UNIVERSIDAD DE SANTIAGO DE COMPOSTELA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA

Paralelización del algoritmo Progressive Hedging para la resolución de problemas estocásticos

Autor:
Cristofer Canosa Domínguez

Directores:
Juan Carlos Pichel Campos
Diego Rodríguez Martínez

Grado en Ingeniería Informática

Julio 2018

Trabajo de Fin de Grado presentado en la Escuela Técnica Superior de Ingeniería de la Universidad de Santiago de Compostela para la obtención del Grado en Ingeniería Informática

D. (Juan Carlos Pichel Campos), Profesor do Departamento de Electrónica e Computación da Universidade de Santiago de Compostela, e **D. (Diego Rodríguez Martínez)**, Profesor do Departamento de Electrónica e Computación da Universidade de Santiago de Compostela,

INFORMAN:

Que la presente memoria, titulada *(Paralelización del algoritmo Progressive Hedging para la resolución de problemas estocásticos)*, presentada por **D. (Cristofer Canosa Domínguez)** para superar los créditos correspondientes al Trabajo de Fin de Grado de la titulación de Grado en Ingeniería Informática, se realizó bajo nuestra dirección en el Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela.

Y para que así conste a los efectos oportunos, expiden el presente informe en Santiago de Compostela a 26 de Julio de 2018:

El director, El codirector, El alumno,

(Juan Carlos Pichel Campos) (Diego Rodríguez Martínez) (Cristofer Canosa Domínguez)

Resumo

La paralelización de un código busca adaptar la ejecución secuencial de forma que se ejecuten varias instrucciones al mismo tiempo. Esto no solo reduce el tiempo de ejecución si no que permite abordar problemas de mayor tamaño y aprovechar clústeres de computación con múltiples nodos. De esta forma el código se puede escalar más fácilmente y asignar un mayor número de recursos a su ejecución.

En este proyecto estudiaremos el módulo de resolución de problemas estocásticos de Pyomo y estudiaremos herramientas que nos permitan adaptar el algoritmo existente a una ejecución paralela.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Este proyecto nace con la intención de trabajar sobre el proyecto de código abierto Pyomo [5], y aportar una implementación paralela de su módulo de resolución de problemas mediante programación estocástica. El objetivo es que la aplicación pueda resolver problemas de mayor tamaño en un tiempo razonable aprovechando el uso de computación distribuida.

1.1. Pyomo y la Programación Estocástica

Pyomo [5] es un paquete de software basado en Python destinado a la formulación y solución de modelos de optimización. Fue desarrollado por *Sandia National Laboratories* y *University of California, Davis* y permite la solución de multitud de problemas distintos, permitiendo su utilización con multitud de solvers de terceros como CPLEX [10] o GLPK [11]. Pyomo es un proyecto extensible mediante la integración de plugins que pueden ser programados por la comunidad para uso privado o público gracias a su filosofía de código abierto.

En este proyecto se ampliará el módulo PySP, el paquete de Pyomo dedicado a la solución de problemas estocásticos. La Programación Estocástica [2] resuelve problemas de optimización donde existe un cierto grado de incertidumbre. Esta incertidumbre hace que no podamos pensar en un problema como algo estático. Los distintos valores posibles generarán escenarios diferentes y, si contamos con varias variables inciertas y dependencias entre ellas, podemos tener un problema que se desarrolle en más de una fase. Por esto, este tipo de problemas suelen representarse como un árbol donde cada nodo es un posible escenario.

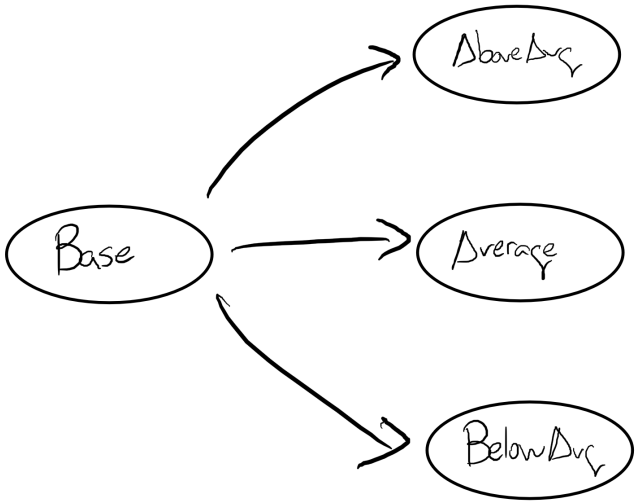


Figura 1.1: Árbol de escenarios en un problema estocástico

Este tipo de problemas es muy común en el mundo real. Por ejemplo, en una tesis defendida en la Universidad de Chile [6], se utiliza la programación estocástica aplicada a la coordinación hidrotérmica. Este problema "busca encontrar la operación óptima para un Sistema Eléctrico Mixto, combinando en la solución los efectos de las etapas futuras así como los efectos que la hidrología tiene en la operación del sistema". Es un problema de optimización para una red energética gubernamental donde se deben tener en cuenta costes y eficiencia de cada tipo de generación de energía, además de contar con factores externos.

Los problemas estocásticos pueden subdividirse en problemas individuales (un problema para cada combinación de posibles escenarios generados) lo que los hace muy buenos candidatos para su paralelización. Estos problemas individuales deben converger a una solución única, lo que se consigue con un algoritmo como el *Progressive Hedging* [3] el cual hace uso de las variables concretas de cada escenario y el valor de control ρ para hacer converger las soluciones a un valor equivalente a la resolución del problema antes de subdividirlo.

1.2. Motivación del proyecto

Como se ha explicado en el apartado anterior, los problemas estocásticos representan situaciones comunes y su solución puede ser aplicable a muchos campos diferentes. Además, Pyomo ya dispone de herramientas para solucionar este tipo

de problemas de forma sencilla, siendo incluso posible hacerlo de forma paralela.

Este proyecto tiene como objetivo realizar una nueva implementación alternativa haciendo uso de herramientas Big Data como Spark [13] o modelos de programación clásicos como MPI [14]. Esta implementación funcionará como base para la resolución de problemas estocásticos en entornos distribuidos, siendo susceptible de recibir optimizaciones concretas para la tecnología escogida y plataformas futuras.

1.3. Estado del arte

Existen múltiples métodos para solucionar problemas estocásticos. Una forma es solucionar la "representación extendida" del problema. En este caso se preprocesa el árbol de escenarios para resolverlo como un problema único.

El método en el que se centrará este proyecto es el algoritmo *Progressive Hedging* (PH) que, como se explicó anteriormente, consiste en la solución de subproblemas y la convergencia de sus soluciones. Podemos ver el pseudo-código del algoritmo concreto en la Figura 3.1.

A continuación se especifican las posibles herramientas para implementar este algoritmo y Pyro, la librería utilizada en la implementación paralela ya existente en Pyomo.

1.3.1. Tecnologías Big Data

El término Big Data se refiere al manejo masivo de datos cada vez más relevante en los últimos años. El software orientado a Big Data está optimizado para la obtención, almacenamiento y procesamiento de grandes cantidades de datos que sería inviable analizar con software tradicional.

Estudiaremos la herramienta Spark [13] de Apache para la implementación del algoritmo. Spark es un framework de código abierto para la programación en entornos distribuidos. Proporciona una capa de abstracción que permite ejecutar trabajos de forma distribuida sin tener conocimiento de las características del cluster. Spark se encarga de la distribución de los datos, el mantenimiento de su consistencia y la optimización de la repartición.

1.3.2. MPI

MPI, siglas para *"Message Passing Interface"*, es un estándar de paso de mensajes que permite la comunicación entre programas a través de la red. Esta interfaz puede utilizarse para implementar una red de objetos distribuidos y solucionar problemas de forma paralela.

MPI permite el paso de mensajes utilizando tipos primitivos o la creación de tipos personalizados. La comunicación puede ser punto a punto entre dos objetos, sincronizando las llamadas a *send* y *receive*, o puede ser colectiva. MPI también provee funciones para facilitar la ejecución de algoritmos distribuidos como MPI_BCAST, MPI_SCATTER o MPI_REDUCE.

Dado que MPI es un estándar, existen librerías para multidud de lenguajes de programación diferentes. Esto permite que programas escritos en lenguajes diferentes puedan colaborar usando una interfaz común. Una utilidad de esta interoperabilidad es poder escribir un nuevo programa que ejecute trabajos en un lenguaje moderno y que se comunique con un programa anterior sin tener que modificarlo.

MPI define una interfaz de bajo nivel para ejecutar trabajos en paralelo. Esto permite una mayor optimización pero debemos ser más cuidadosos con la implementación. Si queremos paralelizar un programa debemos gestionar manualmente la división de la memoria así como sincronizar los mensajes enviados y recibidos.

1.3.3. Pyro

La intención de este proyecto es realizar una implementación paralela del algoritmo *Progressive Hedging*. Para cumplir este objetivo satisfactoriamente, debemos fijarnos en la implementación existente que, en este caso, utiliza la librería Pyro [7].

Pyro es una librería de Python para la implementación de objetos remotos. Funciona de una forma similar a RMI de Java, utilizando un servidor de nombres donde los objetos son registrados. Una vez hecho el lookup de un objeto remoto se puede utilizar como un objeto nativo de Python, pero la ejecución se realizará sobre el objeto remoto.

Este tipo de comunicación entre objetos permite adaptar de forma sencilla un algoritmo implementado con una arquitectura basada en objetos a un entorno paralelo. Sin embargo, este tipo de comunicación tiene ciertas desventajas:

- Se debe gestionar manualmente la repartición de trabajo en un entorno distribuido.

- No es sencillo optimizar la distribución de memoria entre los nodos de trabajo.

1.4. Objetivos

El objetivo principal de este trabajo es paralelizar el algoritmo *Progressive Hedging* del módulo PySP. Este objetivo principal puede subdividirse en los siguientes objetivos específicos:

- Estudiar y analizar el funcionamiento del algoritmo *Progressive Hedging* en PySP.
- Análisis de las diferentes alternativas de paralelización disponibles que mejor se adapten al problema. Se tendrán en cuenta tecnologías Big Data (Apache Spark) o modelos tradicionales de paralelización (MPI).
- Diseño e implementación del nuevo módulo e integración con Pyomo.
- Análisis y evaluación del rendimiento.

1.5. Estructura de la Memoria

En los siguientes apartados de este documento se especificará el desarrollo del proyecto necesario para el cumplimiento de los objetivos anteriores.

- **Capítulo 2:** Especifica aspectos relativos a la gestión del proyecto indicando alcance, requisitos, riesgos, costes y el cronograma del proyecto.
- **Capítulo 3:** Describe el procedimiento de análisis del proyecto y las tecnologías necesarias.
- **Capítulo 4:** Define el diseño del código que se implementará sobre Pyomo.
- **Capítulo 5:** Explica el código desarrollado y los métodos utilizados para su implementación.
- **Capítulo 6:** Determina las pruebas a realizar sobre la implementación para establecer un nivel de confianza concreto sobre el funcionamiento del mismo. Adicionalmente, se establecerán medidas de rendimiento de la nueva implementación.
- **Capítulo 7:** Resume las conclusiones extraidas de la realización del proyecto, lecciones aprendidas y trabajo futuro que ayudaría a mejorar el resultado final.

Capítulo 2

Gestión del proyecto

2.1. Alcance

2.1.1. Descripción del alcance del proyecto

En este proyecto se construirá un módulo software como parte del proyecto Pyomo. Este nuevo módulo adaptará el funcionamiento del actual módulo de programación estocástica (*PySP*) a una implementación paralela.

Se realizará un estudio inicial de Pyomo para decidir la integración y la tecnología a utilizar para la nueva implementación. El objetivo de esta nueva implementación es el de proporcionar una ejecución más escalable que permita abordar problemas de mayor tamaño o, al menos, proporcionar una implementación base que, con futuras optimizaciones, permita conseguir ese objetivo.

El rendimiento de esta nueva implementación estará reflejado en un informe tras hacer pruebas con distintos problemas y en distintos sistemas.

2.1.2. Criterios de aceptación

El proyecto será aceptado cuando se superen las pruebas definidas en el Capítulo 6.

El proyecto persigue dos objetivos: la implementación del algoritmo en paralelo y un estudio de rendimiento. La nueva implementación debe ser capaz de resolver problemas estocásticos de forma paralela usando el algoritmo *Progressive Hedging* y su rendimiento debe escalar añadiendo nodos de computación. Esta implementación debe poder ejecutarse sobre una instalación remota de Spark y utilizando como entrada los mismos modelos de problema que la versión actual de Pyomo.

2.1.3. Entregables del proyecto

Los elementos a entregar tras la finalización del proyecto son:

- Código de Pyomo actualizado con el módulo de ejecución de PH paralelo.
- Estudio de rendimiento (incluído en el documento actual).
- Memoria de realización del proyecto.
- Otra documentación asociada a la realización del proyecto.

2.1.4. Exclusiones

El proyecto producirá una implementación paralela del algoritmo PH existente. Esta implementación debe ser funcional pero no es un objetivo del proyecto hacer que esta nueva implementación sea mejor que la anterior en términos de eficiencia o rapidez.

Pyomo permite el uso de plugins externos que se pueden ejecutar antes o después de calcular la solución y pueden modificar la entrada/salida del mismo. No se harán pruebas de compatibilidad de la nueva solución con estos plugins de terceros en Pyomo. Sólo se verificará su funcionamiento con la ejecución incluida por defecto.

2.1.5. Restricciones

El proyecto debe finalizar el día 18/06/2018. En caso de ser necesario se puede postponer la fecha de finalización hasta el 27/07/2018.

2.2. Casos de uso

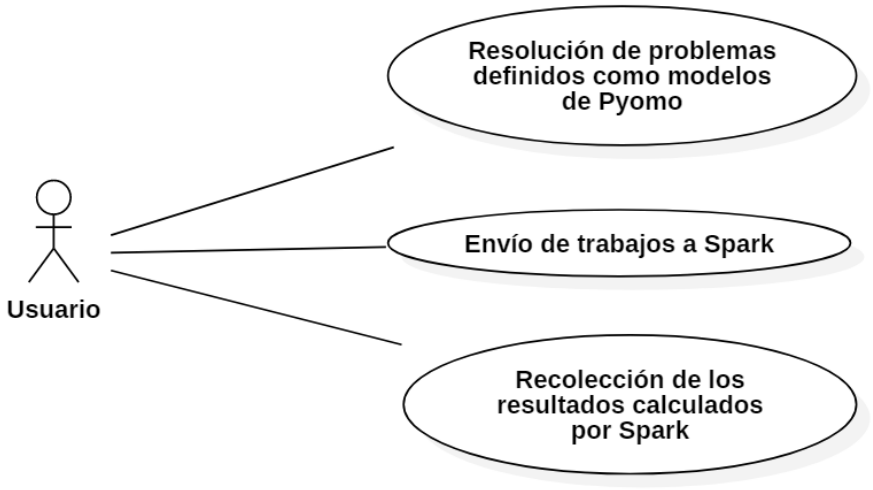


Figura 2.1: Casos de uso

2.3. Análisis de requisitos

El objetivo principal de este trabajo es el de paralelizar el algoritmo *Progressive Hedging* del módulo PySP. Para conseguirlo se establecen una serie de requisitos funcionales y no funcionales.

Cada requisito vendrá definido por:

- **ID:** Identificador único para cada requisito, imprescindible para referenciarlo desde otros puntos del documento como pruebas u objetivos. Tendrá el prefijo "RF" en caso de ser un requisito funcional, y "RNF" para un requisito no funcional.
- **Nombre:** Breve explicación que identifique el objetivo que cumple este requisito.
- **Descripción:** Describe brevemente los criterios de cumplimiento necesarios para aceptar este requisito como completado.
- **Importancia:** Especifica la prioridad de realización de este requisito en función del impacto que tendría su ausencia en el producto final. Podrá ser:
  - *Imprescindible:* No cumplir este requisito puede impedir la finalización satisfactoria del proyecto.

- *Importante:* No cumplir este requisito supondrá una degradación importante de la calidad del resultado.
- *Deseable:* Añadir este requisito supone una mejora de la calidad del resultado pero su falta no tiene consecuencias importantes.
- *Irrelevante:* La existencia de este requisito no tiene un impacto importante (positivo o negativo) en el resultado final.

