UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA INDUSTRIAL

EVALUACIÓN DE LA RESOLUCIÓN EN PARALELO DE UN PROBLEMA ESTOCÁSTICO DE PLANIFICACIÓN MINERA DE LARGO PLAZO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL INDUSTRIAL

JORGE ALARCÓN DÍAZ

SANTIAGO DE CHILE
JUNIO 2012

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA INDUSTRIAL

EVALUACIÓN DE LA RESOLUCIÓN EN PARALELO DE UN PROBLEMA ESTOCÁSTICO DE PLANIFICACIÓN MINERA DE LARGO PLAZO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL INDUSTRIAL

JORGE ALARCÓN DÍAZ

PROFESOR GUÍA:
RODOLFO URRUTIA URIBE

MIEMBROS DE LA COMISIÓN:
RAFAEL EPSTEIN NUMHAUSER
PATRICIO CONCA KEHL

SANTIAGO DE CHILE
JUNIO 2012

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INDUSTRIAL
POR: JORGE ALARCÓN DÍAZ
FECHA: 12/06/2012
PROF. GUÍA: SR. RODOLFO URRUTIA

EVALUACIÓN DE LA RESOLUCIÓN EN PARALELO DE UN PROBLEMA ESTOCÁSTICO DE PLANIFICACIÓN MINERA DE LARGO PLAZO

La minería, que alcanza hoy en día casi un 20 % de participación en el PIB nacional, corresponde a la principal actividad económica que ha tenido Chile desde la revolución industrial. Dentro de este contexto destaca la empresa estatal CODELCO como la mayor productora de cobre a nivel mundial.

Considerando el impacto que tiene la industria minera, los altísimos niveles de inversión involucrados, la larga vida útil de una mina y su complejidad, el apoyo a las decisiones mediante modelos de optimización matemáticos aparece como altamente necesario. Hoy en día estos modelos son capaces de planificar tanto la extracción como el procesamiento del mineral considerando sólo variables determinísticas.

Una variable muy importante dentro de estos modelos corresponde al precio del cobre, cuya aleatoriedad la hace dificil de predecir. Es por ello que actualmente se trabaja en la implementación de modelos de planificación de largo plazo que incorporen la estocasticidad de esta variable con el fin de evaluar proyectos mineros y así dar apoyo a las decisiones de inversión.

Sin embargo, y debido a la complejidad y tamaño de los procesos, el problema estocástico es demasiado grande desde el punto de vista computacional. Para resolver esto se utiliza Progressive Hedging (PH), un algoritmo que reduce el problema estocástico en muchos sub-problemas determinísticos de menor tamaño. A pesar de esto, el tiempo necesario para su resolución puede llegar a ser del orden de días o incluso semanas.

Una de las cualidades que entrega el algoritmo PH es la independencia en la resolución de los sub-problemas, lo cual puede ser aprovechado para su procesamiento en paralelo de modo de reducir los tiempos de procesamiento. En este trabajo se estudia la complejidad y factores críticos en la implementación de la paralelización, además de la ganancia esperada en súper-computadores tipo clúster.

El análisis algorítmico del problema da luces de que la paralelización es capaz de entregar ahorros en tiempo del orden de un 90 %. Por otra parte una simulación en la escalabilidad de una pequeña implementación realizada entrega un ahorro de entre 93 % y 98 % dependiendo de la arquitectura de memoria del computador, compartida o distribuída, compuesto de 9 procesadores con 8 núcleos cada uno.

Considerando todo lo anterior, y lo que se puede observar con más detalle en este trabajo, la utilización de PH en paralelo es un camino tentador a seguir si lo que se pretende es incluir la estocasticidad de una variable como el precio del cobre en el modelo de planificación minera de largo plazo.

A mis padres, Patricia y Jorge.

A mis hermanas, Natalia y Daniela.

A mi abuelita Lila.

A mi Karen.

A mis amigos, Sebastian y Felipe.

