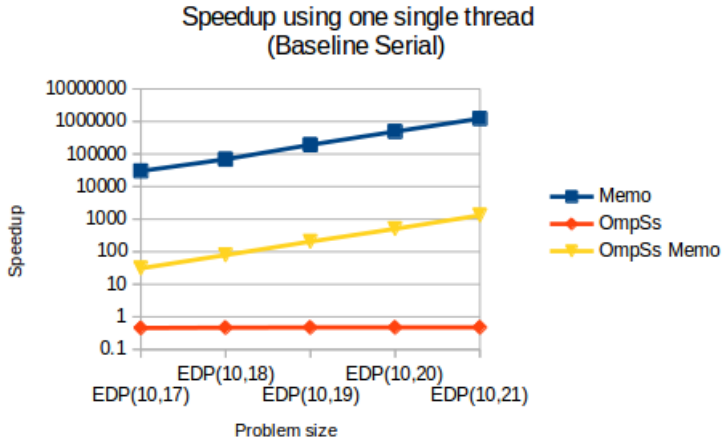
Experimento: Overhead. Problema: Binary Search Tree



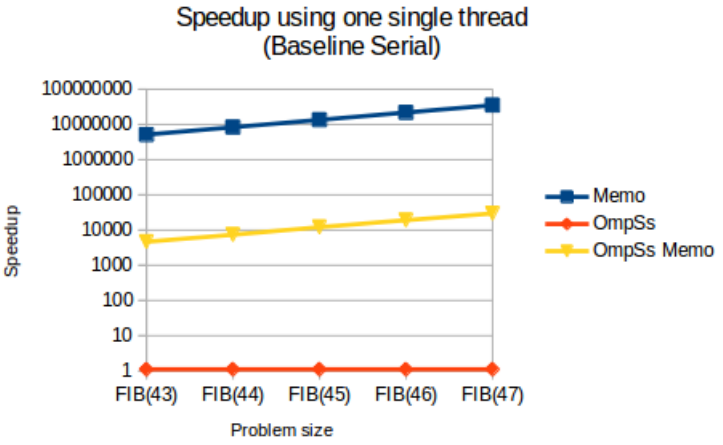
Experimento: Overhead. Problema: Coin change



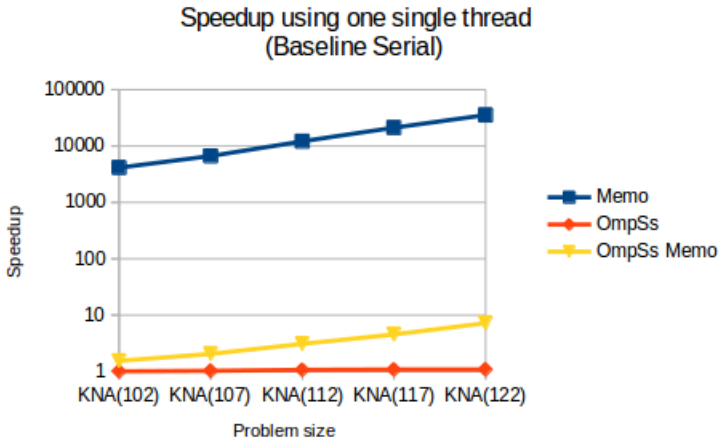
Experimento: Overhead. Problema: Cutting a Rod



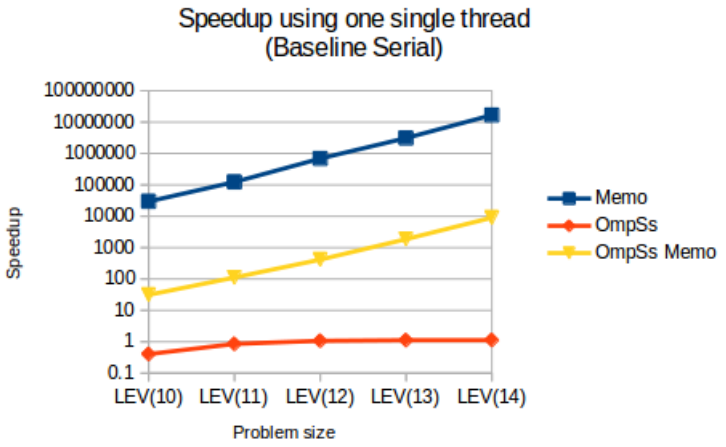
Experimento: Overhead. Problema: Egg Dropping Puzzle