RF-01	Ejecución de trabajos en Spark
Descripción	Ejecutar el algoritmo PH en Spark de forma paralela.
Importancia	Imprescindible

Cuadro 2.1: RF-01: Ejecución de trabajos en Spark

RF-02	Integración con Pyomo
Descripción	La solución implementada debe funcionar como parte del programa Pyomo.
Importancia	Imprescindible

Cuadro 2.2: RF-02: Integración con Pyomo

RF-03	Interoperabilidad con funcionalidad previa
Descripción	La nueva implementación no debe impedir el correcto funcionamiento de los módulos de Pyomo existentes.
Importancia	Importante

Cuadro 2.3: RF-03: Interoperabilidad con funcionalidad previa

RF-04	Establecer medición de rendimiento de Spark
Descripción	Cuantificar la diferencia de rendimiento y escalabilidad de la nueva implementación.
Importancia	Importante

Cuadro 2.4: RF-04: Establecer medición de rendimiento de Spark

RNF-01	Aprovechar la escalabilidad de Spark
Descripción	La implementación debe mostrar una mejora en el tiempo de ejecución cuando se utiliza un mayor número de recursos en paralelo
Importancia	Importante

Cuadro 2.5: RNF-01: Aprovechar la escalabilidad de Spark

RNF-02	Compatibilidad con entradas y salidas previas
Descripción	Las entradas y salidas del programa deben usar el mismo formato que para la ejecución secuencial.
Importancia	Deseable

Cuadro 2.6: RNF-02: Compatibilidad con entradas y salidas previas

RNF-03	Extensibilidad
Descripción	La implementación diseñada debe permitir la adición de mejoras en un futuro de la forma más sencilla posible.
Importancia	Deseable

Cuadro 2.7: RNF-03: Extensibilidad

2.4. Análisis de riesgos

Para un proyecto del tamaño y duración estimados para este TFG es de vital importancia analizar los posibles riesgos que pueden poner en peligro la satisfactoria finalización del trabajo.

A continuación se especifican los potenciales riesgos junto a su probabilidad, impacto y tratamiento.

La probabilidad se medirá como:

- **Muy alta:** Probabilidad de ocurrencia superior al 90 %.
- **Alta:** Probabilidad de ocurrencia superior al 70 %.
- **Moderada:** Probabilidad de ocurrencia superior al 40 %.
- **Baja:** Probabilidad de ocurrencia inferior al 40 %.

El impacto que tendrá la ocurrencia del riesgo sobre los activos afectados se medirá como:

- **Catastrófico:** Supone la cancelación de tareas, inhabilidad de cumplir la fecha límite o, por último, la cancelación del trabajo.
- **Serio:** Supone modificaciones importantes pero asumibles en la realización de las tareas o el tiempo asignado a las mismas.
- **Tolerable:** La aparición del riesgo causará la necesidad de trabajo extra pero asumible. También puede suponer eliminar tareas o resultados poco importantes.
- **Irrelevante:** Las consecuencias generadas por el riesgo pueden ignorarse sin efectos demasiado negativos.

Riesgo R-01	Mala gestión del tiempo de realización del anteproyecto
Descripción	Un anteproyecto erróneo puede suponer el rechazo del mismo. En menor medida, una descripción demasiado concreta puede suponer un limitante a la hora de realización del trabajo.
Activos afectados	Anteproyecto
Probabilidad	Moderada
Impacto	Catastrófico
Tratamiento	Prevención – Generar una lista de elementos necesarios para el anteproyecto, planificarlos temporalmente y cumplir dicha planificación.
Indicadores	No cumplir la planificación.
Seguimiento	Semanal.

Cuadro 2.8: R-01: Mala gestión del tiempo de realización del anteproyecto

Riesgo R-02	No identificar correctamente los objetivos del trabajo
Descripción	Unos objetivos poco claros pueden llevar a realizar trabajo innecesario o no realizar trabajo imprescindible. También pueden ocasionar desarrollo más lento por no saber qué hacer a continuación.
Activos afectados	Anteproyecto, Planificación temporal, Código
Probabilidad	Moderada
Impacto	Serio
Tratamiento	Minimización – Revisión de los objetivos dispuestos en el anteproyecto con el tutor del TFG. Esta revisión se hará con antelación suficiente para realizar cambios si fuesen necesarios.
Indicadores	Redacción poco clara del propósito del trabajo en la realización del anteproyecto.
Seguimiento	Semanal.

Cuadro 2.9: R-02: No identificar correctamente los objetivos del trabajo

Riesgo R-03	Inhabilidad para ajustar el trabajo a las horas requeridas
Descripción	La realización de un TFG tiene establecida una cantidad de horas fija. Su incumplimiento debe estar justificado.
Activos afectados	Planificación temporal
Probabilidad	Alta
Impacto	Tolerable
Tratamiento	Minimización – Se estimará la duración de las tareas al alza. En caso de que la planificación acabé con más horas de las permitidas, se podrá disminuir la duración de las tareas menos importantes.
Indicadores	La planificación suma más horas de las permitidas.
Seguimiento	Cada vez que se modifique la planificación.

Cuadro 2.10: R-03: Inhabilidad para ajustar el trabajo a las horas requeridas

Riesgo R-04	Memoria final poco concreta
Descripción	La memoria del proyecto debe representar fielmente el desarrollo del mismo. Si se retrasa demasiado su realización es posible perder detalles relevantes del proyecto.
Activos afectados	Memoria
Probabilidad	Alta
Impacto	Serio
Tratamiento	Minimización – Se realizarán, semanalmente o cada 15 días, documentos de progreso que especifiquen las tareas realizadas. Con esto se tendrá una documentación informal pero muy actualizada y concreta.
Indicadores	Parte del desarrollo no está especificado en ningún documento de progreso ni en la memoria final.
Seguimiento	Semanalmente.

Cuadro 2.11: R-04: Memoria final poco concreta

Riesgo R-05	Incumplimiento del cronograma del proyecto
Descripción	Por errores de código, inexperiencia o una carga de trabajo externa mayor de lo esperada, algunas tareas pueden durar más de lo planificado.
Activos afectados	Planificación temporal
Probabilidad	Muy alta
Impacto	Serio
Tratamiento	Prevención – Seguir el porcentaje de cumplimiento de las tareas sobre el cronograma.
Indicadores	Una tarea no se finaliza en plazo o el porcentaje de realización no es suficiente para el tiempo establecido en la tarea.
Seguimiento	Diario.

Cuadro 2.12: R-05: Incumplimiento del cronograma del proyecto

Riesgo R-06	Incompatibilidad de la herramienta escogida con Pyomo
Descripción	Se hará un estudio de herramientas para paralelizar el algoritmo. Si esta herramienta sufre algún tipo de incompatibilidad con el proyecto existente supondrá un retraso importante.
Activos afectados	Diseño, Código
Probabilidad	Baja
Impacto	Catastrófico
Tratamiento	Prevención – Durante la elección de la herramienta se harán pruebas sencillas sobre el proyecto. Tras crear el diseño se hará una implementación de prueba.
Indicadores	Aparecen errores en el programa que no son causados por fallos de programación.
Seguimiento	Semanal durante la fase de diseño. Cada 15 días durante la implementación.

Cuadro 2.13: R-06: Incompatibilidad de la herramienta escogida con Pyomo

Riesgo R-07	Retraso del proyecto
Descripción	Alguna de las tareas planificadas dura más de lo especificado. El impacto dependerá de la gravedad del retraso.
Activos afectados	Todos los ECS
Probabilidad	Moderada
Impacto	Variable
Tratamiento	Minimización – La planificación se creará con un margen de retraso proporcional a la complicación de la tarea.
Indicadores	Una tarea está tardando en finalizar más de lo esperado.
Seguimiento	Semanalmente.

Cuadro 2.14: R-07: Retraso del proyecto

Riesgo R-08	No conseguir acceso a un cluster
Descripción	Para probar el software es ideal utilizar un cluster de computación que permita evaluar la escalabilidad de la solución.
Activos afectados	Memoria, Informe de rendimiento
Probabilidad	Moderada
Impacto	Serio
Tratamiento	Aceptar – La pruebas se realizarán en un ordenador personal y es posible que tengan resultados menos relevantes.
Indicadores	Una tarea está tardando en finalizar más de lo esperado.
Seguimiento	Al finalizar la implementación y al comenzar la ejecución de las pruebas.

Cuadro 2.15: R-08: No conseguir acceso a un cluster

2.5. Análisis de costes

En esta sección se realizará un cálculo de los costes del proyecto basados en la planificación temporal establecida.

2.5.1. Costes materiales

En primer lugar se hará un cálculo de los costes materiales necesarios para la realización del proyecto.

El único material utilizado ha sido el ordenador de desarrollo. Considerando una vida útil de 4 años y un precio base de 800€ el precio de su uso a lo largo de 78 días es:

$$\frac{800}{365 * 4} * 78 = 42€ \tag{2.1}$$

Todo el software utilizado para la realización del proyecto es gratuito a excepción de MSProject, del cual se hizo uso de la licencia de prueba gratuita.

2.5.2. Costes personales

El coste del desarrollador se estima en 16.300€ brutos. Deduciendo IRPF y costes de Seguridad social, se estima el coste por hora en 17€.

Se deben tener en cuenta los gastos que suponen las reuniones con el tutor del TFG. Se calcula un coste por hora de 28€ y un tiempo total de reuniones en

25h.

Si el proyecto dura 78 días y está planificado para un trabajo diario de 5h, los costes personales son:

Desarrollador	$17 * 78 * 5 = 6630$€
Tutor	$28 * 25 = 700$€

2.5.3. Costes indirectos

Estos costes no están directamente relacionados con el desarrollo del proyecto pero proveen bienes necesarios para la realización del mismo. Entre ellos se encuentran servicios básicos como electricidad e internet.

Durante la realización del proyecto se hace uso de estos servicios proporcionados por la universidad mediante el Servicio Universitario de Residencias.

Considerando que el proyecto supone 5h diarias, esto es un 13 % del tiempo de uso de dichos servicios. Por lo tanto, en 5 meses (teniendo en cuenta la realización del anteproyecto): 29,25€

2.5.4. Costes tras la ampliación del proyecto

Una vez modificada la fecha de finalización del proyecto contamos con un mes adicional de gastos. Este nuevo mes debe contar con nuevos gastos indirectos por residir en la vivienda personal, lo que también implica un coste de 10€ por cada reunión que suponga un desplazamiento a Santiago.

Esta nueva fecha de finalización supone 40 días extra de gasto.

En cuanto a costes materiales, se sumarán 40 días al uso del ordenador, sumando 22€ extra.

Los gastos personales, asumiendo otras 15h extra para el tutor:

Desarrollador	$17 * 40 * 5 = 3400$€
Tutor	$28 * 15 = 420$€

Cuadro 2.16: Costes personales del proyecto

En los gastos indirectos, el precio ahora es superior, contando 57€ de conexión a internet a mayores del resto de facturas, manteniendo el 13 % de uso, suma otros

90€ en los meses de Junio y Julio.

Adicionalmente, se consideran 3 reuniones en este periodo adicional, con un coste de 13€ cada una.

2.5.5. Resumen de costes

Personales	10.850€
Materiales	64€
Indirectos	158,25€

Cuadro 2.17: Resumen de costes del proyecto

De esta forma determinamos un **coste total del proyecto** de 11.072,25€

2.6. Gestión de la configuración

Para especificar la Gestión de la Configuración de este proyecto primero debemos identificar los Elementos de Configuración de Software (ECS), es decir, los elementos que se verán afectados por esta Gestión de la Configuración.

Elementos de Configuración:

- Proyecto de software.
- Memoria del proyecto.
- Anteproyecto.
- Documentos de análisis y diseño.
- Informes de progreso.
- Plantillas de documentos.

Para gestionar las diferentes versiones de estos elementos de configuración se creará un repositorio git alojado en Github. Este repositorio contendrá toda la documentación. Para añadir el proyecto de software se realizará un *fork* del proyecto Pyomo original y se añadirá como un submódulo a nuestro repositorio base.

Las modificaciones sobre el software serán individuales por lo que no será necesario considerar técnicas de consistencia en trabajo colaborativo. Para los

cambios que se realicen se crea una rama específica sobre el *fork* del proyecto.

Para relacionar de forma concreta tareas de la planificación con cambios en el repositorio se utiliza un tablero en la plataforma *Trello*. Estas tareas se reflejan en los informes de progreso del repositorio. Además, funciona como herramienta organizativa.

2.7. Planificación temporal

2.7.1. Metodología de desarrollo

Antes de especificar el orden de las tareas y el tiempo asignado a cada una, debemos decidir el ciclo de vida que adoptaremos para la realización de este proyecto.

Este trabajo tiene un tiempo límite fijado que debemos respetar. Los aspectos más importantes serán la investigación inicial y la fase final de optimización y recogida de datos. Inicialmente no se estima que el desarrollo tenga mucho peso en el total del proyecto.

Con el objetivo de controlar estos tiempos y asumiendo que la fase de implementación no necesitará de un ciclo de desarrollo complicado, utilizaremos un ciclo de vida en cascada. Este ciclo de vida nos permite definir la lista de tareas a realizar en orden y ajustándose al tiempo disponible. Como el tiempo de cada tarea está claramente delimitado, será sencillo realizar un seguimiento de la realización de este proyecto, facilitando la detección temprana de retrasos.

2.7.2. EDT

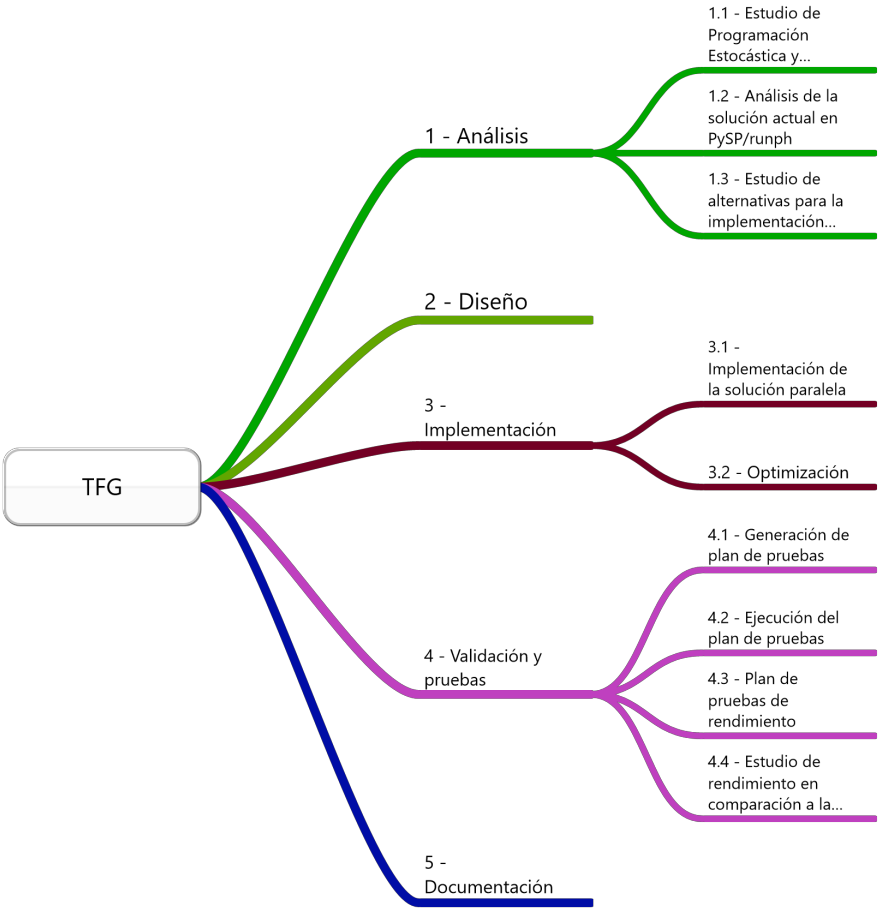


Figura 2.2: EDT inicial

Id	1.1
Tarea	Estudio de Programación Estocástica y *Progressive Hedging*
Duración	7 días
Descripción	Se investigará el funcionamiento del algoritmo *Progressive Hedging* haciendo uso principalmente de [2] como referencia.
Salida	N/A

Cuadro 2.18: Tarea 1.1: Estudio de Programación Estocástica y *Progressive Hedging*

Id	1.2
Tarea	Análisis de la solución actual en PySP/runph
Duración	8 días
Descripción	Una vez conocido el funcionamiento teórico del algoritmo, estudiar su implementación sobre el proyecto Pyomo.
Salida	Diagrama de funcionamiento PH [15].

Cuadro 2.19: Tarea 1.2: Análisis de la solución actual en PySP/runph

Id	1.3
Tarea	Estudio de alternativas para la implementación paralela
Duración	3 días
Descripción	Se barajarán tecnologías Big Data (Spark) o modelos tradicionales (MPI).
Salida	Resultado del análisis en la Sección 3.3

Cuadro 2.20: Tarea 1.3: Estudio de alternativas para la implementación paralela

Id	2
Tarea	Diseño
Duración	20 días
Descripción	Generar un diseño de la implementación a realizar con la tecnología escogida. Será importante la integración con la implementación actual.
Salida	Diseño de la implementación.