Índice de contenidos

1. Introducción **1**
1.1. Estructura del documento 3

2. Objetivos **4**
2.1. Objetivo General 4
2.2. Objetivos Específicos 4
2.3. Justificación 5
2.4. Alcances 7

3. Marco Teórico **8**
3.1. Árboles y Escenarios 8
3.2. No-anticipatividad 9
3.3. Progressive Hedging 10
3.4. Computación en paralelo 10
3.4.1. Conceptos 11
3.4.2. Nivel de paralelización 13
3.4.3. Coordinación 14
3.4.4. Arquitectura de memoria 15
3.4.5. Consideraciones al paralelizar 17
3.4.6. Teoría de paralelización 19

4. El Problema Minero **22**
4.1. Descripción operacional de una mina 22
4.1.1. La extracción 22
4.1.2. La red de procesamiento 24
4.2. Modelo determinista 25
4.3. Modelo estocástico 25
4.4. Aplicación de PH al modelo 27

5. Paralelización de PH **28**
5.1. Métodos simples 28
5.1.1. Método síncrono 29

5.1.2. Método asíncrono . . . 29
5.2. Métodos de bloques-cíclico . . . 30
5.2.1. Método de bloque-cíclico ingenuo . . . 30
5.2.2. Método de bloque-cíclico con los escenarios de menor convergencia 30
5.3. Método del nodo de menor convergencia . . . 31
5.4. Método de los escenarios que no han convergido . . . 32
5.5. Método completamente asíncrono . . . 32
5.6. Método combinado . . . 32
5.7. Mejora en el tiempo . . . 33
5.7.1. Nomenclatura . . . 34
5.7.2. Inicialización del modelo determinista . . . 34
5.7.3. Iteración inicial PH . . . 35
5.7.4. Preparación PH . . . 37
5.7.5. Iteraciones PH . . . 37
5.7.6. Resultado . . . 40
5.7.7. Análisis . . . 40

6. Implementación **42**
6.1. GAMS . . . 42
6.1.1. Principales características . . . 42
6.2. Paralelización . . . 43
6.2.1. Paralelización GAMS . . . 44
6.2.2. Paralelización Java . . . 47
6.3. Resultados numéricos . . . 49
6.3.1. GAMS vs Java . . . 49
6.3.2. La función de overhead . . . 53
6.4. Simulación de escenarios . . . 55
6.4.1. Índices a medir . . . 55
6.4.2. Parámetros de entrada . . . 55
6.4.3. Casos de estudio . . . 56
6.4.4. Procedimiento . . . 57
6.4.5. Resultados . . . 58

7. Conclusiones **64**
7.1. Trabajo futuro . . . 66

8. Bibliografía **67**

Anexo A. Modelo matemático de planificación minera de largo plazo **71**
A.1. Conjuntos . . . 71
A.2. Conjuntos adicionales . . . 71
A.3. Parámetros . . . 72
A.4. Variables de decisión . . . 72

A.5. Restricciones 73
A.5.1. Extracción 73
A.5.2. Disponibilidad de nodos y arcos 73
A.5.3. Conservación de flujos 73
A.5.4. Capacidades 74
A.5.5. Uso de equipos 74
A.5.6. Contaminantes 75
A.6. Función objetivo 75
A.6.1. Costos de inversión 75
A.6.2. Costos fijos 75
A.6.3. Costos variables 75
A.6.4. Beneficios 75
A.6.5. Función objetivo 75
A.7. Naturaleza de las variables 76

Anexo B. Resultados de la simulación para el caso 200-escenarios **77**

Índice de tablas

2.1. Algoritmo PH con 32 escenarios de precios diferentes en un proyecto real 6

6.1. Overhead porcentual por escenario con Java paralelo 53
6.2. Curvas de tendencia del overhead con Java Paralelo 55
6.3. Casos de estudio 56
6.4. Ahorros obtenidos en la simulación del caso base 58
6.5. Comparación del ahorro obtenido para casos con sensibilidad en la desviación estándar 61
6.6. Comparación del ahorro obtenido para casos con sensibilidad en la curva de overhead 62

Índice de figuras

2.1. Precio histórico del Cobre . . . 5
2.2. 32 series de precios . . . 6

3.1. Árbol de escenarios . . . 9
3.2. El sistema operativo como puente en un computador . . . 11
3.3. Diagrama de estados de un proceso . . . 13
3.4. Arquitecturas de memoria compartida . . . 16
3.5. Arquitectura de memoria distribuida . . . 17
3.6. Carga desbalanceada entre los distintos procesadores . . . 18
3.7. Esquema de granulidad . . . 19
3.8. Ley de Amdahl . . . 20
3.9. Ley de Gustafson . . . 21