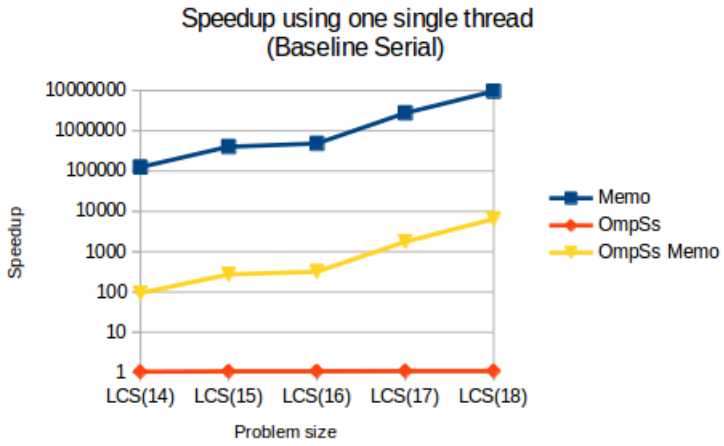
Experimento: Overhead. Problema: Fibonacci



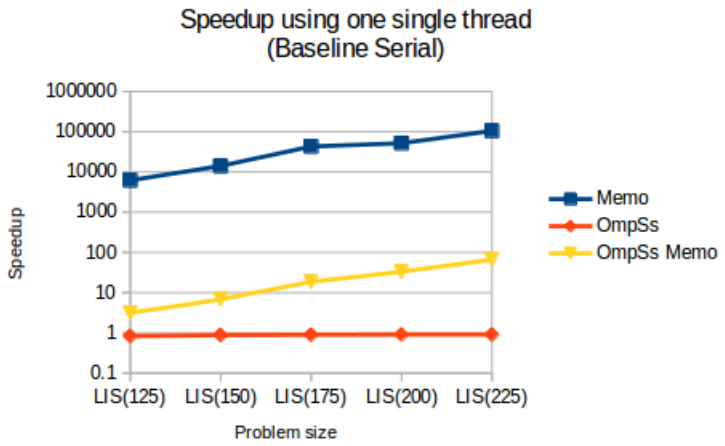
Experimento: Overhead. Problema: Problema de la mochila



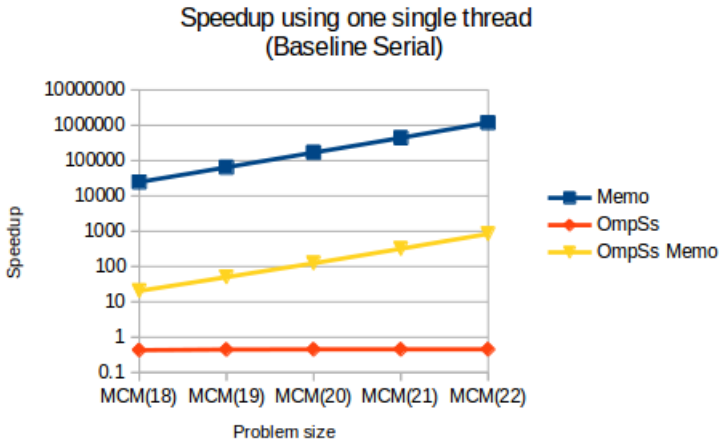
Experimento: Overhead. Problema: Distancia entre palabras



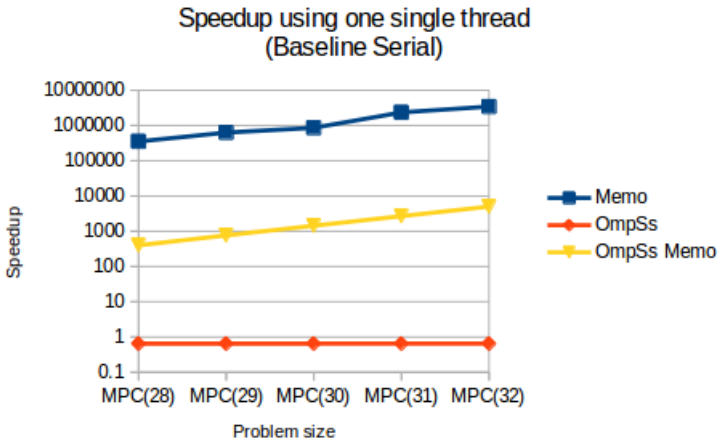
Experimento: Overhead. Problema: Subsecuencia más larga en común entre dos secuencias