Cuadro 2.21: Tarea 2: Diseño

Id	3.1
Tarea	Implementación de la solución paralela
Duración	15 días
Descripción	Escribir los nuevos módulos de código e integrarlos en el proyecto.
Salida	Nuevos ficheros de código y modificaciones a archivos existentes.

Cuadro 2.22: Tarea 3.1: Implementación de la solución paralela

Id	3.2
Tarea	Optimización
Duración	5 días
Descripción	Una vez la integración es correcta y la implementación funciona, se realizarán optimizaciones de rendimiento intentando aprovechar las carácterísticas de la tecnología escogida para la nueva implementación paralela.
Salida	Modificaciones a la implementación anterior.

Cuadro 2.23: Tarea 3.2: Optimización

Id	4.1
Tarea	Generación de plan de pruebas
Duración	4 días
Descripción	Idear un plan de pruebas para el nuevo módulo.
Salida	Documento de pruebas [**?**].

Cuadro 2.24: Tarea 4.1: Generación de plan de pruebas

Id	4.2
Tarea	Ejecución del plan de pruebas
Duración	1 días
Descripción	Implementar y ejecutar las pruebas establecidas para establecer confianza sobre el correcto funcionamiento de la implementación.
Salida	Informe de ejecución de pruebas.

Cuadro 2.25: Tarea 4.2: Ejecución del plan de pruebas

Id	4.3
Tarea	Plan de pruebas de rendimiento
Duración	3 días
Descripción	Idear un plan de pruebas con problemas que permitan estudiar el rendimiento del programa.
Salida	Documento de pruebas de rendimiento [**?**].

Cuadro 2.26: Tarea 4.3: Plan de pruebas de rendimiento

Id	4.4
Tarea	Estudio de rendimiento
Duración	2 días
Descripción	Ejecutar el plan de pruebas anterior y compararlo con las versiones originales tanto secuencial como con Pyro.
Salida	Informe de rendimiento [**?**].

Cuadro 2.27: Tarea 4.4: Estudio de rendimiento

Id	5
Tarea	Documentación
Duración	10 días
Descripción	Generar los documentos asociados al desarrollo del proyecto.
Salida	Memoria del proyecto y documentos asociados.

Cuadro 2.28: Tarea 5: Documentación

Tareas no planificadas

Tras las modificaciones realizadas sobre el cronograma y explicadas en Subsección 2.7.4, se generan nuevas tareas para el proyecto:

Id	3.*
Tarea	Prototipo aislado
Duración	10 días
Descripción	Crear una aplicación en Python que interactúe con Spark de forma similar a como lo hará la implementación en Pyomo.
Salida	Memoria del proyecto y documentos asociados.

Cuadro 2.29: Tarea 3.*: Prototipo aislado

Id	3.*
Tarea	Prototipo de integración inicial
Duración	5 días
Descripción	Comenzar la implementación sobre Pyomo comprobando que la integración del nuevo módulo con el código existente es correcta y el flujo de ejecución es correcto con respecto al funcionamiento anterior.
Salida	Código añadido a Pyomo.

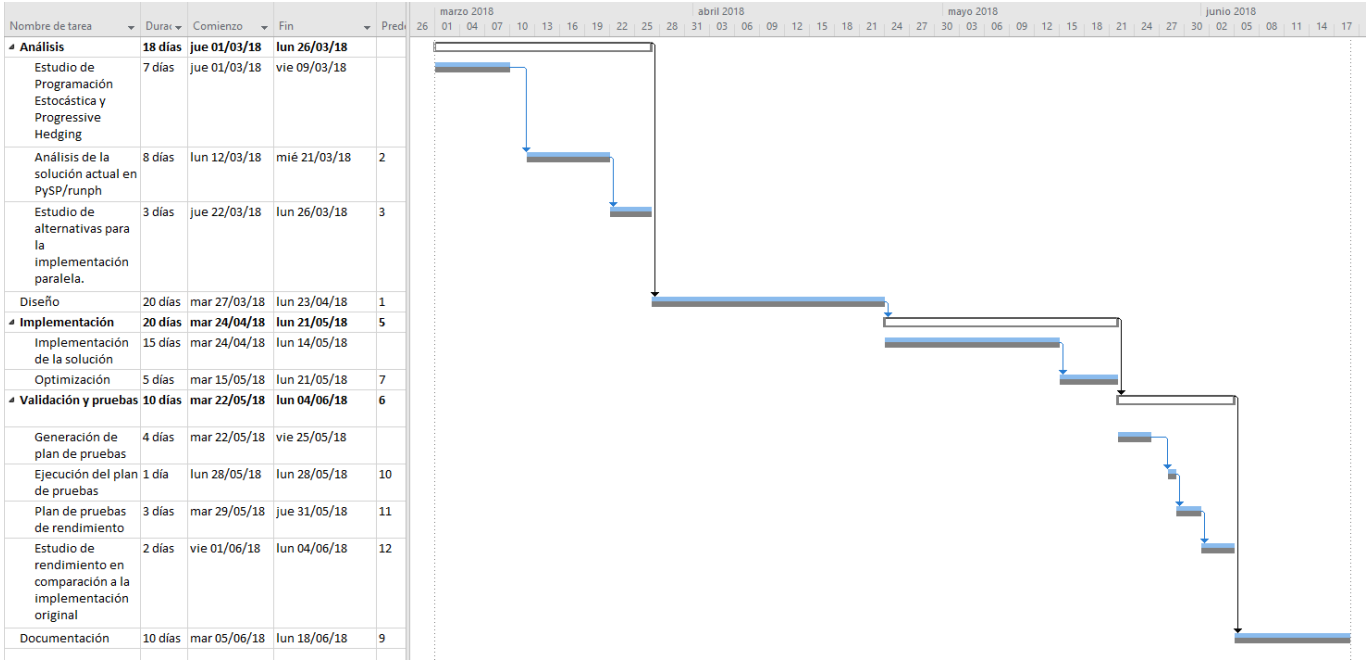
Cuadro 2.30: Tarea 3.*: Prototipo de integración inicial

Id	3.*
Tarea	Prototipo funcional
Duración	10 días
Descripción	Modificar el prototipo anterior añadiendo las funcionalidades esperadas del programa.
Salida	Código añadido a Pyomo.

Cuadro 2.31: Tarea 3.*: Prototipo funcional

2.7.3. Cronograma inicial

Para la realización del trabajo se plantea un ciclo de vida en cascada. Este ciclo de vida nos permitirá realizar un seguimiento del progreso del proyecto en función del tiempo disponible.



Nombre de tarea	Dura	Comienzo	Fin	Pred
Análisis	**18 días**	**jue 01/03/18**	**lun 26/03/18**	
Estudio de Programación Estocástica y Progressive Hedging	7 días	jue 01/03/18	vie 09/03/18	
Análisis de la solución actual en PySP/runph	8 días	lun 12/03/18	mié 21/03/18	2
Estudio de alternativas para la implementación paralela.	3 días	jue 22/03/18	lun 26/03/18	3
Diseño	20 días	mar 27/03/18	lun 23/04/18	1
Implementación	**20 días**	**mar 24/04/18**	**lun 21/05/18**	**5**
Implementación de la solución	15 días	mar 24/04/18	lun 14/05/18	
Optimización	5 días	mar 15/05/18	lun 21/05/18	7
Validación y pruebas	**10 días**	**mar 22/05/18**	**lun 04/06/18**	**6**
Generación de plan de pruebas	4 días	mar 22/05/18	vie 25/05/18	
Ejecución del plan de pruebas	1 día	lun 28/05/18	lun 28/05/18	10
Plan de pruebas de rendimiento	3 días	mar 29/05/18	jue 31/05/18	11
Estudio de rendimiento en comparación a la implementación original	2 días	vie 01/06/18	lun 04/06/18	12
Documentación	10 días	mar 05/06/18	lun 18/06/18	9

Figura 2.3: Línea base

2.7.4. Modificaciones al cronograma inicial

Se ha realizado una estimación temporal inicial poco precisa por no utilizar ningún tipo de método probado ni datos concretos.

Esta es la razón principal para los retrasos que se explican a continuación.

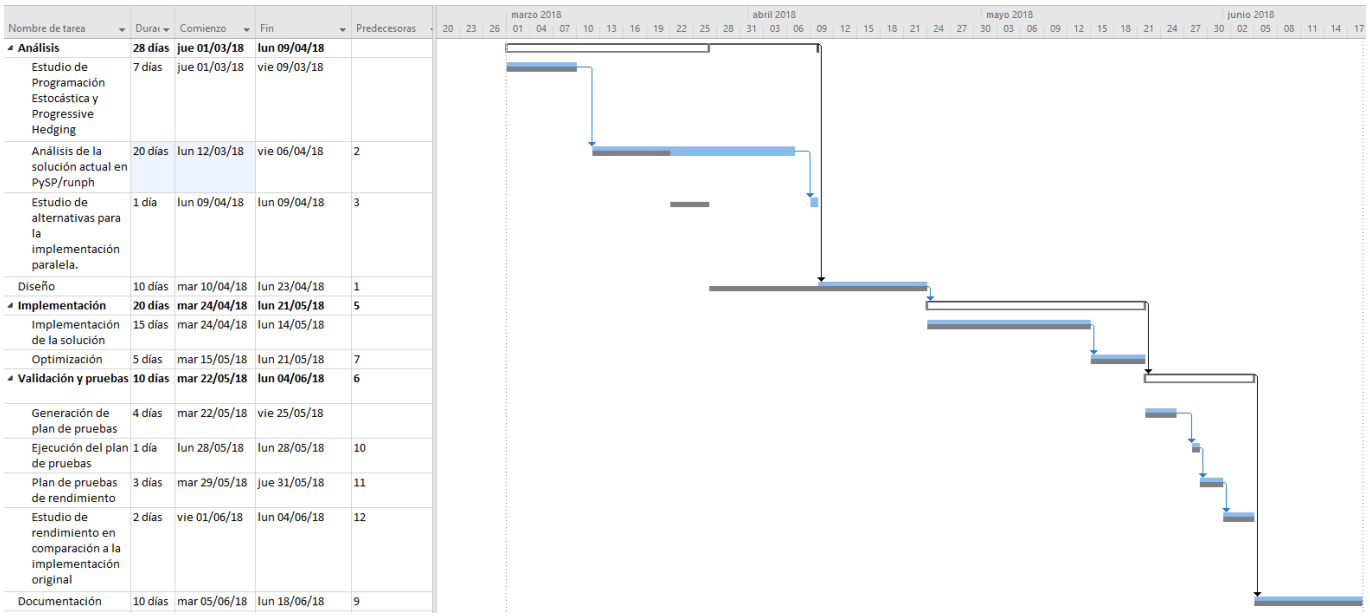
Retraso en análisis

El primer retraso se produce en la fase de análisis de la implementación actual. En esta fase se debe estudiar el funcionamiento del proyecto Pyomo y, en concreto, el módulo de resolución de problemas mediante *Progressive Hedging*.

A pesar de conocer el funcionamiento teórico del algoritmo mediante [3], Pyomo es un proyecto complejo, con multitud de funcionalidades para resolver otros tipos de problemas, soporte para plugins, etc. Todo esto hace que la complejidad del código aumente y sea necesario estudiar múltiples capas de abstracción para entender correctamente el funcionamiento del programa.

Otra complicación añadida es el personal desconocimiento del lenguaje Python previo a la realización de este trabajo.

Tras este primer retraso se intenta ajustar la planificación reduciendo el tiempo de diseño a la mitad:



Nombre de tarea	Dura	Comienzo	Fin	Predecesoras
Análisis	**28 días**	**jue 01/03/18**	**lun 09/04/18**	
Estudio de Programación Estocástica y Progressive Hedging	7 días	jue 01/03/18	vie 09/03/18	
Análisis de la solución actual en PySP/runph	20 días	lun 12/03/18	vie 06/04/18	2
Estudio de alternativas para la implementación paralela.	1 día	lun 09/04/18	lun 09/04/18	3
Diseño	10 días	mar 10/04/18	lun 23/04/18	1
Implementación	**20 días**	**mar 24/04/18**	**lun 21/05/18**	**5**
Implementación de la solución	15 días	mar 24/04/18	lun 14/05/18	
Optimización	5 días	mar 15/05/18	lun 21/05/18	7
Validación y pruebas	**10 días**	**mar 22/05/18**	**lun 04/06/18**	**6**
Generación de plan de pruebas	4 días	mar 22/05/18	vie 25/05/18	
Ejecución del plan de pruebas	1 día	lun 28/05/18	lun 28/05/18	10
Plan de pruebas de rendimiento	3 días	mar 29/05/18	jue 31/05/18	11
Estudio de rendimiento en comparación a la implementación original	2 días	vie 01/06/18	lun 04/06/18	12
Documentación	10 días	mar 05/06/18	lun 18/06/18	9

Figura 2.4: Primer retraso

Teniendo en cuenta que los primeros retrasos fueron principalmente causados por el desconocimiento de la tecnología a usar así como de una mala estimación, es muy probable que en las fases siguientes se produzcan otros retrasos. En este punto se considera entonces abandonar el ciclo de vida en cascada. Su principal atractivo era poder ajustarnos a una planificación temporal que nos permita acabar el proyecto dentro de tiempo, pero esta ventaja no se está cumpliendo en la práctica. Buscando reducir el tiempo de implementación con tecnologías que serán usadas por primera vez, se decide adaptar la planificación a un ciclo de vida por prototipos.

La creación de sucesivos prototipos permite ir acostumbrándose a las tecnologías desconocidas, en este caso Python y Spark, así como ir probando el rendimiento y la integración a medida que se desarrolla.

En primer lugar se crea un prototipo aislado para comprobar la implementación de Spark con una arquitectura similar a la que se implementará en Pyomo. Este primer prototipo sirve como aprendizaje de la instalación de Spark y el despliegue de una aplicación en el mismo, así como la implementación en Python que interactuará con Spark. Es deseable utilizar una arquitectura de objetos Python similar a la que se usará en Pyomo para concretar el uso de Spark y descubrir posibles problemas con la implementación elegida.

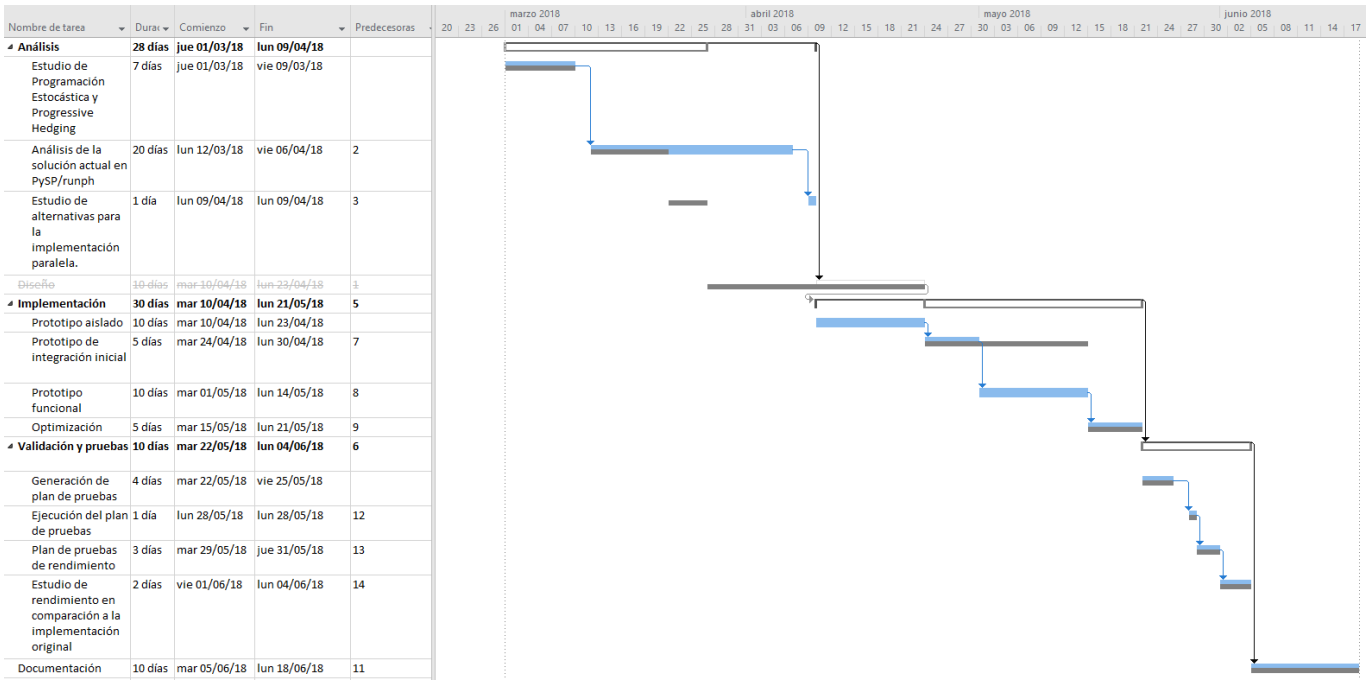
Posteriormente se realizarán prototipos sucesivos sobre Pyomo para integrar el nuevo módulo e ir solucionando posibles problemas de rendimiento o funcionamiento que vayan surgiendo.