4.1. Diseño de expansiones de una mina . . . 23
4.2. Visión esquemática de una mina a cielo abierto y sus expansiones . . . 23
4.3. Esquema de la red de procesamiento . . . 24
4.4. Árbol de decisiones . . . 26

5.1. Método síncrono y asíncrono . . . 29
5.2. Método del nodo de menor convergencia . . . 31
5.3. Sistema clúster con el que se cuenta . . . 33

6.1. Comparación en los tiempos de resolución de cada escenario según el programa utilizado . . . 50
6.2. Comparación en los tiempos de resolución de la iteración inicial PH según el programa utilizado . . . 52
6.3. Overhead porcentual promedio con Java paralelo . . . 54
6.4. Resultados de la simulación del caso base . . . 59
6.5. Resultados de la simulación del caso 100-escenarios . . . 60
6.6. Sensibilidad en la curva de overhead . . . 61
6.7. Resultados de la simulación con sensibilidad en la curva de overhead . . . 63

B.1. Resultados de la simulación del caso 200-escenarios . . . 77

Capítulo 1

Introducción

Desde el siglo XIX, tras la independencia de España y contemporánea a la revolución industrial, la minería comenzó a tomar vuelo como sector productivo clave, comenzando con la extracción de la plata y carbón, luego al salitre, para finalmente llegar al punto actual en donde Chile es el mayor productor de cobre a nivel mundial, alcanzando 5,4 millones de toneladas en el año 2009, lo que equivale al 34 % de la producción del globo [8]. Más aún, el aporte de la minería en el Producto Intero Bruto (PIB) del país alcanzó el 19,2 % en el año 2010 [13].

Dentro del mercado nacional destaca la Corporación Nacional del Cobre de Chile CODELCO, empresa autónoma propiedad del estado chileno, la cual corresponde a la mayor productora de cobre en el mundo, alcanzando una producción de 1,7 millones de toneladas el 2009 [7], lo que equivale al 32 % de la producción nacional para dicho año.

Una particularidad del mercado minero, a diferencia de otros mercados, es que la vida útil de las minas puede llegar a ser de varias decenas de año. Es por esto que se elabora un plan minero para toda la vida de una mina (life of mine), el cual contempla todos los recursos minerales (reservas probadas y probables) detectados por medio de sondajes, análisis, interpretación y modelamiento geo-minero-metalúrgico.

El resultado del plan minero consiste en la planificación de extracción de minerales desde la mina, transporte y almacenamiento de material, procesos de concentración, preparación y procesamiento del mineral, en donde las principales decisiones responden las interrogantes de cuánto invertir y cuándo, teniendo en consideración qué material se extraerá, a dónde se enviará y cuánto, con qué plantas se procesará y cuándo se hará (qué año).

Esta planificación a largo plazo, como se puede prever, resulta una tarea compleja debido a la cantidad de decisiones y consideraciones que se deben tener en cuenta. Es por esto que ya desde hace varios años se trabaja con métodos computacionales como

medio de apoyo a estas decisiones mediante formulaciones matemáticas de problemas de optimización. Un ejemplo de tales modelos se encuentra en [19].

Un resultado de esta planificación se puede apreciar en el estudio de pre-factibilidad del proyecto Mina Ministro Hales (MMH), el cuál considera una inversión de US$ 2.016 millones y entraría en funcionamiento el último trimestre de 2013, considerando una vida útil superior a los 50 años [16].

Una variable muy importante dentro del modelo corresponde al precio del mineral. Su principal problema es que es una variable difícil de predecir, tal como se puede apreciar en la figura 2.1, y cuya variación es fuertemente influyente en la rentabilidad de los proyectos mineros. Sólo en los primeros nueve meses de 2011 Codelco obtuvo un excedente 37 % mayor al mismo periodo del año anterior, en donde la producción varió un 3,4 % y el precio promedio del cobre varió un 29 % [23].