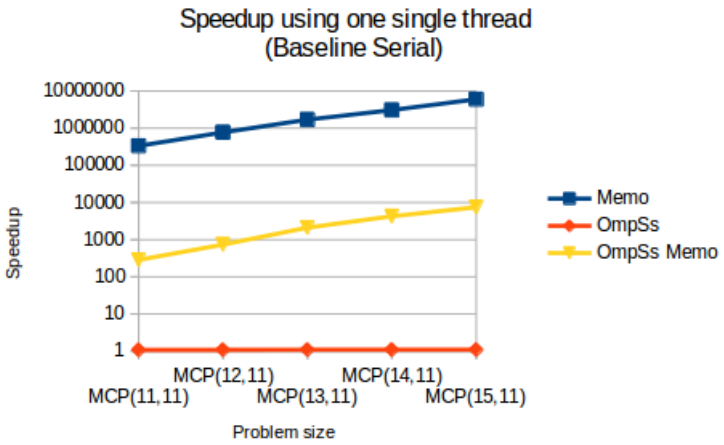
Experimento: Overhead. Problema: Subsecuencia creciente más larga de una secuencia



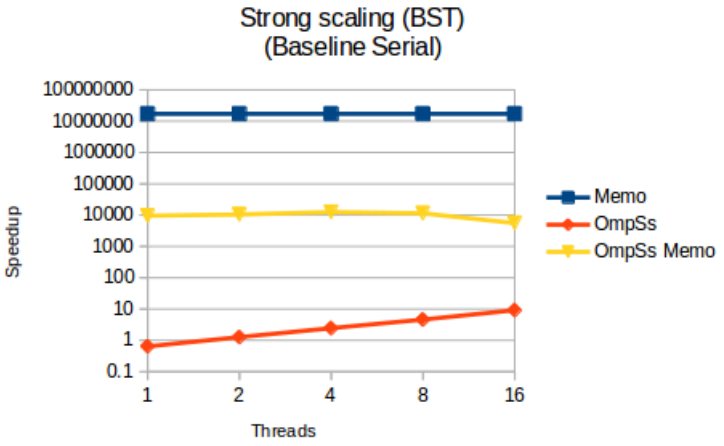
Experimento: Overhead. Problema: Multiplicación en cadena de matrices



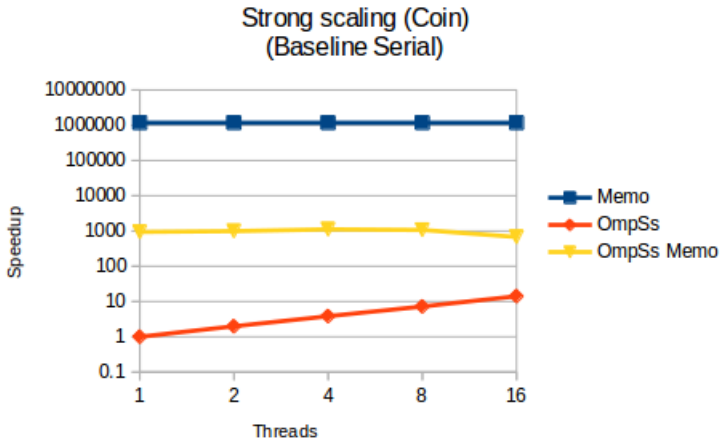
Experimento: Overhead. Problema: Maximum Product Cutting



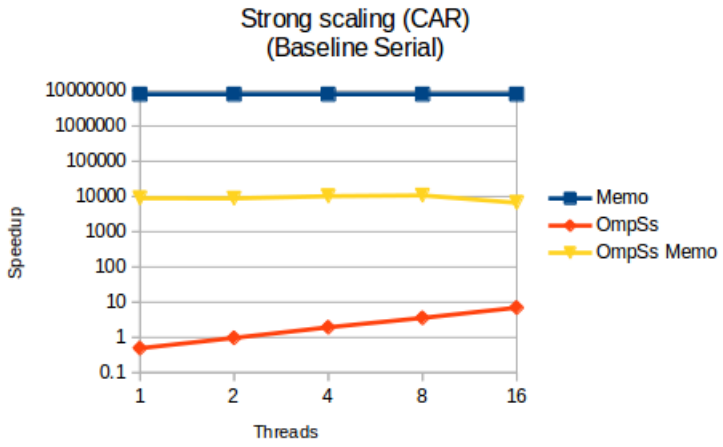
Experimento: Overhead. Problema: Camino más corto