Dado que la implementación partirá de un protipo inicial de baja calidad será importante realizar una fase de optimización y refactorización al final de la

implementación para asegurarse un código final de calidad. Definir la calidad del código no es algo trivial y en este caso calificaremos el código como "de calidad"si cumple:

- Funciona correctamente y es resistente a errores. Para esto nos apoyaremos en un plan de pruebas funcional.
- Funciona eficientemente y otorga un buen rendimiento, en comparación a las implementaciones existentes. En este caso nos apoyaremos en el plan de pruebas de rendimiento.
- Se integra adecuadamente al proyecto actual. Debe seguir una filosofía de diseño análoga al resto del código así como funcionar correctamente de forma paralela a todo lo implementado previamente.

Tras esta modificación en la planificación, se genera una nueva planificación que podemos ver en la figura y se guardará como una nueva línea base.



Nombre de tarea	Dura	Comienzo	Fin	Predecesoras
Análisis	**28 días**	**jue 01/03/18**	**lun 09/04/18**	
Estudio de Programación Estocástica y Progressive Hedging	7 días	jue 01/03/18	vie 09/03/18	
Análisis de la solución actual en PySP/runph	20 días	lun 12/03/18	vie 06/04/18	2
Estudio de alternativas para la implementación paralela.	1 día	lun 09/04/18	lun 09/04/18	3
~~Diseño~~	~~10 días~~	~~mar 10/04/18~~	~~lun 23/04/18~~	~~1~~
Implementación	**30 días**	**mar 10/04/18**	**lun 21/05/18**	**5**
Prototipo aislado	10 días	mar 10/04/18	lun 23/04/18	
Prototipo de integración inicial	5 días	mar 24/04/18	lun 30/04/18	7
Prototipo funcional	10 días	mar 01/05/18	lun 14/05/18	8
Optimización	5 días	mar 15/05/18	lun 21/05/18	9
Validación y pruebas	**10 días**	**mar 22/05/18**	**lun 04/06/18**	**6**
Generación de plan de pruebas	4 días	mar 22/05/18	vie 25/05/18	
Ejecución del plan de pruebas	1 día	lun 28/05/18	lun 28/05/18	12
Plan de pruebas de rendimiento	3 días	mar 29/05/18	jue 31/05/18	13
Estudio de rendimiento en comparación a la implementación original	2 días	vie 01/06/18	lun 04/06/18	14
Documentación	10 días	mar 05/06/18	lun 18/06/18	11

Figura 2.5: Línea base prototipos

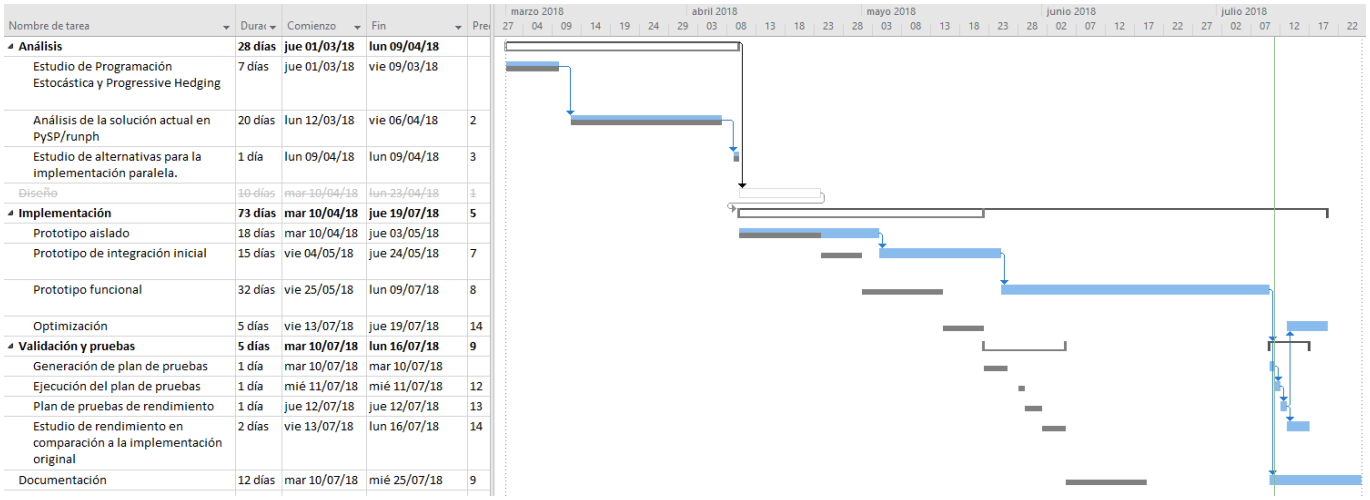
En este punto hemos eliminado la fase de Diseño para poder aumentar el tiempo asignado a Análisis e Implementación. En caso de sufrir más retrasos en la fase de implementación podremos reducir el tiempo asignado a pruebas si el retraso no es grave. En caso de ser un retraso mayor, no cumpliremos la fecha de finalización establecida.

Retraso en implementación

Durante la implementación del prototipo funcional el desarrollo llega a un punto muerto. Las funciones implementadas no devuelven el resultado correcto y se debe hacer una búsqueda de los errores que lo causan. Por falta de experiencia y desconocimiento de las tecnologías, esta fase de implementación se alarga hasta el día 07/07/2018.

Este retraso sumado a un retraso de 10 días en la creación del prototipo aislado nos fuerza a retrasar la fecha de finalización del proyecto al día 25/07/2018.

Con estos nuevos cambios es necesaria una nueva planificación temporal:



Nombre de tarea	Duración	Comienzo	Fin	Pred
Análisis	**28 días**	**jue 01/03/18**	**lun 09/04/18**	
Estudio de Programación Estocástica y Progressive Hedging	7 días	jue 01/03/18	vie 09/03/18	
Análisis de la solución actual en PySP/runph	20 días	lun 12/03/18	vie 06/04/18	2
Estudio de alternativas para la implementación paralela.	1 día	lun 09/04/18	lun 09/04/18	3
~~Diseño~~	~~10 días~~	~~mar 10/04/18~~	~~lun 23/04/18~~	~~1~~
Implementación	**73 días**	**mar 10/04/18**	**jue 19/07/18**	**5**
Prototipo aislado	18 días	mar 10/04/18	jue 03/05/18	
Prototipo de integración inicial	15 días	vie 04/05/18	jue 24/05/18	7
Prototipo funcional	32 días	vie 25/05/18	lun 09/07/18	8
Optimización	5 días	vie 13/07/18	jue 19/07/18	14
Validación y pruebas	**5 días**	**mar 10/07/18**	**lun 16/07/18**	**9**
Generación de plan de pruebas	1 día	mar 10/07/18	mar 10/07/18	
Ejecución del plan de pruebas	1 día	mié 11/07/18	mié 11/07/18	12
Plan de pruebas de rendimiento	1 día	jue 12/07/18	jue 12/07/18	13
Estudio de rendimiento en comparación a la implementación original	2 días	vie 13/07/18	lun 16/07/18	14
Documentación	12 días	mar 10/07/18	mié 25/07/18	9

Figura 2.6: Línea base final

Capítulo 3

Análisis

3.1. Funcionamiento del algoritmo *Progressive Hedging*

El primer paso imprescindible para la realización de este proyecto será entender el algoritmo con el que se trabajará. Se debe estudiar el funcionamiento teórico de la Programación Estocástica y el algoritmo *Progressive Hedging* con el objetivo de entender posteriormente su implementación en PySP.

Como referencia principal se utiliza el libro *Introduction to Stochastic Programming* [2]. Como se explicó anteriormente, la programación estocástica resuelve problemas donde existe un cierto nivel de incertidumbre. A lo largo de esta fase de análisis utilizaremos el ejemplo "Farmer" para ver un problema real en funcionamiento que sea sencillo de entender. Este problema está explicado en el libro anterior e incluido como ejemplo en el repositorio de Pyomo por lo que será una referencia muy útil.

En el problema del granjero, nos encontramos con una superficie concreta donde podemos plantar 3 tipos diferentes de cultivos. El problema de optimización es especificar la cantidad de superficie óptima para cada uno de los cultivos. Para llegar a la solución óptima se deben tener en cuenta múltiples restricciones como el precio por tonelada de cada cultivo, la cantidad de cultivo que conseguimos por unidad de superficie, su coste o cantidad mínima necesaria para venderlo. La incertidumbre en este problema viene dada por la eficiencia de cada cultivo. Aunque plantemos una cantidad concreta en una superficie específica, no podemos asegurar la cantidad de producto que conseguiremos pues depende de multitud de factores externos. Conocemos por datos pasados una aproximación de lo que producirá una cierta plantación, pero debemos considerar los escenarios en que esta producción será menor o superior. Con esto, tenemos un árbol de escenarios como el explicado en la Figura 1.1.

Para resolver este tipo de problemas, se utilizará el algoritmo *Progressive Hedging*. Este algoritmo se ejecuta iterativamente y funciona descomponiendo el árbol de escenarios en problemas individuales. Cada rama del árbol se transforma en un problema lineal que se puede procesar directamente para obtener un resultado. Si simplemente hacemos esta descomposición y resolución, tendremos múltiples soluciones diferentes en función de los escenarios que existían en cada camino. Para poder obtener una solución correcta que tenga en cuenta todos los escenarios, debemos conseguir que todas las soluciones individuales convergan a un mismo valor (considerando un cierto margen de error). Esta convergencia se consigue iterando sobre el procesamiento de estos subproblemas. Evidentemente debemos hacer modificaciones en los mismos para poder conseguir la convergencia. Entre cada iteración se van a modificar las variables de las que depende cada subproblema siguiendo la fórmula definida por el algoritmo. El parámetro principal es la variable ρ, que no es específica de ningún problema. Esta variable se puede denominar como "penalizador" que influirá en cómo se modifican el resto de variables para hacer que sus valores tiendan a la solución buscada.

En la siguiente figura vemos el pseudo-código de este algoritmo. *Progressive Hedging* puede denominarse como un método de descomposición horizontal porque descompone el problema en escenarios en vez de dividirlo por las fases temporales. Para cada escenario $s \in S$, buscamos minimizar la expresión $c \cdot x_s$, donde x_s es un vector de decisiones y c es un vector de coeficientes de coste. Se debe cumplir que $x_s \in Q_s$, es decir, que la solución se ajusta a las limitaciones definidas para el escenario. La incertidumbre del problema se define como la probabilidad de ocurrencia de los escenarios. Esta probabilidad se especifica como $Pr(s)$. Siendo x el vector de decisiones agnóstico al escenario y $f_s \cdot y_s$ el coste y valores de decisión para la segunda fase del problema, debemos minimizar la expresión $(c \cdot x) + \sum_{s \in S} Pr(s)(f_s \cdot y_s)$.

$k := 0$

$\Delta s \in S, x_s^{(k)} := argmin_x(c \cdot x + f_s \cdot y_s) : (x, y_s) \in Q_s$

$\hat{x}^{(k)} := \sum_{s \in S} Pr(s) x_s^{(k)}$

$w_s^{(k)} := \rho(x_s^{(k)} - \hat{x}_s^{(k)})$

$k := k + 1$

$\Delta s \in S, x_s^{(k)} := argmin_x(c \cdot x + w_s^{(k-1)} x + \rho/2 \, ||x - \hat{x}^{(k-1)}||^2 + f_s \cdot y_s) \in Q_s$

$\hat{x}^{(k)} := \sum_{s \in S} Pr(s) x_s^{(k)}$

$\Delta s \in S, w_s^{(k)} := w_s^{(k-1)} + \rho(x_s^{(k)} - \hat{x}^{(k)})$

$g^{(k)} := \sum_{x \in S} Pr(s) ||x^{(k)} - \hat{x}^{(k)}||$

If $g^{(k)} < \epsilon$ repeat from 5, else terminate

Figura 3.1: Pseudocódigo del algoritmo *Progressive Hedging* [3]

En la primera iteración $k := 0$, se calcula la solución de cada escenario (x_s) mediante una minimización como vimos anteriormente. Se calcula un valor $\hat{x}$ como solución única para todos los escenarios teniendo en cuenta la probabilidad de cada uno. En este punto calculamos un peso para cada escenario (w_s) utilizando el penalizador ρ sobre la diferencia entre la solución concreta del escenario y la solución que tiene en cuenta las probabilidades.

En las siguientes iteraciones, el cálculo de la solución tiene en cuenta los pesos calculados en la iteración anterior y el valor ρ. Como criterio de parada, se calcula el valor de convergencia g.

3.2. Analizando Pyomo

Una vez entendido el funcionamiento teórico del algoritmo, procedemos a ver la implementación del mismo en Pyomo. Podemos acudir a la propia documentación de PySP para hacernos una idea inicial de su funcionamiento. En este documento [17] se especifica cómo ejecutar el algoritmo mediante el comando `runph`. También especifica cómo utilizar Pyro para resolver el problema de forma paralela.

En este punto, necesitamos desplegar el proyecto para poder verlo en funcionamiento. El entorno sobre el que trabajaremos es el siguiente:

- Para trabajar con el código se ha realizado un *fork* del proyecto original en GitHub.
- Como IDE se utilizará PyCharm sobre Fedora 27.
- El código funcionará sobre un entorno virtual de Python 3.6 creado específicamente para este proyecto.

El primer paso será establecer una configuración de ejecución para el ejemplo *Farmer*. Pyomo incluye una opción `--verbose` que proporciona una salida muy detallada de la ejecución y será extremadamente útil para este análisis, sobre todo, analizando la ejecución con Pyro.

Pyomo define comandos concretos para la ejecución de sus distintos módulos así que debemos buscar cuál es el punto de entrada del comando `runph`. Vemos que se utiliza un lenguaje de etiquetas personalizadas `@pyomo_command` para definir estos puntos de entrada concretos. Si queremos ejecutar estos comandos desde el IDE, debemos hacer una modificación al archivo que queramos ejecutar para hacerlo ejecutable como script. En este caso, el archivo *phinit.py* es el punto de entrada del método `runph`:

```
1  @pyomo_command('runph', 'Optimize with the PH solver (primal search)')
2  def PH_main(args=None):
3      return main(args=args)
4
5
6  if __name__ == "__main__":
7      PH_main()
```

Con esta modificación podemos ejecutar el archivo como un script y será equivalente a ejecutar el comando `runph`. Podemos crear una configuración de ejecución en el IDE y usar la ejecución paso a paso para tener una visión general de la implementación.

Usando una herramienta incluida en PyCharm, podemos generar un árbol visual de llamadas a métodos. Esto genera la imagen Figura 3.2 que representa el flujo que sigue la ejecución secuencial del algoritmo PH.