Debido a la magnitud de las inversiones, la larga vida útil de una mina y lo importante que es el precio del cobre en el modelo, es que últimamente se está trabajando en incorporar la incertidumbre en el precio de manera estocástica y mediante muchas series de precio representativas de la distribución de probabilidad que se espera tenga la variable del precio. Sin embargo, y tal como se puede prever, el tamaño del problema matemático crece considerablemente, lo cual hace que simplemente ajustar el modelo anterior a su versión estocástica sea inmanejable para la tecnología existente actualmente.

Es por ello que se propuso trabajar el problema por medio de una técnica de descomposición para resolver el modelo estocástico. Esta técnica descompone el problema global en muchos escenarios deterministas para luego mezclarlos y converger a la solución óptima del problema global. Este algoritmo a utilizar se llama Progressive Hedging (PH), y fue publicada por Roger J-B Wets y R.T. Rockafellar en el año 1991 [32].

Aunque esta metodología ya lleva 2 décadas desde su publicación, aún es difícil su aplicación debido a los altos tiempos que demora en procesar un modelo del tamaño que aquí se plantea. Por otra parte, una de las cualidades y ventajas que posee el algoritmo es que éste resuelve cada escenario como un problema determinístico, lo que implica que entre escenarios su resolución se realiza de manera independiente para luego forzar la convergencia entre los escenarios. Esta independencia entre escenarios es lo que hace atractivo resolver el problema de forma paralela.

Es por eso que en este trabajo se propone evaluar la viabilidad, factibilidad y eficiencia, desde el punto de vista de tiempo de resolución, de la resolución en paralelo de la formulación estocástica del problema minero utilizando PH en sistemas de alto rendimiento. Más específicamente se pretende contrastar la ganancia y los costos asociados a la paralelización, estimar ganancias en terminos de tiempo, detectar factores claves para llevar a cabo su implementación y prever sus dificultades.

Se propone adicionalmente una metodología simple de paralelización, la cual utiliza GAMS como lenguaje de modelación, CPLEX como solver y JAVA como lenguaje de programación adicional para la utilización de distintos métodos y funciones específicas para paralelizar.

1.1. Estructura del documento

En el capítulo 2 se establecen los objetivos de este trabajo, además de su justificación. En el capítulo 3 se revisan los conceptos a utilizar en el trabajo de modo de familiarizarse con la optimización estocástica y la paralelización. En el capítulo 4 se presenta el problema minero, su modelación determinista, estocástica y la aplicación de PH a éste. En el capítulo 5 se discuten los distintos métodos de paralelización para el algoritmo PH y se calcula la ganancia que se obtiene al paralelizar el algoritmo. Finalmente en el capítulo 6 se comentan los resultados obtenidos con una implementación simple de la paralelización de PH para el problema minero.

Capítulo 2

Objetivos

El foco de este trabajo consiste en explorar la posibilidad de paralelizar el algoritmo PH utilizado en el problema estocástico de planificación minera de largo plazo. Además se discuten los distintos métodos de paralelización y se realiza una pequeña implementación de una de estas metodologías.

2.1. Objetivo General

Evaluar el impacto que tiene el resolver en paralelo instancias de la gran minería del cobre para el problema de planificación minera de largo plazo bajo incertidumbre en el precio del cobre.

2.2. Objetivos Específicos

- Análisis algorítmico de la metodología actual (PH secuencial).
- Evaluación de los distintos métodos de paralelización para el algoritmo de PH.
- Estudio analítico de la mejora en el tiempo de resolución del algoritmo en paralelo respecto al algoritmo secuencial.
- Determinación de factores claves y mayores dificultades en la implementación de la paralelización del algoritmo PH.
- Validación del estudio analítico mediante una implementación simple de paralelización del algoritmo de optimización.

2.3. Justificación

El precio del cobre, tal y como se puede apreciar en la figura 2.1, corresponde a una variable de difícil predicción. Debido al crecimiento de los países asiáticos y la especulación de los actores del mercado, no es posible definir un patrón claro o tendencia en la trayectoria.