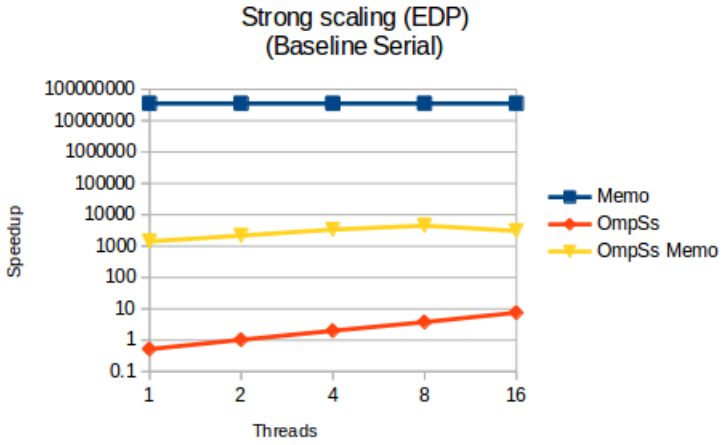
Experimento: Strong scaling. Problema: Binary Search Tree



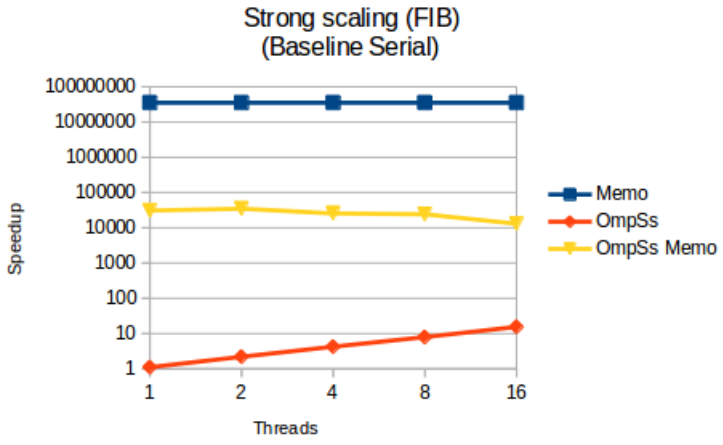
Experimento: Strong scaling. Problema: Coin change



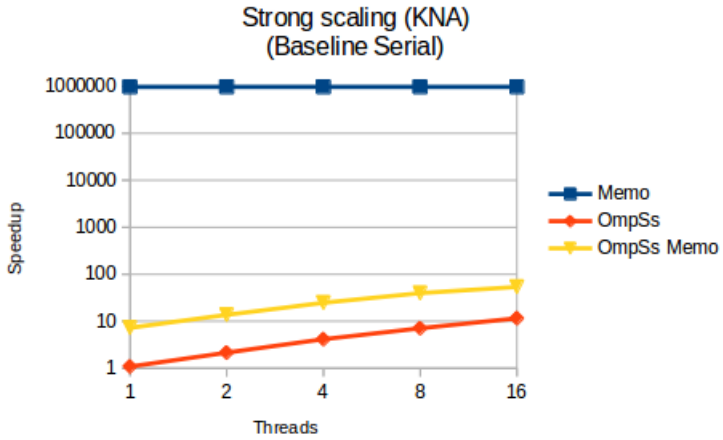
Experimento: Strong scaling. Problema: Cutting a Rod



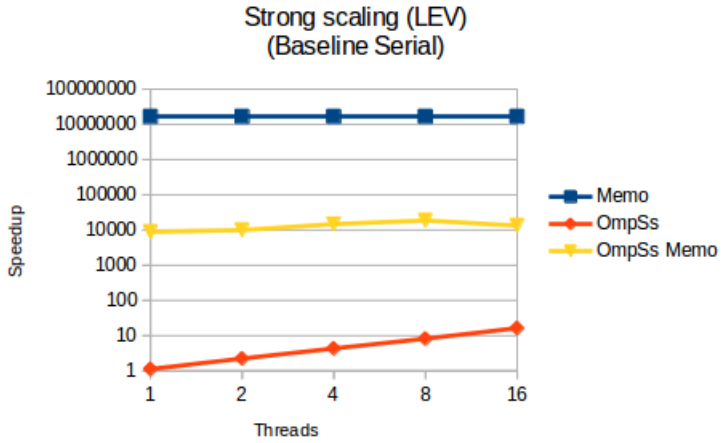
Experimento: Strong scaling. Problema: Egg Dropping Puzzle