Podemos ignorar la rama de la derecha que se limita a la inicialización de los módulos Python. Lo relevante comienza en el método `run_ph`. Vemos que se ejecuta el método `iteration_k_solves` y, aunque no está especificado en el diagrama, existe un método similar para la iteración 0, pues esta primera iteración es diferente al resto en el algoritmo PH. En el siguiente paso observamos la primera

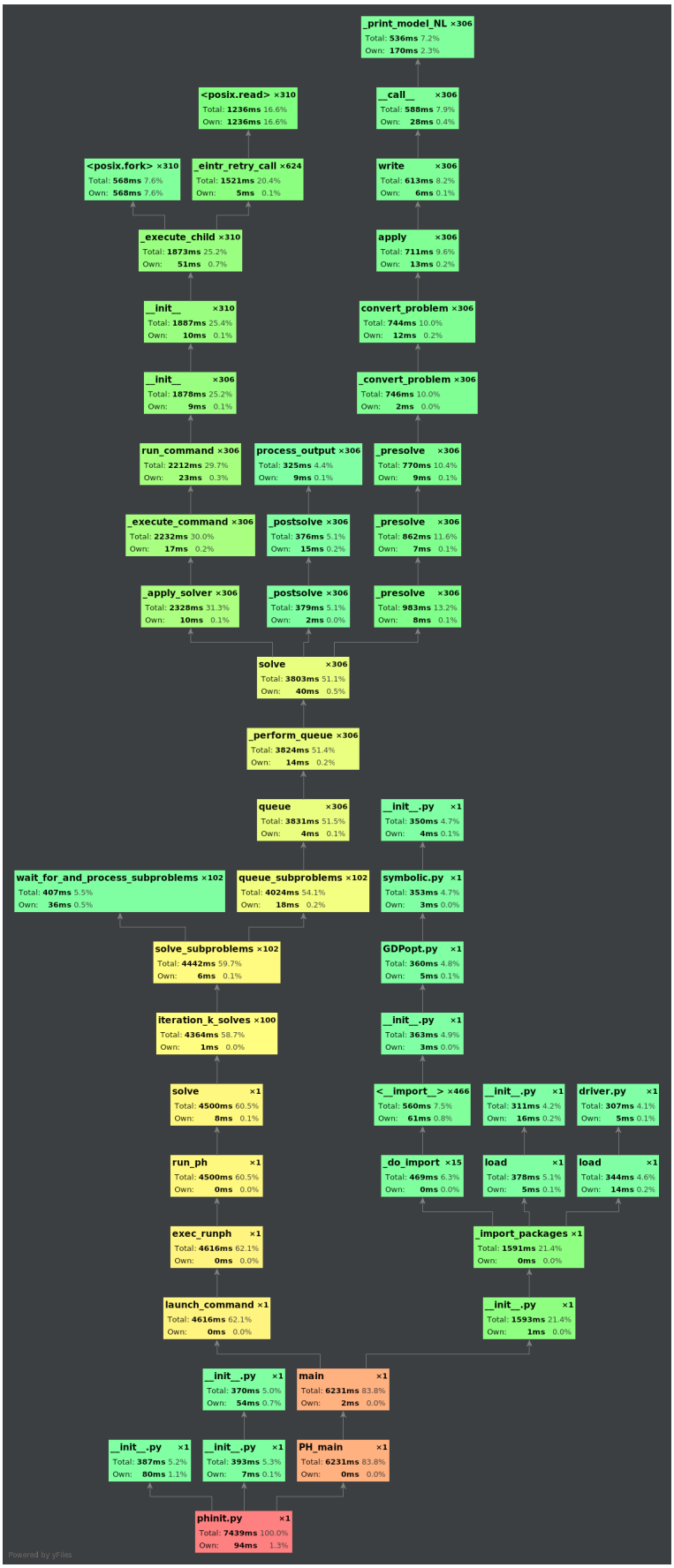


Figura 3.2: Árbol de llamadas en runph

decisión de diseño importante, se utiliza una cola para solucionar los problemas. Para cada subproblema especificado en el árbol se genera una tarea y se pone en cola. En un segundo método, se espera por los resultados. En esta ejecución secuencial esto no es muy relevante, pero lo será cuando veamos la implementación paralela.

En cuanto a la ejecución de la resolución, podemos distinguir tres fases: presolve, solve y postsolve:

- En el **presolve** se preprocesan los escenarios del problema y se escriben en un archivo con el formato especificado. En este caso se escribe un archivo con el formato "nl" (`_print_model_NL`) aceptado por solvers que siguen la interfaz AMPL [9].
- En **_apply_solver** se utiliza el archivo generado en el paso anterior y la ruta al ejecutable del solver escogido (en nuestro caso usaremos "minos" [12]) para crear un comando. Esto se ejecuta como un comando en el sistema operativo y Python recoge el resultado.
- En el **postsolve** se procesa la salida recuperada del solver y se adapta en función del formato de salida.

En el método `solve` de `runph` es donde podemos encontrar el bucle principal que solicita al solver externo el resultado de cada iteración y realiza los cálculos necesarios para modificar los valores de cada subproblema. Al final de cada iteración se hace una prueba de convergencia que será la que indique la finalización del algoritmo.

Para nuestra implementación paralela analizaremos a continuación la implementación actual con Pyro. En este caso, por ser necesario lanzar múltiples procesos a la vez (el servidor de nombres, el propio `runph` y los workers), no podemos utilizar la ejecución paso a paso del IDE. En esta fase del análisis haremos usaremos la salida generada por la ejecución que, como dijimos antes, puede ser bastante específica haciendo uso de la opción `--verbose`. Analizando esta salida, podemos generar una traza con las operaciones realizadas:

- Import model & scenario.
- Construct solver manager.
- Construct solver.
- Wait for n servers to initialize.

- Instance each scenario.
- Post-instance-creation PHSolverServer plugins.
- Post-initialization PHSolverServer plugins.
- Post-initialization PH plugins.
- **Start PH**
  - Iteration 0.
    - Sync fixed variable status.
    - Queue each solve for each scenario.
    - Wait for scenario sub-problem solves.
    - Pre-iteration-0-solve PHSolverServer plugins.
    - Solve scenarios.
    - Load all solutions.
    - Post-iteration-0-solve PHSolverServer plugins.
    - Convergence test.
    - Transmitting request to activate PH objective proximal terms to PH solver servers.
    - Transmitting request to activate PH objective weight terms to PH solver servers.
  - Iteration k.
    - Sync fixed variable status.
    - Transmit xbars.
    - Transmit weights.
    - Transmit rhos.
    - Queue solve for each scenrio.
    - Wait for scenario sub-problem solves.
    - Pre-iteration-k-solve PHSolverServer plugins.
    - Solve scenarios.
    - Load all solutions.
    - Post-iteration-k-solve PHSolverServer plugins.

En esta traza vemos la importancia que tiene el uso de una cola en la resolución de los problemas de forma paralela. El hilo principal pondrá tareas en esta cola y estas serán distribuidas a los distintos nodos de trabajo. Debemos identificar los objetos que están llevando esto a cabo. El objeto `PHSolverServer` es el encargado de ejecutar la resolución de cada subproblema y todas las tareas que se le pasen desde el hilo principal. Tendremos una instancia de este objeto por

cada subproblema a resolver. En el ejemplo del *Farmer*, como disponemos de 3 escenarios, tendremos tres instancias de `PHSolverServer` trabajando en paralelo. A estos objetos que realizan el trabajo en paralelo se los denominará normalmente como *workers* a lo largo de este trabajo. El otro objeto relevante es el solver manager. Este objeto es el que define la distribución del trabajo. En el caso de la ejecución secuencial se utiliza `--solver-manager=serial` y, en este ejemplo, `--solver-manager=phpyro`. El solver manager se encarga de gestionar la cola de trabajos y especifica cómo se ejecutan.

En la traza anterior podemos observar claramente la comunicación entre el hilo principal y los workers. En cada iteración se solicita la resolución de un problema y se recoge su solución. Con esta solución se calculan las nuevas variables de control que serán retransmitidas al principio de la siguiente iteración.

Llegados a este punto tenemos una idea bastante concreta de la implementación del algoritmo en Pyomo. Con este conocimiento podemos comenzar el análisis de tecnologías teniendo en cuenta su posible integración y, posteriormente, comenzar con el diseño de la solución.

3.3. Análisis de herramientas

3.3.1. Spark

La principal opción para la implementación de este proyecto es Spark. Como se explicó anteriormente, Spark es un framework de código abierto para la programación en entornos distribuidos. Es una herramienta encuadrada en el campo del Big Data y ofrece distintos módulos de funcionamiento.

El módulo principal de Spark se basa en la utilización de RDDs (*"Resilient Distributed Datasets"*). Un RDD es una colección de elementos particionados en los nodos del cluster en el que se ejecute Spark. Las operaciones sobre estos objetos se realizarán automáticamente en paralelo. Para trabajar con estos RDDs, Spark define transformaciones y acciones. Una transformación modifica los elementos del RDD para generar uno nuevo. Las acciones procesan el RDD para devolver un resultado concreto. Esta diferencia es clave para entender el funcionamiento de Spark. Spark hace uso de la evaluación perezosa para las transformaciones, es decir, cuando definimos una transformación no se ejecuta directamente, se guarda en memoria. Estas transformaciones se pueden encadenar, facilitando que Spark optimice su ejecución, y se procesan cuando se llama a una acción. En resumen, el flujo de un programa en Spark usando RDDs comienza definiendo el RDD, encadenando las transformaciones que queremos ejecutar y finalizamos con una acción que nos devuelva el resultado esperado.

Otros módulos de Spark son:

- **Spark SQL:** Utiliza *DataFrames* que son objetos distribuidos pero con información estructurada. Esto facilita la recolección de datos identificables y permite usar el lenguaje SQL.
- **Spark Streaming:** Facilita el análisis de datos en tiempo real. Utilizando una fuente continua de datos (como puede ser una red de sensores), puede generar resultados en tiempo real.
- **Spark MLib:** Librería optimizada para el machine learning. Implementa funciones comunes en este paradigma de programación.
- **Spark GraphX:** Librería optimizada para el análisis de datos ordenados como grafos. Es un caso de uso muy común en redes sociales o otro tipo de datos distribuidos e interconectados.

Para el problema entre manos, lo más adecuado sería el uso de RDDs, instanciando los workers como un objeto distribuido y enviando las tareas mediante la interfaz de Spark.

3.3.2. MPI

El estándar MPI permite el paso de mensajes entre objetos remotos. En este proyecto, MPI funcionaría de una forma similar a la implementación actual con Pyro. Sería necesario definir una interfaz como parte de los workers para recibir las tareas concretas. Podemos aprovechar las características concretas de MPI para enviar datos en masa a todos los workers al mismo tiempo (con MPI_BCAST o MPI_SCATTER) y, de forma similar, recoger las soluciones de todos en una sola operación (con MPI_GATHER).

Como MPI es una interfaz de bajo nivel, hay varias cuestiones que debemos tener en cuenta para llevar a cabo una buena implementación. En cuanto a la gestión de la memoria, debemos establecer un método eficiente para subdividir los problemas y asignarlos a los procesos adecuados. Otros datos tendrán que ser enviados a todos los procesos a la vez y también debemos gestionar la recepción de soluciones. Todos estos envios y recepciones deben estar sincronizados para evitar bloqueos. Si queremos aprovechar las funciones asíncronas habrá que tener cuidado adicional para asegurar su correcto funcionamiento. Adicionalmente, el hecho de que MPI sea un estándar compatible con múltiples lenguajes hace que los mensajes enviados sigan un formato concreto de MPI. Si queremos enviar algún tipo de dato complejo tendremos que definirlo utilizando las herramientas

proporcionadas por MPI.

Estos cambios podrían suponer una optimización relevante con respecto a la implementación actual con Pyro.

Capítulo 4

Diseño

Tras analizar las características tanto de Spark como de MPI, decidimos continuar la implementación haciendo uso de Spark. La herramienta de Apache tiene un mayor potencial de optimización a largo plazo y debería ser más escalable. Además, una implementaión con MPI no se alejaría demasiado de la implementación actual con Pyro.

Tal y como se especifica en la Subsección 2.7.4, la fase de diseño es eliminada por falta de tiempo. Sin embargo, esto no significa que pasemos directamente a la implementación de los primeros prototipos. Sigue siendo necesario especificar el diseño que se va a implementar, aunque solo sea una representación de alto nivel.

A partir del análisis anterior podemos realizar un diagrama de alto nivel.

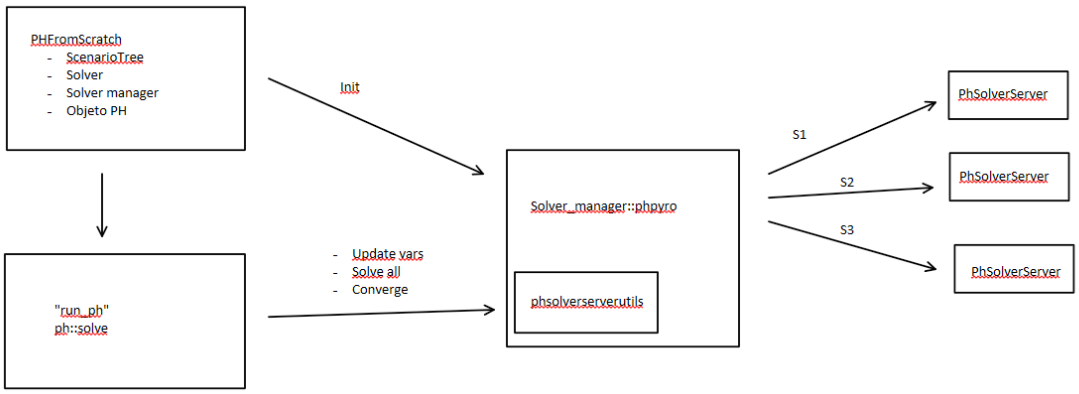


Figura 4.1: Diseño de runph

Si tomamos el solver manager como el punto central, la parte de la derecha se ejecuta en paralelo, es decir, debemos implementarla en Spark; y la parte iz-

quierda es la que genera los trabajos.

Teniendo en cuenta esta arquitectura y el tiempo disponible se propone una solución cuyo objetivo es aprovechar al máximo el código ya implementado en Pyomo y que facilite al máximo la integración de la nueva implementación. La solución propuesta es la creación de un nuevo solver manager que gestione los objetos *PHSolverServer* como RDDs, donde se distribuyen los subproblemas. Con esto mantenemos el mismo sistema de ejecución mediante cola de tareas y sólo modificamos el gestor que las distribuye.

Capítulo 5

Implementación

5.1. Prototipo inicial

El objetivo de este primer prototipo es familiarizarse con el uso de Spark desde Python así como el manejo de una lista de objetos mediante un RDD.

Para que el prototipo sea de utilidad, sigue una arquitectura similar al diseño planteado para Pyomo. En este caso contamos con dos archivos distintos:

- **main.py:** Representa las funciones del nuevo solver manager, gestionando la interacción con Spark y el envio de tareas a los workers.
- **worker.py:** Implementa el objeto que realizará el procesamiento de los datos. Representa a los objetos PHSolverServer.

En la Figura 5.1 se muestra el código de `main.py`. En este prototipo vemos cómo conectarnos a una instancia de Spark utilizando el objeto `SparkConf`. A continuación creamos un RDD a partir de una lista de objetos utilizando el método `parallelize`. La ejecución de cada iteración se hace iterando sobre los elementos del RDD, es decir, la lista de workers, mediante la función `map` e indicando una función como parámetro de la expresión lambda. Esto ejecuta una transformación sobre el RDD, ejecutando la función definida en cada elemento. Para que que esta transformación se llegue a ejecutar, es necesario utilizar una acción de Spark. En este prototipo, la acción realizada es `collect`, que nos devuelve una lista con los contenidos del RDD tras ejecutar las transformaciones anteriores.

Los workers en este prototipo simplemente realizan un cierto número de operaciones aleatorias para simular un cierto tiempo de ejecución y comprobar si la ejecución de Spark se está realizando en paralelo y qué niveles de escalabilidad nos podemos esperar.

```python
def do_iteration(worker, scenario_name, var1, var2):
    worker.update_vars(var1.value, var2.value)
    worker.solve(scenario_name)


if __name__ == "__main__":
    conf = SparkConf().setMaster("spark://localhost:7077").setAppName("Test")
    sc = SparkContext(conf=conf)

    scenarioSize = 20

    workerList = []
    scenarioList = []
    for i in range(scenarioSize):
        workerList.append(Worker(i))
        scenarioList.append("Scenario " + str(i))

    parallelWorkerList = sc.parallelize(zip(workerList, scenarioList)).persist()

    for x in range(20):
        var1 = randint(100, 10000)
        var2 = randint(100, 10000)
        print("Initializing iteration " + str(x))

        print("Updating variables: ")
        print("\tvar1: " + str(var1))
        print("\tvar2: " + str(var2))

        broadcast1 = sc.broadcast(var1)
        broadcast2 = sc.broadcast(var2)

        solved_values = parallelWorkerList.map(lambda item: do_iteration(item[0], item[1], broadcast1, broadcast2)).collect()
```

Figura 5.1: Prototipo inicial

5.2. Integración

La siguiente fase del prototipado es comenzar la implementación sobre el proyecto final. Este prototipo tiene como objetivo probar la integración con el proyecto y verificar que el flujo de ejecución es correcto.

El primer paso es crear el nuevo solver manager que denominaremos *PHSpark*. Se crea una nueva clase en la ruta *"/pyomo/pyomo/solvers/plugins/smanager"* y comenzamos copiando el código de phpyro. De esta forma nos aseguramos de mantener una interfaz común y sólo tendremos que cambiar la implementación de los métodos. Si queremos que la integración sea correcta, este solver manager debe poder seleccionarse como una opción de `runph` desde la línea de comandos. Para conseguir que funcione la opción `--solver-manager=phspark` debemos especificar "phspark" como nombre del plugin que será el nuevo solver manager. Esto se consigue usando el método `pyomo.util.plugin.alias(‘‘phspark’’)` en la definición de nuestro solver manager. También es necesario importarlo como parte del paquete de smanagers en el archivo `/plugins/smanager/__init__.py`.