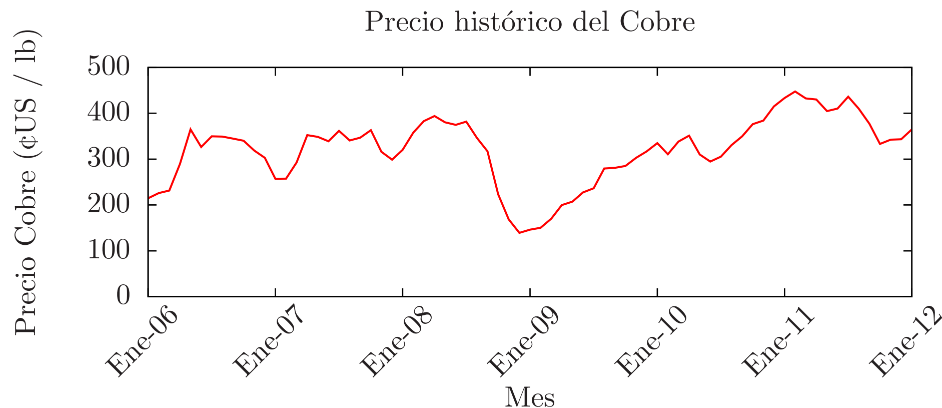


Figura 2.1: Precio histórico del Cobre de los últimos 5 años, en centavos de dólar la libra.

Entre Enero y Septiembre del 2011 Codelco registró un alza de un 37 % en sus utilidades respecto al mismo periodo del año anterior, debido principalmente al aumento en el precio del cobre, en donde el promedio estuvo en un 29 % por encima del promedio del mismo periodo del año anterior. Así, las utilidades subieron desde US$ 3.877 millones hasta los US$ 5.301 millones, la producción aumentó de 1.208 a 1.250 miles de toneladas de cobre fino, y el precio promedio pasó de 325,2 ¢U$/lb a 419,8 ¢U$/lb en dicho periodo.

Debido a esta variación y los efectos que produce en la evaluación de los proyectos mineros, parece necesario incorporar la incertidumbre en el precio del cobre a los modelos de planificación minera de largo plazo. Debido a los altos montos de inversión asociados, la decisión de un planificador de ejecutar o no un proyecto será notablemente distinta en caso de que el precio esté por sobre 500 ¢U$/lb que bajo 200 ¢U$/lb, ya que el desempeño del negocio es altamente sensible a la variable en cuestión.

A modo de ejemplo se muestra en la tabla 2.1 el Valor Presente Neto del negocio entregado por el algoritmo PH con datos de una mina real, la cantidad de material destinado a botadero y la cantidad de material destinado al proceso más caro, considerando 32 series de precios diferentes. El ESC1 considera la serie de precios optimista, el ESC32 la serie pesimista, y el resto de los escenarios se componen de series de precios que suben y bajan según el proceso descrito en la sección 3.1 del informe.

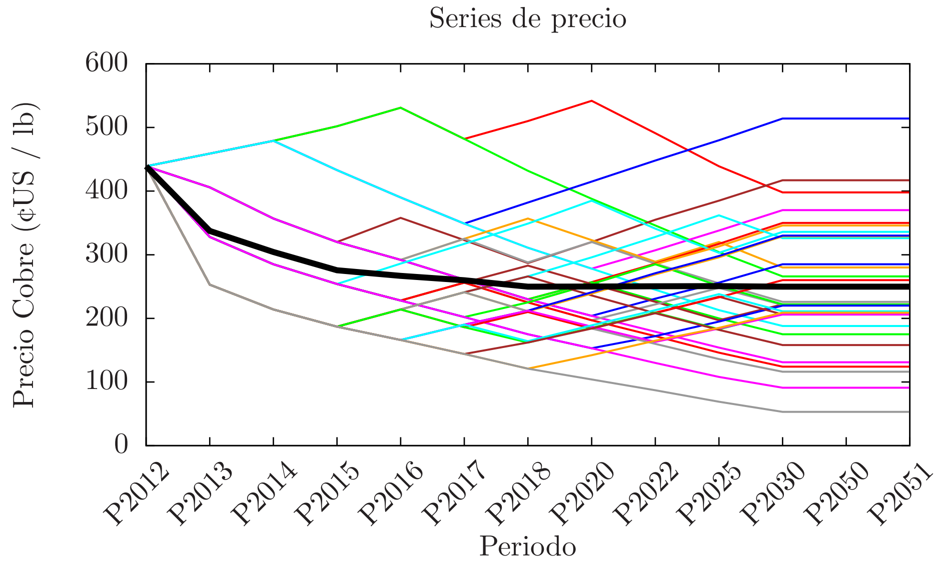


Figura 2.2: 32 series de precios, más la serie de precios promedio en color negro.