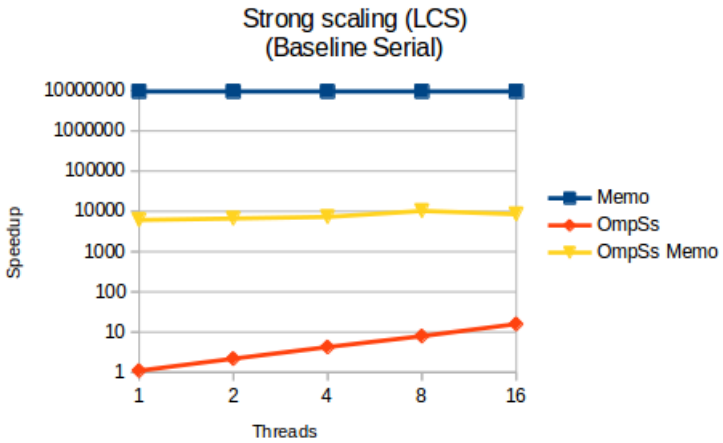
Experimento: Strong scaling. Problema: Fibonacci



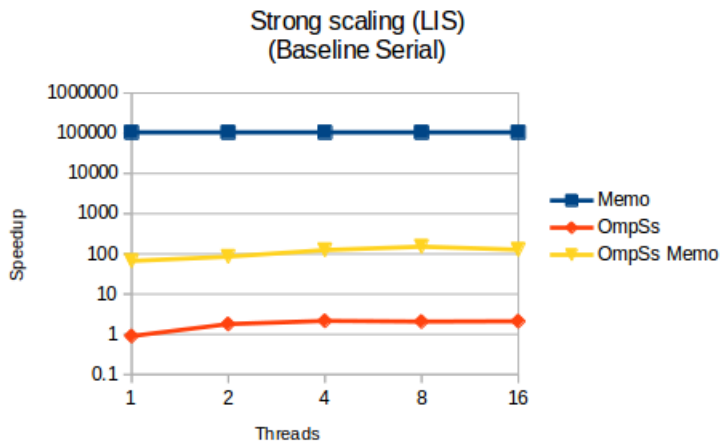
Experimento: Strong scaling. Problema: Problema de la mochila



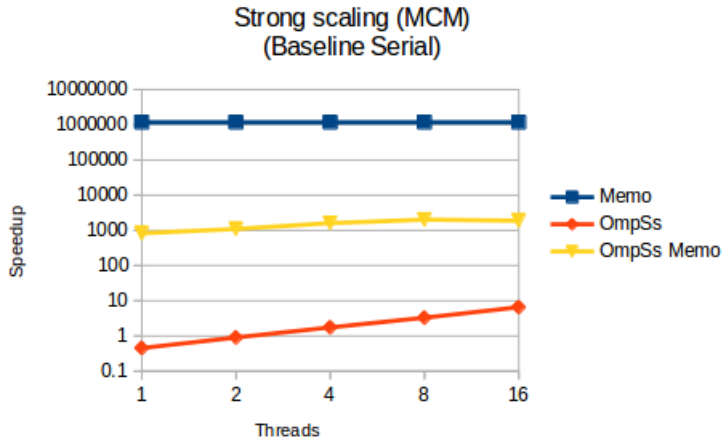
Experimento: Strong scaling. Problema: Distancia entre palabras



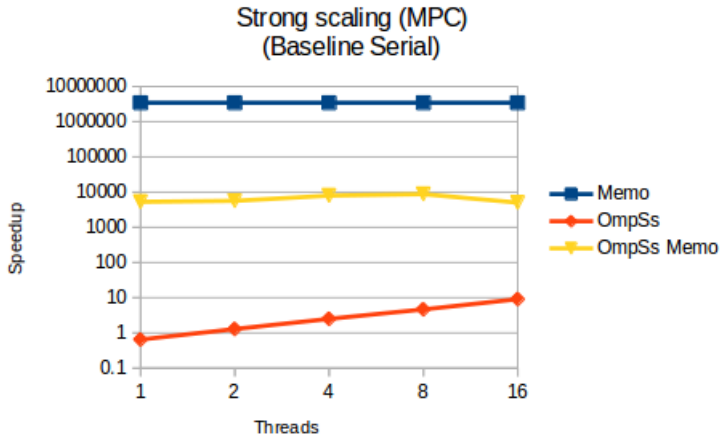
Experimento: Strong scaling. Problema: Subsecuencia más larga en común entre dos secuencias