La primera tarea que realizan los workers es la inicialización por lo que se establece como criterio de aceptación de este prototipo la ejecución correcta de esta tarea por parte de los workers (funcionando en Spark) y la devolución correcta de los resultados.

```python
def acquire_servers(self, servers_requested, timeout=None):

        # TODO: Manage errors
        spark_url = "spark://" + self.host + ":" + str(self.port)

        if self._verbose:
            print("Initializing spark context on %s" % spark_url)

        # conf = SparkConf().setMaster("spark://" + self.host + ":" + str(self.port)).setAppName("pyomo")

        # TODO: connect to actual spark
        conf = SparkConf().setMaster("local").setAppName("Test")

        self._sparkContext = SparkContext(conf=conf, serializer=CloudPickleSerializer())
        dependency_path = pkg_resources.resource_filename('pyomo.pysp', 'phsolverserver.py')
        print ("Trying to add " + dependency_path)
        self._sparkContext.addPyFile(dependency_path)

        from phsolverserver import PHSparkWorker
        server_list = []
        for i in range(servers_requested):
            server_list.append(PHSparkWorker(i))
            self.server_pool.append(i)

        self._rddWorkerList = self._sparkContext.parallelize(server_list).persist()

        print("Requested %d servers" % servers_requested)
```

Figura 5.2: Método `acquire_servers` del prototipo de integración.

El primer paso es la creación del RDD. Esto se realiza mediante la función `acquire_servers`. Como podemos ver en la Figura 5.2, es una implementación muy similar al prototipo anterior. En este punto destaca la necesidad de cargar el archivo donde se encuentra la implementación de los solvers (*phsolverserver.py*) en Spark. Si no cargamos este archivo manualmente se producirá un error porque el worker desde la instancia de Spark no puede resolver la dependencia. Este tipo de problemas serán comunes a lo largo de todo este proyecto. También cabe destacar la sustitución del serializador por defecto de pyspark. En este método estamos inicializando `CloudPickleSerializer` pues es necesario para serializar las funciones lambda que se pasarán como argumento al método `map` del RDD.

Una vez tenemos el RDD creado, el solver manager queda a la espera de que el hilo principal le ponga tareas en cola. Estas tareas son aceptadas en el método `_perform_queue`. Podemos ver la implementación inicial en la Figura 5.3. De nuevo, es muy similar al prototipo anterior. Se genera una tarea de Pyro y se envía a cada worker. La variable `queue_name` indica a qué worker concreto está destinada la tarea y la usaremos para filtrar la ejecución.

Gracias a la evaluación perezosa de Spark, como en este método sólo realizamos una transformación sobre el RDD, podemos encadenar varias tareas y se ejecutarán cuando se solicite un resultado por parte del hilo principal.

En la implementación del worker, generamos un wrapper (`PHSparkWorker`)

```
def _perform_queue(self, ah, *args, **kwds):
        """
        Perform the queue operation.  This method returns the
        ActionHandle, and the ActionHandle status indicates whether
        the queue was successful.
        """

        def _do_parallel_work(worker, task, id):
            print("Requested work on worker " + str(worker.id) + " to queue with id: " + str(id))
            if worker.id == id:
                worker.process(task)
            return worker

        queue_name = kwds["queue_name"]

        task = pyutilib.pyro.Task(data=kwds,
                                  id=ah.id,
                                  generateResponse=generateResponse)

        print("")
        print ("[PHSpark_Manager]: Requested action " + task['data']['action'])
        print ("[PHSpark_Manager]: Task id " + str(task['id']))
        print("Requested action on queue with name: " + str(queue_name))

        if self._bulk_transmit_mode:
            if queue_name not in self._bulk_task_dict:
                self._bulk_task_dict[queue_name] = []
            self._bulk_task_dict[queue_name].append(task)

        else:
            self._rddWorkerList = self._rddWorkerList.map(lambda worker:
                                                          _do_parallel_work(worker, task, queue_name))

        # only populate the action_handle-to-task dictionary is a
        # response is expected.
        if generateResponse:
            self._ah[task['id']] = ah
```

Figura 5.3: Método `_peform_queue` del prototipo de integración.

```python
class PHSparkWorker():

    def __init__(self, id):
        self._solver_server = _PHSolverServer(modules_imported=None)
        self.WORKERNAME = "SparkWorker_%d@%s" % (os.getpid(),
                                                 socket.gethostname())
        self._solver_server.WORKERNAME = self.WORKERNAME
        self.id = id
        self._result_queue = []

    def process(self, task):
        data = pyutilib.misc.Bunch(**task['data'])
        print("")
        print ("[PHSparkWorker]: Requested action " + data.action)

        with PauseGC():
            result = self._solver_server.process(data)

        if task['generateResponse']:
            task['result'] = result
            self._result_queue.append(task)
        return result

    def get_results(self):
        print("Requested results. Stored: " + str(self._result_queue))
        if len(self._result_queue):
            return_list = self._result_queue
            self._result_queue = []
            return return_list
```

Figura 5.4: Objeto `PHSparkWorker` del prototipo de integración.

que encapsula el objeto `PHSolverServer` ya implementado. Este objeto pasará las tareas al objeto que encapsula y guarda los resultados en una lista privada. Cuando se solicite un resultado desde el hilo principal, se devolverán los resultados acumulados en esta lista hasta el momento. La implementación de este objeto se puede ver en la Figura 5.4.

Tras la inicialización del worker, debemos eliminar el parámetro *scenario_tree_factory*, pues no es serializable e impide que se devuelvan los datos al hilo principal.

Con esta implementación conseguimos generar el RDD y ejecutar la tarea de inicialización satisfactoriamente. Aunque las siguientes iteraciones producen errores, el objetivo de este prototipo está cumplido con éxito y pasamos a la siguiente fase.

5.3. Prototipo funcional

El esqueleto del nuevo solver manager ya está implementada y en esta fase nos centraremos en implementar lo necesario para que el algoritme finalice correctamente. En lugar de estar centrado en generar nuevo código, será una etapa donde se irán analizando los errores que aparezcan y solucionándolos.

Tras la finalización del anterior prototipo, observamos un error en las siguientes iteraciones del algoritmo. Este primer error sólo se trata de un problema de dependencias en el entorno de ejecución de Spark y se puede solucionar simplemente retrasando la importación del módulo `pyomo.environ` dentro del worker.

Esta primera versión llega a ejecutar las 100 iteraciones que tiene el algoritmo como máximo por defecto, pero acaba con un error. Si observamos el historial de convergencia que se imprime al final, los valores del algoritmo no avanzan. Sabemos que el algoritmo es diferente para la iteración 0 que para el resto y estamos observando que sólo la primera iteracion funciona correctamente. Encontramo que en los workers hay una variable que regula el comportamiento del `solve`, indicando si se encuentra en la primera iteración. Si seguimos su valor en las múltiples iteraciones, siempre estará como `True`, a pesar de que se le asigne expresamente el valor `False`. Una primera conclusión es la existencia de errores en la fase de serialización del RDD que, por alguna razón, impidan que este dato se guarde actualizado. Como no se encuentra el problema raiz, se fuerza el envío del número de iteración desde el hilo principal para poder avanzar. Con esto conseguimo que el algoritmo avance, pero converge en 33 iteraciones cuando debería llegar a la 100. El error en esta variable puede indicar que haya errores en otros datos que no persistan y esto supuso el primer retraso importante en el proyecto.

Este programa es bastante resistente frente a errores lo que implica que, aunque suceda un error, el programa puede continuar funcionando. Sin embargo, el resto de la ejecución, aunque no se pare, puede devolver resultados erróneos. Esto hace que encontrar errores sea bastante complicado.

Un problema recurrente en este proyecto son las dependencias. Al ser un proyecto con múltiples paquetes dependientes unos de otros, se hace complicado llevar uno de estos objetos a Spark. Cuando ejecutamos el objeto `PHSolverServer` dentro del entorno de Spark, aunque teóricamente se serializan sus dependencias, no accede correctamente a todos los archivos a los que tendría acceso en local. En primer lugar, debemos hacer que el solver esté disponible para los workers. Como estamos usando el solver *minos* que consta de un solo archivo ejecutable, es sencillo añadir este archivo únicamente como dependencia dentro de Spark mediante el método `addFile`. Para que los workers puedan utilizarlo deben conocer la ruta que tendrá este archivo en los nodos que se ejecuten. Para conseguir esta nueva

ruta podemos utilizar la función de Spark `SparkFiles.get`. Por último debemos añadir el archivo que define el problema a solucionar, normalmente con el nombre de *ReferenceModel.py*, pues se intenta añadir el archivo como un módulo y esto lo guarda con una ruta de acceso al archivo.

Otro caso de dependencias que debemos solucionar es el uso del patrón *factory* desde dentro de Spark. Estas clases deben tener acceso a una lista de clases que pueden instanciar y esto puede no estar disponible desde un worker. La solución temporal a este problema es instanciar las clases necesarias para la ejecución fuera del RDD y luego pasar las instancias a los workers. Esto en principio limita las opciones que soporta nuestra nueva implementación, pues sólo tenemos acceso a un objeto para escribir archivos "nl", sólo podemos transformar modelos con "mpec.nl" y sólo podemos abrir soluciones en formato "sol".

Por último, observamos un problema en la creación del archivo necesario para la resolución del subproblema. Para generar este archivo se utiliza un objeto del tipo `ComponentMap` que contiene información sobre las variables del problema. Sin embargo, después de la primera iteración del solve, este objeto no tiene los datos correctos. Haciendo un seguimiento de sus contenidos, observamos que en el momento previo a la serialización, los contenidos son correctos, pero justo tras deserializar y comenzar la siguiente iteración, el objeto no se genera correctamente porque falta un atributo. Este atributo que falta es el `_ampl_repn` y, buscando en el código, se encontró lo siguiente en la función de serialización de uno de los objetos:

```
1  # Note sure why we are deleting these...
2  if '_canonical_repn' in ans:
3      del ans['_canonical_repn']
4  if '_ampl_repn' in ans:
5      del ans['_ampl_repn']
6  return ans
```

Figura 5.5: Borrado del parámetro `_ampl_repn`.

Una vez comentadas las líneas del segundo *if*, podemos ejecutar el ejemplo "Farmer" y obtenemos el resultado correcto. En este punto consideramos el proyecto como funcionalmente correcto y damos esta fase como concluida.

En la Figura 5.6 podemos ver cómo se inicializan los workers. Si lo comparamos con el anterior prototipo podemos ver una gestión de errores más completa y la instanciación de los workers se hace dentro de Spark (en lugar de paralelizar los objetos instanciados).

```
def acquire_servers(self, servers_requested, timeout=None):

        os.environ["PYSPARK_PYTHON"] = "/home/crist/python-venv/pyomo3/bin/python"

        connection_url = "spark://%s:%d" % (self.host, self.port)

        if self._verbose:
            print("Connecting to Spark on %s" % connection_url)

        conf = SparkConf().setMaster(connection_url).setAppName("Pyomo")

        try:
            self._sparkContext = SparkContext(conf=conf, serializer=CloudPickleSerializer())
            dependency_path = pkg_resources.resource_filename('pyomo.pysp',  'phsolverserver.py')
            self._sparkContext.addPyFile(dependency_path)
            # TODO: Get paths
            self._sparkContext.addFile("/home/crist/Downloads/minos/minos")
        except Py4JJavaError as e:
            if e.java_exception.getClass().getName() == 'java.lang.IllegalArgumentException':
                raise RuntimeError("ERROR connecting with Spark at %s. Check if Spark is running on that IP. (%s)%s" %
                                   (connection_url,
                                    e.java_exception.getClass().getName(),
                                    e.java_exception.getMessage()))
            elif e.java_exception.getClass().getName() == 'org.apache.spark.SparkException':
                raise RuntimeError("ERROR connecting with Spark at %s. URL might be incorrect (%s)%s" %
                                   (connection_url,
                                    e.java_exception.getClass().getName(),
                                    e.java_exception.getMessage()))
            else:
                raise RuntimeError("ERROR connecting with Spark at %s. (%s)%s" %
                                   (connection_url,
                                    e.java_exception.getClass().getName(),
                                    e.java_exception.getMessage()))

        from phsolverserver import PHSparkWorker

        self.server_pool = range(servers_requested)

        factories_created = {
            '_transformationFactoryInstance': TransformationFactory('mpec.nl'),
            '_problemConverters': [c for c in ExtensionPoint(IProblemConverter)],
            '_nlWriter': WriterFactory('nl'),
            '_solReader': ReaderFactory('sol'),
            '_pluginList': [p for p in ExtensionPoint(IPHSolverServerExtension)]
        }

        self._rddWorkerList = self._sparkContext.parallelize(self.server_pool)
        self._rddWorkerList = self._rddWorkerList.map(lambda id : PHSparkWorker(id, **factories_created))
```

Figura 5.6: Método `acquire_servers` del prototipo funcional.

En el método de asignación de tareas (Figura 5.7) no hay demasiadas diferencias con el anterior. La principal adición está en la ruta relativa del ejecutable del solver dentro de Spark, que se pasa como un keyword de la tarea “initialize”.

```
def _perform_queue(self, ah, *args, **kwds):
        """
        Perform the queue operation.  This method returns the
        ActionHandle, and the ActionHandle status indicates whether
        the queue was successful.
        """

        def _do_parallel_work(worker, task, id):
            if worker.id == id:
                worker.process(task)
            return worker

        task = pyutilib.pyro.Task(data=kwds,
                                  id=ah.id,
                                  generateResponse=generateResponse)

        if self._verbose:
            print ("[PHSpark_Manager]: Requested action (%d)%s" %
                   (task['id'], task['data']['action']))

        data = pyutilib.misc.Bunch(**task['data'])
        if data.action == "initialize" and data.solver_type == "minos":
            minosPath = SparkFiles.get("minos")
            kwds["solver_path"] = os.path.dirname(os.path.abspath(minosPath))
            self._sparkContext.addPyFile(data.model_location)

        if self._bulk_transmit_mode:
            if queue_name not in self._bulk_task_dict:
                self._bulk_task_dict[queue_name] = []
            self._bulk_task_dict[queue_name].append(task)

        else:
            self._rddWorkerList = self._rddWorkerList.map(lambda worker:
                                                          _do_parallel_work(worker, task, queue_name)).cache()

        # only populate the action_handle-to-task dictionary is a
        # response is expected.
        if generateResponse:
            self._ah[task['id']] = ah

        return ah
```

Figura 5.7: Método `_peform_queue` del prototipo funcional.

El último método que merece explicación adicional es el encargado de la recolección de resultados desde Spark (Figura 5.8). Como estamos trabajando con un algoritmo iterativo, siempre que se realice una transformación sobre el RDD debemos mantener las instancias de `PHSparkWorker` actualizadas. En este caso, cuando realicemos la transformación en el RDD, queremos devolver la lista de resultados pendientes, pero también es necesario devolver el worker actualizado. Para conseguirlo, transformamos el RDD en un RDD formado por tuplas con el worker actualizado y sus resultados pendientes. Ahora filtramos esta tupla dos veces: primero cogemos sólo los resultados pendientes y los devolvemos al manager y luego cogemos sólo los workers actualizados para tener el RDD correcto en la siguiente iteración.

```
def _perform_wait_any(self):

        def _get_result_pair(worker):
            results = worker.get_results()
            return worker, results

        start_time = time.time()

        if len(self._results_waiting) > 0:
            return self._extract_result()

        self._rddWorkerList = self._rddWorkerList.map(lambda worker: _get_result_pair(worker)).cache()
        result_list = self._rddWorkerList.map(lambda pair: pair[1]).collect()
        self._rddWorkerList = self._rddWorkerList.map(lambda pair: pair[0])

        all_results = None
        if len(result_list):
            all_results = [item for sublist in result_list for item in sublist]

        end_time = time.time()

        self.waiting_time += (end_time - start_time)

        if all_results is not None and len(all_results) > 0:
            for task in all_results:
                if task['id'] not in self._computed_tasks:
                    self._results_waiting.append(task)
                    self._computed_tasks.append(task['id'])
                else:
                    # TODO: log as warning
                    print("[SolverManager_PHSpark] Got repeated task from worker: %s " % task)
```

Figura 5.8: Método `_peform_wait_any` del prototipo funcional.

Capítulo 6

Plan de pruebas

6.1. Pruebas funcionales

6.1.1. Introducción

El actual plan de pruebas tiene como objetivo asegurar que los requisitos especificados son cumplidos satisfactoriamente por el módulo de código implementado. Adicionalmente se buscará que el código implementado funcione correctamente cuando dispone de entradas incorrecta.

Este plan de pruebas, así como todos los creados como parte de este proyecto, está basado en el estándar IEEE 829 [4].