Escenario	VPN	Botadero	Proceso	Escenario	VPN	Botadero	Proceso
ESC1	2.455	517,7	11.775,3	ESC17	388	578,0	11,4
ESC2	1.988	517,7	19.316,8	ESC18	443	518,3	2.544,8
ESC3	1.701	517,7	11.466,1	ESC19	268	519,0	77.946,8
ESC4	1.262	517,7	84.348,2	ESC20	268	638,5	77.946,8
ESC5	1.176	518,3	66.889,2	ESC21	671	517,7	0,0
ESC6	1.083	517,7	78.433,8	ESC22	373	519,0	0,0
ESC7	1.018	517,7	47.20,3	ESC23	398	517,7	57.567,0
ESC8	902	518,3	11,4	ESC24	238	593,8	14.866,5
ESC9	715	517,7	10,5	ESC25	166	517,7	46.893,2
ESC10	621	518,3	82.380,1	ESC26	109	519,0	67.169,1
ESC11	584	517,7	81.122,3	ESC27	354	517,7	21.937,6
ESC12	578	560,5	81.122,3	ESC28	134	519,0	0,0
ESC13	1.046	517,7	10.998,3	ESC29	111	519,0	98.731,9
ESC14	517	519,0	12.896,4	ESC30	58	523,2	0,0
ESC15	540	517,7	37.860,1	ESC31	58	520,6	0,0
ESC16	388	519,0	11,4	ESC32	58	864,5	0,0

Tabla 2.1: VPN (en MM US$), cantidad de material que se va a Botadero (en Mega-toneladas) y cantidad de material que se va al proceso más caro (en Mega-toneladas) de un proyecto minero considerando 32 escenarios diferentes de precios (ver figura 2.2).

El tiempo que demora el algoritmo actual (PH) en entregar una solución como la presentada anteriormente considerando una instancia pequeña de 12 períodos, 8 fases, 157 bancos y 32 escenarios, es de menos de 2 horas en un computador tipo de US$ 2.000 (Pentium i7, 8 núcleos, 12 GB RAM).

Sin embargo, lo que se espera resolver con este algoritmo son entre 1000 y 1500 escenarios, minas de un tamaño superior, e incluso considerar dentro del modelo más de una mina. En este sentido, el tiempo de resolución puede llegar a ser tan largo que finalmente no es factible la utilización de esta herramienta tal cual se utiliza actualmente.

Es por ello que en este trabajo se pretende evaluar el "riesgo tecnológico" asociado a la resolución de este modelo en paralelo al ser procesado por un sistema de alto rendimiento. ¿Cuánto tiempo se gana? ¿De qué depende? ¿Cómo se paraleliza? ¿Qué tan complejo es hacerlo? Son algunas de las preguntas que se esperan responder a continuación.

2.4. Alcances

- El computador sobre el que se realiza el análisis analítico de la ganancia del algoritmo PH paralelizado corresponde a un Clúster de memoria distribuída con 9 procesadores Intel Xeon de 2,4 GHz y 8 núcleos cada uno. Uno de estos nodos cuenta con 24 Gb de memoria RAM, mientras que los 8 restantes cuentan con 48 Gb.
- La implementación desarrollada considera sólo la iteración inicial del algoritmo PH.
- La instancia considerada en la implementación de la paralelización considera 8 etapas, 157 bancos, 16 escenarios y 13 periodos.
- El computador utilizado en la implementación consiste en un procesador Intel Core i7 920 de 2,67 GHz, 8 núcleos, 12 Gb de memoria RAM y Windows 7.