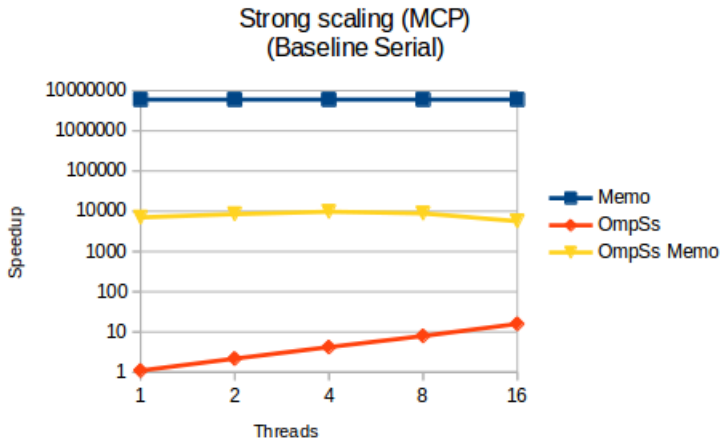
Experimento: Strong scaling. Problema: Subsecuencia creciente más larga de una secuencia



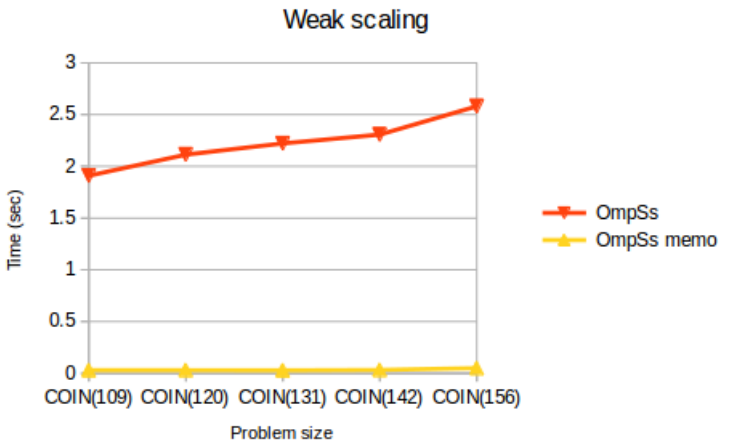
Experimento: Strong scaling. Problema: Multiplicación en cadena de matrices



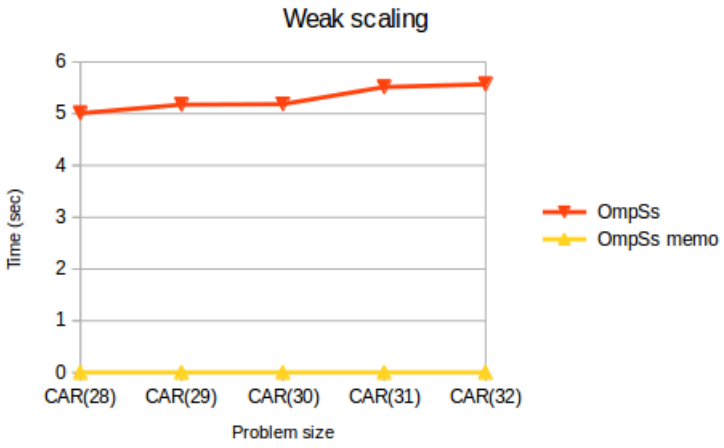
Experimento: Strong scaling. Problema: Maximum Product Cutting



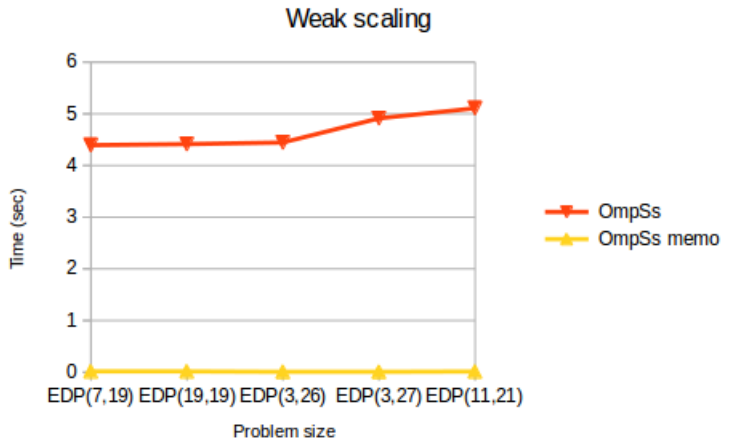
Experimento: Strong scaling. Problema: Camino más corto