6.1.2. Restricciones

La principal restricción de este plan de pruebas es el tiempo. Sólo se dispone de 1 día para la creación del plan y otro día para su ejecución y la recolección de resultados.

La complejidad de las entradas que recibe el módulo limita el alcance de las pruebas de caja negra que podremos diseñar en el tiempo disponible.

6.1.3. Objeto de pruebas

El código a probar será la implementación del gestor de Spark, concretamente la clase *SolverManager_PHSpark*. Su ejecución se realiza mediante el comando *runph* con la opción *-solver-manager=phspark*.

Se realizarán pruebas de caja negra sobre las diferentes opciones de configuración del solver manager. Adicionalmente se ejecutarán los ejemplos existentes en el proyecto y se comparará la solución generada mediante spark y mediante el algoritmo secuencial.

6.1.4. Características a probar y exclusiones

Se probarán las diferentes opciones de configuración que modifican la ejecución de phspark. Con esto verificamos el requisito RF-01 y RF-02.

Adicionalmente se ejecutan los ejemplos existentes para PySP mediante phspark y la implementación secuencial, comparando los resultados. De esta forma se verifica un funcionamiento correcto en distintos escenarios así como el requisito RF-03.

Queda excluído de este plan de pruebas la generación de pruebas mediante estrategias de caja blanca. No podremos asegurar una cobertura óptima de código pero no es posible realizarlas en el tiempo disponible dada la complejidad de las entradas.

6.1.5. Estrategia

Se utilizará el módulo "unittest" de Python para generar una clase que ejecute los casos de prueba diseñados.

Será necesario generar un archivo de salida correcto para usar como punto de comparación con las ejecuciones de pruebas. Será necesario filtrar las diferencais en la salida que causa la ejecución con Spark o, alternativamente, comprobar manualmente el resultado de cada pareja de archivos.

6.1.6. Criterios de aceptación

Se considerará que una prueba ha sido pasada correctamente si, tras introducir los datos de entrada escogidos, la aplicación devuelve una salida que concuerda exactamente con la que se estableció como esperada.

Será posible concluir que la aplicación aprobada pasó las pruebas si el 100 % de los casos válidos pasaron los criterios de aceptación. En caso contrario será necesaria la corrección de la implementación.

6.1.7. Diseño de pruebas

Prueba P-01	Conexión a Spark
Objetivo	Valida la creación de una conexión a Spark.
Requisitos validados	RF-01, RF-02
Técnicas aplicadas	▪ **Por caja negra:** • host ◦ 1. Clase válida: Cadena que represente una IP correcta y disponible. ◦ 2. Clase válida: Null. ◦ 3. Clase no válida: Cadena que represente una IP errónea. ◦ 4. Clase no válida: Cadena que represente una IP correcta pero no disponible. • port ◦ 5. Clase válida: Entero que represente un puerto correcto y disponible. ◦ 6. Clase válida: Null. ◦ 7. Clase no válida: Entero que represente un puerto erróneo. ◦ 8. Clase no válida: Entero que represente un puerto correcto pero ocupado.
Casos de prueba	CP-01, CP-02, CP-03, CP-04, CP-05, CP-06
Salida esperada	Conexión correcta y ejecución que genere la salida esperada en el caso de las pruebas para clases válidas. Para las pruebas con entradas no válidas se debe mostrar un error especificando el problema.

Cuadro 6.1: P-01: Conexión a Spark

Prueba P-02	Ejecución correcta de ejemplos
Objetivo	Valida que los resultados devueltos son correctos.
Requisitos validados	RF-01, RF-02, RF-03
Técnicas aplicadas	Se ejecutarán todos los ejemplos disponibles en *"/pyomo/examples/pysp"* usando la versión secuencial para generar el resultado esperado y la versión Spark para verificar su funcionamiento.
Casos de prueba	CP-01, CP-02, CP-03
Salida esperada	Los resultados de todos los ejemplos son iguales para la versión secuencial y paralela.

Cuadro 6.2: P-02: Ejecución correcta de ejemplos

6.1.8. Casos de prueba

Caso de prueba	CP-01
Descripción	Prueba la entrada correcta
Clases que valida	P-01(1,5)
Necesidades del entorno	Instancia de Spark funcionando en "spark://localhost:7077"
Entrada	host = "localhost" port = 7077
Salida esperada	Igual a la ejecución con -solver-manager=serial

Cuadro 6.3: CP-01: Prueba la entrada correcta

Caso de prueba	CP-02
Descripción	Prueba la url por defecto
Clases que valida	P-01(2,6)
Necesidades del entorno	Instancia de Spark funcionando en "spark://localhost:7077"
Entrada	host = None port = None
Salida esperada	Igual a la ejecución con -solver-manager=serial

Cuadro 6.4: CP-02: Prueba la url por defecto

Caso de prueba	CP-03
Descripción	Prueba la gestión de error al intentar conectarse a una IP incorrecta
Clases que valida	P-01(3)
Necesidades del entorno	N/A
Entrada	host = "111.111" port = None
Salida esperada	Mensaje de error indicando que no se ha podido conectar a Spark en la URL especificada.

Cuadro 6.5: CP-03: Prueba una IP incorrecta

Caso de prueba	CP-04
Descripción	Prueba la gestión de error al intentar conectarse a una IP donde no se está ejecutando Spark
Clases que valida	P-01(4)
Necesidades del entorno	La URL "spark://localhost:7077" no debe tener ningún servicio escuchando.
Entrada	host = "localhost" port = 7077
Salida esperada	Mensaje de error indicando que no se ha podido conectar a Spark en la URL especificada.

Cuadro 6.6: CP-04: Prueba una IP sin Spark

Caso de prueba	CP-05
Descripción	Prueba la gestión de error al especificar un puerto inválido
Clases que valida	P-01(7)
Necesidades del entorno	N/A
Entrada	host = None port = -1
Salida esperada	Mensaje de error indicando que no se ha podido conectar a Spark en la URL especificada.

Cuadro 6.7: CP-05: Prueba un puerto inválido

Caso de prueba	CP-06
Descripción	Prueba la gestión de error al especificar un puerto donde no se está ejecutando Spark
Clases que valida	P-01(8)
Necesidades del entorno	La URL "spark://localhost:8080" no debe ser una instancia de Spark
Entrada	host = "localhost" port = 8080
Salida esperada	Mensaje de error indicando que no se ha podido conectar a Spark en la URL especificada.

Cuadro 6.8: CP-06: Prueba un puerto sin Spark

6.1.9. Procedimiento de pruebas

Los casos de prueba especificados son idempotentes y no es necesario especificar un orden concreto para su ejecución. Será necesario para cada uno establecer el entorno especificado en cada caso concreto. Adicionalmente, para verificar los resultados será necesario ejecutar la misma entrada de forma secuencial para comparar.

Para la prueba P-02 no se especifican casos de prueba concretos porque todos se ejecutan de la misma forma cambiando el directorio de entrada. Para esta prueba se ejecutarán todos los ejemplos disponibles para PySP de forma secuencial y, posteriormente, usando Spark. Si Spark devuelve siempre los mismos resultados (dentro de un margen de error) se considerará la prueba como superada.

6.1.10. Ejecución de las pruebas

Por limitaciones temporales, los tests unitarios no están totalmente automatizados. Actualmente, la arquitectura de tests necesita de la creación de archivos de referencia para contrastar con la salida de la ejecución de prueba. Adicionalmente, por las posibles diferencias entre ambos archivos no relevantes para el resultado (como mensajes informativos), se debe programar un filtro concreto para las diferentes salidas.

Por las razones expuestas los tests se programarán para ejecutarse automáticamente, pero se debe comprobar manualmente si la salida es la esperada. Para realizar esta comprobación se ejecutará el mismo comando que ejecuta cada test, pero utilizando *–solver-manager=serial*.

La solución de cada ejecución se guarda en un archivo *ph_solution.json*. Se guarda la solución de la ejecución secuencial y posteriormente es posible ejecutar *diff* con la solución devuelta por el test, siempre que se espere una salida correcta. En caso contrario se comprobará manualmente que la salida es la esperada.

ID	Salida	Superada
CP-01	TestPHFarmerSpark.test1.ph_solution.json.out	Si
CP-02	TestPHFarmerSpark.test2.ph_solution.json.out	Si
CP-03	RuntimeError: ERROR connecting with Spark at spark://111.111:7077. URL might be incorrect (org.apache.spark.SparkException)Invalid master URL: spark://111.111:7077	Si
CP-04	RuntimeError: ERROR connecting with Spark at spark://192.10.10.10:7077. Check if Spark is running on that IP. (java.lang.IllegalArgumentException)requirement failed: Can only call getServletHandlers on a running MetricsSystem	Si
CP-05	RuntimeError: ERROR connecting with Spark at spark://localhost:-1. URL might be incorrect (org.apache.spark.SparkException)Invalid master URL: spark://localhost:-1	Si
CP-06	RuntimeError: ERROR connecting with Spark at spark://localhost:8080. Check if Spark is running on that IP. (java.lang.IllegalArgumentException)requirement failed: Can only call getServletHandlers on a running MetricsSystem	Si

Cuadro 6.9: Resultados de la ejecución de los casos de prueba

A continuación se probará la ejecución de los ejemplos disponibles en el proyecto. Para ello se utilizará un script en bash que se mueva entre los directorios y ejecute la versión secuencial y paralela para cada ejemplo, imprimiendo los resultados de ejecutar el comando *diff*. Aunque con este script se busca automatizar al máximo la ejecución de pruebas, no se consigue totalmente. La naturaleza del algoritmo hace que existan ligeras diferencias en los valores generados. Esto causará que las salidas sean diferentes, pero no implica que los resultados sean incorrectos. Para facilitar la comprobación de esta situación, se utiliza la herramienta *colordiff* que resaltará las diferencias entre ambas soluciones.

A continuación se muestra el primer test del script donde se ejecuta el ejemplo “Farmer”. La ejecución de los ejemplos restantes es equivalente, modificando

únicamente las rutas de los archivos.

```
#!/bin/bash

GREEN='\032[0;31m'
RED='\033[0;31m'
NC='\033[0m'

# FARMER

echo "Creating baseline for Farmer"
cd ../../../pyomo/examples/pysp/farmer

PHHISTORYEXTENSION_USE_JSON=1 runph -r 1 --solver-manager=serial --traceback --solver=minos
    --solver-io=nl --xhat-method=voting --traceback --model-directory=models/
    ReferenceModel.py --instance-directory=scenariodata/ScenarioStructure.dat --
    user-defined-extension=pyomo.pysp.plugins.phhistoryextension --solution-writer=
    pyomo.pysp.plugins.jsonsolutionwriter &> ../../../../Documentación/Pruebas/Salidas\
    ejemplos/TestFarmerSeq.out

mv ph_solution.json ../../../../Documentación/Pruebas/Salidas\ ejemplos/
    TestFarmerSeq.ph_solution.json

echo "Executing Farmer with Spark"

PHHISTORYEXTENSION_USE_JSON=1 runph -r 1 --solver-manager=phspark --traceback --solver=minos
    --solver-io=nl --xhat-method=voting --traceback --model-directory=models/
    ReferenceModel.py --instance-directory=scenariodata/ScenarioStructure.dat --
    user-defined-extension=pyomo.pysp.plugins.phhistoryextension --solution-writer=
    pyomo.pysp.plugins.jsonsolutionwriter &> ../../../../Documentación/Pruebas/Salidas\
    ejemplos/TestFarmerPHSpark.out

mv ph_solution.json ../../../../Documentación/Pruebas/Salidas\ ejemplos/
    TestFarmerPHSpark.ph_solution.json

cd ../../../../Documentación/Pruebas/Salidas\ ejemplos

result=$(colordiff -y TestFarmerPHSpark.ph_solution.json TestFarmerSeq.ph_solution.json)

if [ $? -eq 0 ]
then
    printf "${GREEN}------------------${NC}\n"
        printf "${GREEN}Farmer test passed${NC}\n"
    printf "${GREEN}------------------${NC}\n"
else
    printf "${RED}------------------${NC}\n"
        printf "${RED}Farmer test failed${NC}\n"
        printf "$result\n"
    printf "${RED}------------------${NC}\n"
fi
```

Figura 6.1: Script de pruebas

El resultado de ejecutar este script completo se muestra en la tabla a continuación:

Farmer	passed
Baa99	failed (stackoverflow)
Farmer_generated	failed (stackoverflow)
FarmerWIntegers	passed
FarmerWPieceWise	passed
FarmerWrent	passed
Finance	failed (stackoverflow)
Hydro	passed

Cuadro 6.10: Resultados de la ejecución de los ejemplos

Podemos observar que algunos tests aparecen como fallidos con un error *stackoverflow*. Esto no significa necesariamente que el programa no funcione como se espera, lo más probable es que se deba a limitaciones de la máquina en la que se han realizado las pruebas.

Queda como tarea en un futuro optimizar la gestión de la memoria y el uso de Spark para que ejecutar problemas como este sea posible en máquinas con menos recursos.

6.2. Pruebas de rendimiento

6.2.1. Introducción

Este segundo plan de pruebas se centrará en evaluar el rendimiento de la implementación con Spark. El objetivo es establecer una medición concreta comparando la ejecución en Spark con la ejecución secuencial y con Pyro.

6.2.2. Restricciones

Existen dos restricciones principales sobre la realización de este plan de pruebas. En primer lugar, el tiempo disponible. Se disponen de 3 días para elaborar, ejecutar y analizar las pruebas expuestas en esta sección.

Adicionalmente, las pruebas no se ejecutarán en un cluster de computación, lo que limitará el tamaño de los problemas a probar y la escalabilidad que podremos observar. La máquina de pruebas será un portátil ejecutando una máquina virtual con las siguientes características:

- Fedora 27 (Workstation Edition)
- Procesador Intel Core i7 7700HQ
- Memoria Principal 4GB

6.2.3. Objeto de pruebas

Durante la ejecución del primer plan de pruebas se realizaron mediciones en el tiempo de ejecución de los ejemplos existentes como parte de Pyomo. Con estos valores iniciales decidimos que se utilizará el ejemplo "Hydro" para esta evaluación de rendimiento. Este ejemplo nos proporciona un problema con 9 escenarios, lo que debería favorecer la escalabilidad en paralelo, y se puede ejecutar en un tiempo razonable en nuestra máquina de pruebas.

6.2.4. Características a probar y exclusiones

Se medirá el tiempo de ejecución del ejemplo especificado utilizando diferentes configuraciones hardware, usando desde 1 núcleo hasta 8 para Spark y con variaciones en la memoria disponible.

6.2.5. Estrategia

Se utilizará la opción '*–output-times*' que utiliza la librería "time" de Python para la medición de tiempos. Los valores de las diferentes ejecuciones se recopilarán en una hoja de cálculo para generar gráficas que faciliten el estudio de los resultados.

6.2.6. Criterios de aceptación

Las pruebas se considerarán satisfactorias si producen resultados estudiables. Dichos resultados deben ser replicables utilizando una configuración similar. Deben evitarse resultados que pueden haber sido alterados por condiciones externas durante la ejecución del programa, por lo que deberían realizarse múltiples ejecuciones iguales para filtrar los resultados coherentes.

6.2.7. Casos de prueba

En este plan de pruebas no definiremos casos de prueba siguiendo el estándar. A continuación se listan las diferentes configuraciones de núcleos y memoria con las que realizará la ejecución.

Hilos	Memoria
1	2GB
2	2GB
3	2GB
4	2GB
5	2GB
6	2GB
7	2GB
8	2GB
1	4GB
2	4GB
3	4GB
4	4GB
5	4GB
6	4GB
7	4GB
8	4GB

Cuadro 6.11: Tabla de pruebas de rendimiento

6.2.8. Procedimiento de pruebas

Utilizando las configuraciones anteriores se ejecutará cada una de ellas 3 veces y se guardará una media de los tiempos.

Se establecerá también la media de duración por iteración para observar de forma más concreta el overhead causado por la nueva implementación.

6.2.9. Ejecución de las pruebas

El primer aspecto a probar es el tiempo de ejecución del ejemplo *Hydro*. Este modelo especifica 9 escenarios diferentes que, teniendo en cuenta que la paralelización se realiza a nivel de escenario, es esperable una buena escalabilidad.