Capítulo 3

Marco Teórico

3.1. Árboles y Escenarios

Según la literatura y el estudio realizado por Gacitúa [18], el precio del cobre se puede representar mediante un proceso de difusión del tipo browniano geométrico con reversión a la media. Este modelo, se representa matemáticamente mediante la siguiente ecuación para tiempos discretos:

$$\ln S_t = ln(S_{t-1})e^{-\kappa\Delta t} + \left(1 - e^{-\kappa\Delta t}\right)\left(\mu - \frac{\sigma^2}{2\kappa}\right) + \sigma\sqrt{\frac{1 - e^{-2\kappa\Delta t}}{2\kappa}}N(0,1)$$

En donde:

- S_0 es el valor inicial del precio.
- μ es el valor de largo plazo o de equilibrio.
- κ es la velocidad de reversión al valor de equilibrio.
- σ es una medida para la volatilidad del proceso.

Una vez que se tiene la expresión para simular el precio futuro del cobre, se procede a generar un árbol de escenarios para alimentar el modelo estocástico de planificación.

Aunque existen varias formas de transformar un movimiento browniano en un árbol, la técnica finalmente utilizada es mediante un ajuste de momentos, siendo una versión simplificada del trabajo de Hoyland y Wallace [24].

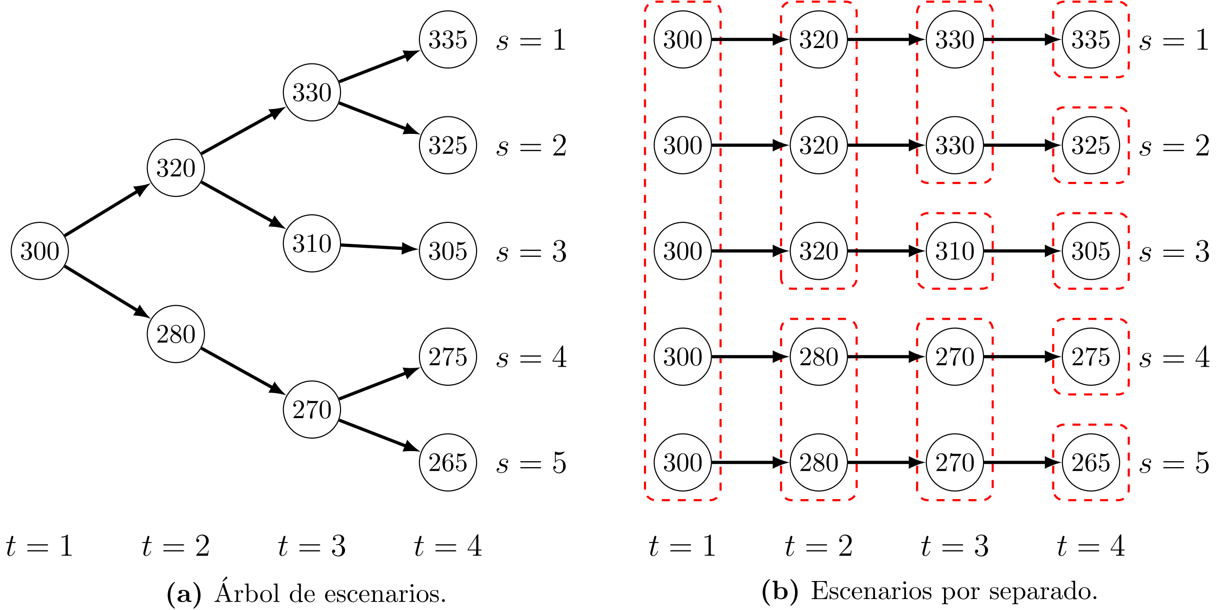


Figura 3.1: A la izquierda se muestra un arbol de 5 escenarios de precios en ¢ de dólar la libra (valor dentro de las circunferencias) y 4 periodos, mientras que a la derecha se muestra cada escenario por separado.

Finalmente, el resultado se puede apreciar en la figura 3.1, en donde se puede ver cómo un árbol de escenarios se traduce en distintos escenarios de precios.

Mayor explicación del método se puede encontrar en la tesis de Gacitúa [18].