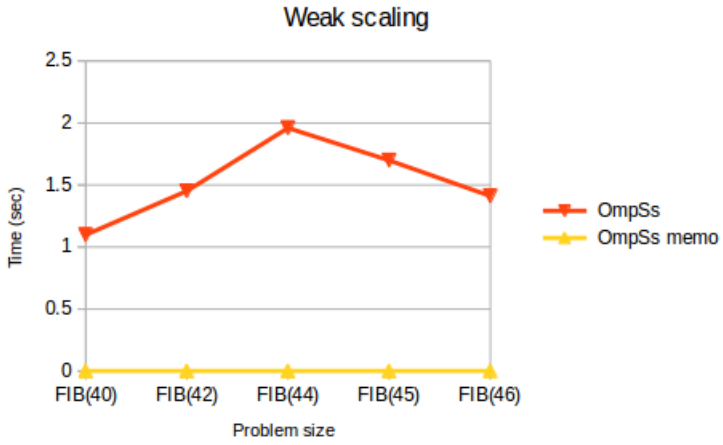
Experimento: Weak scaling. Problema: Coin change



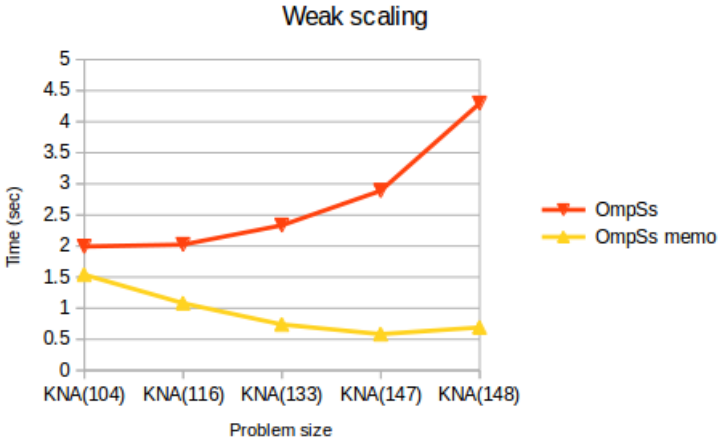
Experimento: Weak scaling. Problema: Cutting a Rod



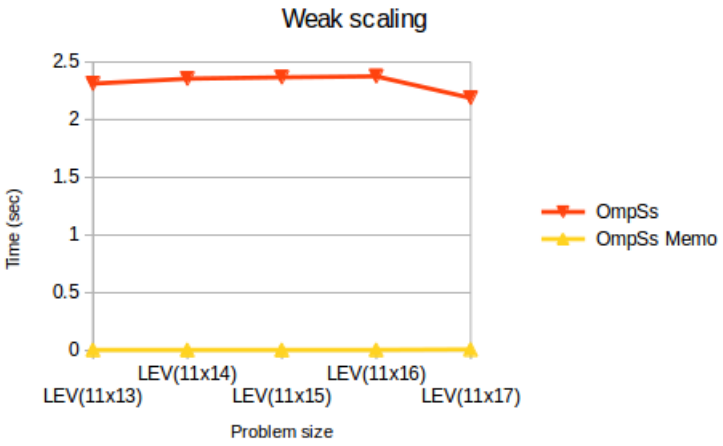
Experimento: Weak scaling. Problema: Egg Dropping Puzzle



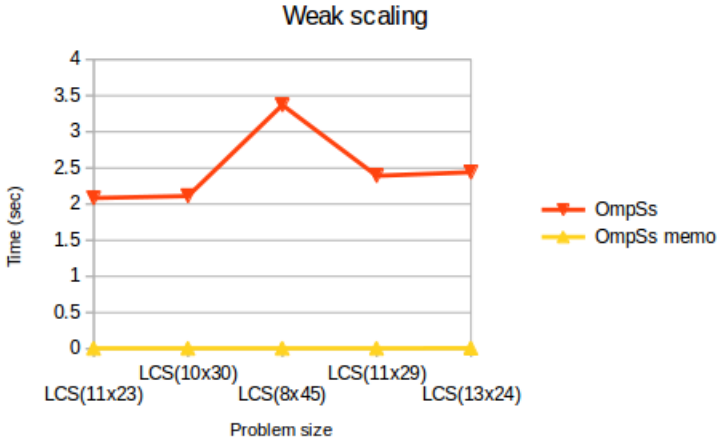
Experimento: Weak scaling. Problema: Fibonacci