Sin embargo, los resultados obtenidos no cumplen lo esperado:

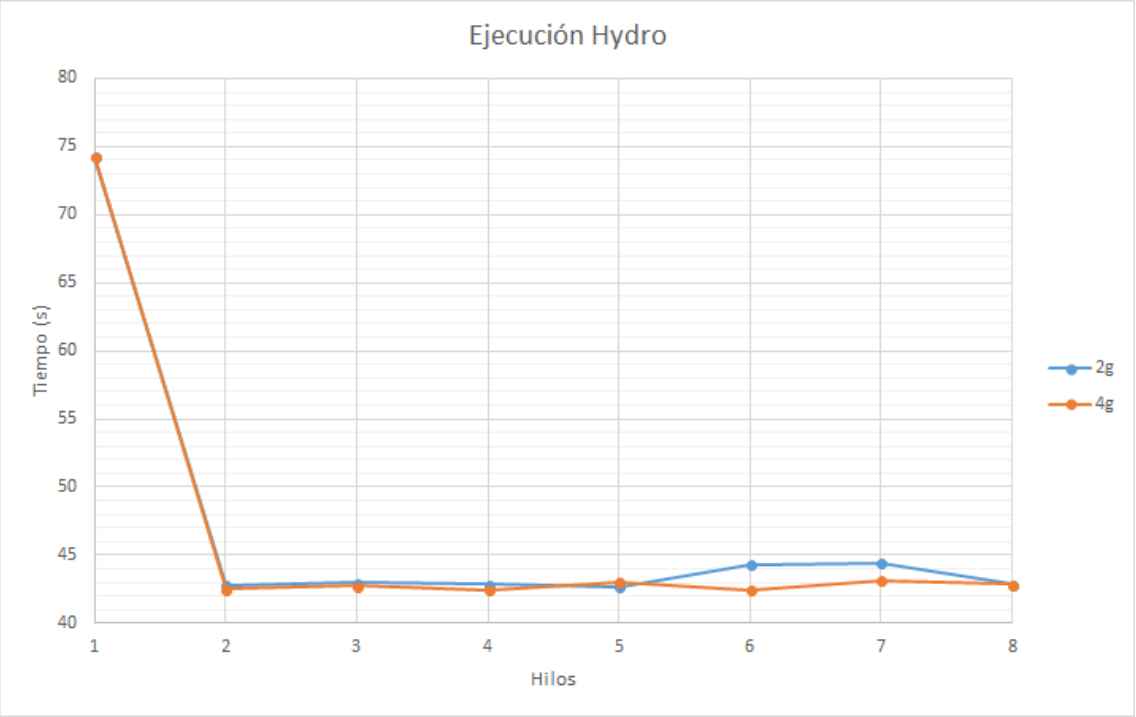


Figura 6.2: Tiempos de ejecución Hydro

Para analizar estos datos, primero debemos tener en cuenta dos puntos importantes: la ejecución no se ha realizado en una máquina adecuada para aprovechar la paralelización y el tiempo de ejecución de la versión secuencia es de aproximadamente 6s.

A parte de ejecutar las pruebas con distinto número de hilos, también se ha probado a limitar la memoria disponible en Spark a 2GB o 4GB. Esto está representado en la gráfica como las líneas naranja y azul. Podemos concluir facilmente que, para este ejemplo concreto, no existe diferencia en la memoria disponible. Esto puede ser porque la memoria necesaria es siempre inferior a 2GB o es siempre superior a 4GB.

En cuanto a los tiempos de ejecución, si tenemos en cuenta que la ejecución normal tarda unos 6s, la implementación con Spark utilizando 1 hilo tarda 74s, siendo una opción claramente peor a nivel temporal.

Aumentando el número de hilos disponible, observamos inicialmente una mejora importante. Pasando de 1 a 2 hilos reducimos el tiempo casi a la mitad (0,57x). Esto es una prueba de la escalabilidad que promete Spark. Sin embargo, aumentar el número de hilos no resulta en ningún tipo de mejora, a pesar de que esperábamos mejoras con la utilización de hasta 9 hilos (por existir 9 escenarios).

Esto puede ser el resultado de ejecutar un programa demasiado pequeño en una plataforma poco adecuada.

Estos tiempos están claramente lejos de lo conseguido con la ejecución secuencial, pero son un claro indicador del potencial de una implementación con Spark para la resolución de problemas estocásticos y, con futuras optimizaciones y un cluster donde ejecutarse, esta implementación podría ser una alternativa viable para abordar problemas que antes no eran resolubles en un tiempo razonable.

Con el objetivo de analizar los tiempos sobre un problema mayor se utilizarán los tiempos establecidos por el ejemplo *Finance*. Este ejemplo no consigue finalizar por lo que parecen ser limitaciones de memoria, pero sí podemos observar los tiempos de ejecución de las primeras iteraciones. Se seleccionan los tiempos de ejecución de las primeras 10 iteraciones y se hace una media de los mismos. En la siguiente figura se muestra el tiempo medio por iteración para el ejemplo *Finance* así como para el anterior ejemplo *Hydro* utilizando 1,2 y 8 hilos.

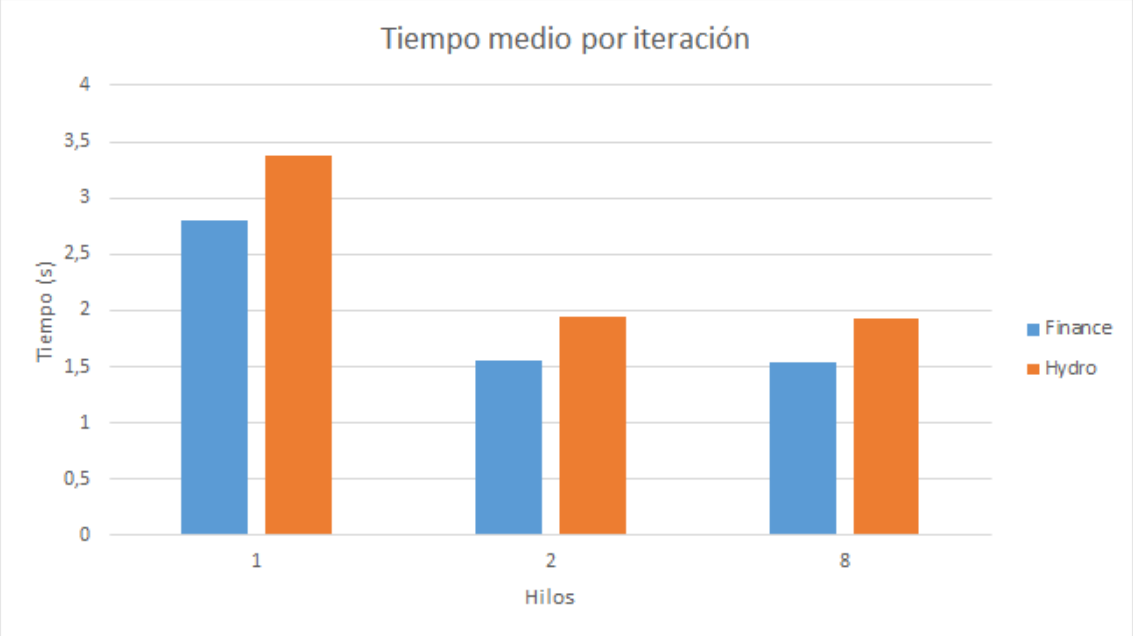


Figura 6.3: Tiempo medio por iteración

Los resultados, de nuevo, no son los esperados. El ejemplo *Finance* no falla porque el modelo a procesar sea demasiado grande, si no porque necesita una mayor cantidad de datos durante más iteraciones.

En esta prueba observamos un comportamiento similar a lo visto en la anterior

gráfica. En primer lugar, el ejemplo *Finance* se ejecuta más rápido que *Hydro* y conseguimos una mejora del 0,55x y 0,57x respectivamente.

Capítulo 7

Conclusiones

En este proyecto hemos visto un uso muy interesante para las tecnologías Big Data. En principio, cuando pensamos en Big Data, pensamos en una gran cantidad de datos como una red de sensores o datos de usuarios en internet que son procesados para realizar análisis estadísticos, por ejemplo. Este trabajo muestra las virtudes del framework Spark como herramienta para ejecutar trabajos de forma paralela. Hemos demostrado con un ejemplo práctico cómo Spark puede soportar una arquitectura de objetos distribuidos para solucionar problemas en el ámbito científico.

Este modelo de computación distribuida supone múltiples ventajas con respecto a utilizar otras herramientas como MPI u OpenMP. La principal ventaja es su sencillez. El framework Spark facilita la gestión del clúster de computación y se encarga de la repartición de datos y la distribución de la carga computacional. Esto permite a los desarrolladores centrarse en el algoritmo concreto y no en los detalles específicos de trabajar en un entorno distribuido. Además, como el clúster y la implementación están desacopladas, se pueden mejorar y modificar de forma aislada, pudiendo aprovechar nuevas versiones de Spark y hardware más potente sin hacer ninguna o pocas modificaciones sobre la implementación.

Otro logro importante es la integración con Pyomo. Aunque se deben mejorar algunas cuestiones menores, el nuevo módulo de trabajos con Spark se integra perfectamente con el resto del proyecto, funcionando con sólo modificar una opción de ejecución. Esto abre las puertas a, en un futuro, añadir esta funcionalidad a la versión estable de Pyomo para que se pueda utilizar libremente y, al ser un proyecto de código abierto, otros grupos de investigación podrán añadir mejoras al mismo.

En cuanto al rendimiento, los valores de escalabilidad prometen ofrecer un buen rendimiento dados un hardware y problemas de entrada adecuados. Podemos observar una mejora superior al 0,5x pasando de 1 a 2 hilos de ejecución. En las pruebas que se pudieron ejecutar, no es posible mejorar estos números con

más hilos, probablemente por los problemas ejecutados y la máquina de pruebas. Sin embargo, con alguna optimización a mayores, es posible que consigamos una escalabilidad casi perfecta con respecto al número de nodos de procesamiento. Esto implicaría tener una implementación muy eficiente que permitiría abordar grandes problemas estocásticos en poco tiempo haciendo uso de un clúster de computación adecuado.

7.1. Lecciones aprendidas

El desarrollo de este trabajo ha requerido cambios en la planificación, llegando a ser necesario un mes extra de trabajo. Incluso con este tiempo extra, no ha sido posible centrarse en algunos puntos puntos de interés del trabajo, principalmente el periodo de optimización en el que tendría que aprender y aprovechar las características más concretas de Spark.

En un proyecto de esta complejidad y duración, la importancia de la gestión se hace más aparente. Desde la realización del anteproyecto, la planificación temporal del proyecto se realizó sin utilizar ningún método de estimación ni datos concretos. A pesar de tener que realizar un proyecto utilizando tecnologías desconocidas (tanto Python como Spark), condiciones muy relevantes para considerar un ciclo en cascada como inadecuado, se decidió planificar el proyecto en cascada porque era más sencillo organizar el tiempo. Se esperaba que, en el caso de retrasarse en una tarea, simplemente se trabajaría más horas y eso sería suficiente. Después de los primeros retrasos sí se modifica el ciclo de vida tomando en consideración estas limitaciones, pero esto es a costa de eliminar la fase de diseño. Para la solución que se implementó finalmente, el diseño no es una fase demasiado importante y creo que se ejecutó satisfactoriamente.

El otro pilar importante de este proyecto es Spark. Adaptar una arquitectura orientada a objetos para funcionar en Spark no es una solución óptima, como podemos comprobar por la cantidad de errores encontrados así como los números de rendimiento finales. Viendo la documentación de Spark, está claro que el caso de uso óptimo es el transformar datos estáticos y ejecutar operaciones sobre ellos. En nuestro caso estamos paralelizando una jerarquía completa de objetos, con multitud de dependencias y estos objetos incluso interactúan con archivos y programas externos. El simple hecho de que esta implementación funcione correctamente demuestra la versatilidad de este framework.

Cuando se encuentran problemas en el código que necesitan múltiples horas de pruebas y búsqueda de errores, es muy útil hacer un seguimiento por escrito de las tareas realizadas. En este caso se utilizaron múltiples herramientas que facilitaron en gran medida la realización del proyecto así como de la memoria

final:

- **Trello:** Es una herramienta web que permite gestionar paneles con listas de tareas. Con esto es sencillo hacer un seguimiento de las tareas pendientes, las que se están ejecutando actualmente y del trabajo ya finalizado.
- **Git:** Un sistema de control de versiones es crucial cuando se están buscando problemas en el código. Permite modificar secciones enteras de código, introducir pruebas, etc, con la tranquilidad de siempre poder volver a una versión correcta. Además nos da una lista de todos los cambios que hemos hecho con lo que podemos identificar dónde están los errores y cómo se han solucionado. Por último, es muy útil poder cambiar entre versiones del código para realizar múltiples pruebas.
- **Informes de seguimiento:** Al principio del proyecto se decidió realizar documentos de progreso cada poco tiempo. Estos archivos no sólo son útiles para documentar el proyecto, también sirven como puntos de reflexión. Generar un informe de progreso nos fuerza a resumir todo lo hecho hasta ahora y a tener claros los objetivos que se deben perseguir a continuación.

7.2. Trabajo futuro

Aunque el desarrollo sufrió múltiples problemas en su realización, el resultado final supone una buena base para una implementación útil de Spark en Pyomo. En su estado actual no provee ninguna ventaja con respecto a las implementaciones existentes pero tiene potencial para ser una alternativa muy eficaz.

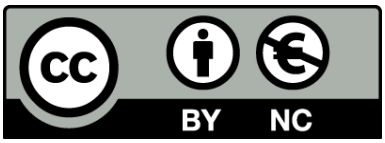
Partiendo de esta base, hay dos caminos claros a seguir para generar una implementación que ofrezca un buen rendimiento. Podemos mantener la arquitectura orientada a objetos actual y centrarnos en optimizar al gestión de las tareas para que las transformaciones ejecutadas en Spark sean menos y contengan mayor trabajo. Por otro lado, podemos simplificar el trabajo que debe realizarse dentro de Spark para acercarse más al tipo de operaciones que funcionan de mejor en Spark.

Todos estos cambios deben apoyarse en una batería de pruebas utilizando un cluster de computación que aproveche al máximo las ventajas que nos ofrece Spark. Los problemas a probar deben tener la complejidad suficiente para visualizar la utilidad de un entorno distribuido.

Si se realizan todos estos cambios, el objetivo final, dado que Pyomo es un proyecto abierto, sería realizar un *pull-request* para que cualquiera pueda aprovechar estas ventajas utilizando la versión estable de Pyomo.

Apéndice A

Licencia



Bibliografía

[1] Matei Zaharia, Patrick Wendell, Andy Konwinski, Holden Karau, *Learning Spark: Lightning-Fast Big Data Analysis*, 1ª edición, O'Reilly Media, 2015.

[2] John R. Birge, François Louveaux, *Introduction to Stochastic Programming*, 2ª Edición, Springer, 2011.

[3] Watson, Jean-Paul and Woodruff, David L. and Strip, David R., *Progressive Hedging Innovations for a Class of Stochastic Resource Allocation Problems* (September 15, 2008). UC Davis Graduate School of Management Research Paper No. 05-08. Available at SSRN: `https://ssrn.com/abstract=1268385` or `http://dx.doi.org/10.2139/ssrn.1268385`. Consultado 17 de julio 2018.

[4] IEEE Std 829-2008, IEEE Standard for Software and System Test Documentation

[5] Pyomo `http://www.pyomo.org/about/`. Consultado 17 de julio 2018.

[6] Andrés Guillermo Iroume Awe, *PROGRESSIVE HEDGING APLICADO A COORDINACION HIDROTERMICA*, Marzo 2012, Disponible en `http://repositorio.uchile.cl/bitstream/handle/2250/114109/cf-iroume_aa.pdf;sequence=1`. Consultado 17 de julio 2018.

[7] Pyro. `https://pythonhosted.org/Pyro4/`. Consultado 17 de julio 2018.

[8] Big Data. Wikipedia (`https://en.wikipedia.org/wiki/Big_data`). Consultado 17 de julio 2018.

[9] AMPL. Wikipedia (`https://en.wikipedia.org/wiki/AMPL`). Consultado 18 de julio 2018.

[10] IBM ILOG CPLEX Optimization Studio. Wikipedia (`https://en.wikipedia.org/wiki/CPLEX`). Consultado 22 de julio 2018.

[11] GNU Linear Programming Kit (GLPK). Wikipedia (`https://en.wikipedia.org/wiki/GNU_Linear_Programming_Kit`). Consultado 22 de julio 2018.

[12] Minos (*Modular In-core Nonlinear Optimization System*). Wikipedia (`https://en.wikipedia.org/wiki/MINOS_(optimization_software)`). Consultado 22 de julio 2018.

[13] Apache Spark. `http://spark.apache.org/`. Consultado 22 de julio 2018.

[14] *Message Passing Interface*. Wikipedia (`https://en.wikipedia.org/wiki/Message_Passing_Interface`). Consultado 22 de julio 2018.

[15] Funcionamiento PH en "/Documentación/Análisis/Funcionamiento PH paralelo.pdf"

[16] Proyecto de pruebas en: "/DistributedTest"

[17] Documentación de PySP en "/pyomo/doc/pysp/pyspdoc.pdf"