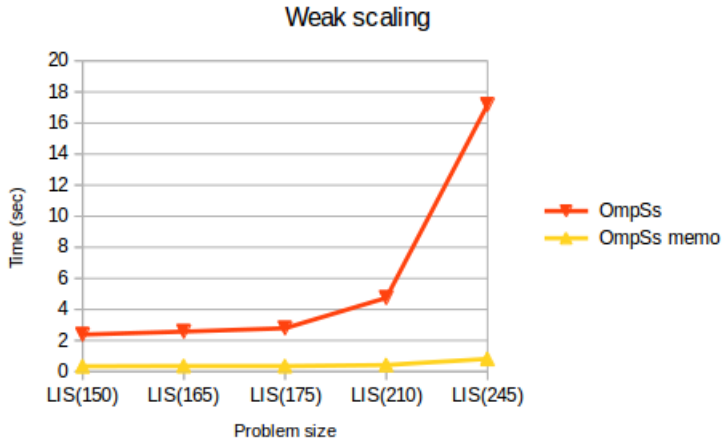
Experimento: Weak scaling. Problema: Problema de la mochila



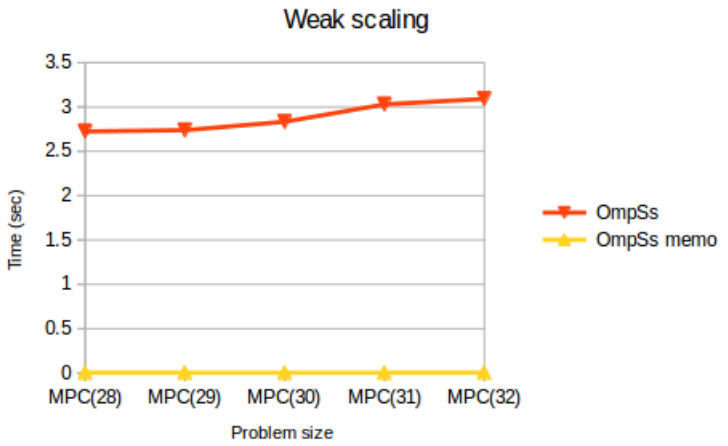
Experimento: Weak scaling. Problema: Distancia entre palabras



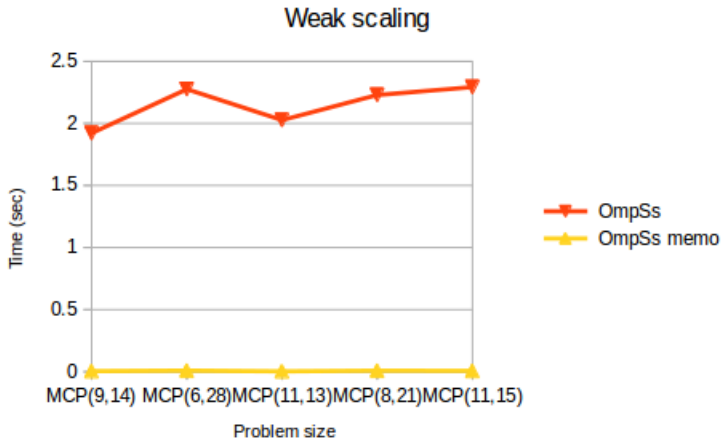
Experimento: Weak scaling. Problema: Subsecuencia más larga en común entre dos secuencias



Experimento: Weak scaling. Problema: Subsecuencia creciente más larga de una secuencia



Experimento: Weak scaling. Problema: Maximum Product Cutting



Experimento: Weak scaling. Problema: Camino más corto

B. Archivos del proyecto

Junto con este documento se incluyen los códigos fuente tanto del soporte para memoización de OmpSs como del marco de evaluación de rendimiento.

En la carpeta *OmpSs_with_memoization_support* se encuentran dos subcarpetas: *mcxx*, que contiene Mercurium y *nanox* que contiene Nanos++. Para compilar e instalar OmpSs sólo es necesario seguir las instrucciones de este enlace:
`http://pm.bsc.es/ompss-docs/user-guide/installation.html`

La carpeta *Performance_assessment_framework* contiene 12 carpetas: una para cada problema del marco. Dentro de cada carpeta correspondiente a un problema se pueden encontrar:

- Las 4 versiones de código detalladas en 10.2.2
- Un *Makefile*
- Los *scripts* usados para lanzar las ejecuciones en Marenostrum.
- Si es necesario, los ficheros de entrada necesarios para ejecutar el problema.
- Si es necesario, un *script*, y/o un pequeño programa en C, para generar los ficheros de entrada necesarios para la ejecución del problema.