3.2. No-anticipatividad

El principio de no-anticipatividad plantea que:

> Para cada par de escenarios, si son idénticos desde el período 1 hasta el período T, entonces las decisiones deben ser idénticas desde el período 1 hasta el período T, para todo período T.

En otras palabras, quiere decir que hoy se tiene que decidir con lo que se sabe hasta hoy, y no con lo que posiblemente pueda pasar en el futuro. El tomador de decisiones, al estar parado en un nodo del árbol, no sabe si está observando el escenario s o s', por lo tanto no hay motivo para tomar decisiones distintas.

Matemáticamente hablando existen dos formulaciones para incluir no-anticipatividad en un modelo: explícita y compacta.

La formulación explícita define variables de decisión para cada escenario por separado y luego, mediante restricciones, se impone la no-anticipatividad de los nodos que corresponde. Por otra parte, la formulación compacta define variables de decisión para cada nodo que pueden ser compartidas por más de un escenario, y en donde cada escenario se define por un "conjunto de información" o "conjunto de nodos" asociados a éste. De esta forma, la no-anticipatividad se cumple implícitamente.

Aunque la implementación de la formulación compacta es más compleja que la formulación explícita, tiene la ventaja de que son necesarias menos variables de decisión y restricciones en un modelo estocástico.

3.3. Progressive Hedging

Progressive Hedging (PH) es una metodología de descomposición por escenarios para solucionar problemas estocásticos publicada por R.T. Rockafellar y Roger J-B Wets en el año 1991 [32]. Es un algoritmo exacto para problemas convexos, lo que significa que es capaz de converger al óptimo global del problema. Su aparición nace de la necesidad de resolver problemas estocásticos de gran tamaño desde el punto de vista computacional.

El pseudo-código intuitivo del algoritmo es:

1. Se resuelve cada escenario determinístico por separado.
2. Se calcula la solución global en cada nodo del árbol de escenarios.
3. Si las soluciones de los escenarios son similares a la solución global en cada nodo, PARAR.
4. Se calcula una penalización por alejarse de a la solución global.
5. Se resuelve cada escenario con la penalización en la función objetivo.
6. Ir al punto 2.

3.4. Computación en paralelo

Tradicionalmente los programas computacionales fueron desarrollados pensando en un procesamiento secuencial. Esto significa entonces que los programas pueden ser divididos en una serie discreta de eventos en donde cada uno de estos eventos corresponde a una instrucción a procesar. Luego, cada una de estas instrucciones se ejecuta una detrás de otra, nunca ejecutándose más de una a la vez.

Sin embargo mucho de estos programas pueden ser divididos en varios bloques o partes que son independientes entre sí lo cuál hace que de igual el orden de procesamiento en el procesador. He aquí donde nace la idea de procesar cada uno de estos bloques en un procesador diferente. Es por ello que las tecnologías de la actualidad cuentan ya sea con computadores compuestos de varios procesadores, varios computadores de un procesador conectados, o una combinación de ambos.

Las principales razones del uso de la paralelización son el ahorro de tiempo y la resolución de problemas más grandes.[1]

A continuación se explican diversos conceptos a utilizar durante el desarrollo del informe, además de cubrir varios tópicos relacionados con la computación en paralelo.

3.4.1. Conceptos

Sistema operativo

Un sistema operativo (OS) corresponde al principal programa dentro de un sistema computacional. Es el encargado de manejar los recursos físicos del sistema (hardware) y proveer servicios básicos a los programas o aplicaciones (software).

El kernel por su parte corresponde al componente más importante del OS, pues es el puente que existe entre las aplicaciones y el procesamiento realizado por el hardware. Dentro de las funciones específicas del Kernel están la ejecución de los programas, el manejo de la memoria disponible, el manejo del acceso a los diferentes discos físicos, entre otras.

Dentro de los kernel más utilizados se encuentran el Kernel de Linux, utilizado por distribuciones tales como Ubuntu y Android, y el Kernel de Micrososft, utilizado en todos los sistemas operativos Windows.

Además del kernel el sistema operativo cuenta con componentes para el manejo de la interconexión de redes, la seguridad informática y la interfaz de usuario, la que puede ser mediante línea de comandos (DOS) o por medio de una interfaz gráfica (windows).

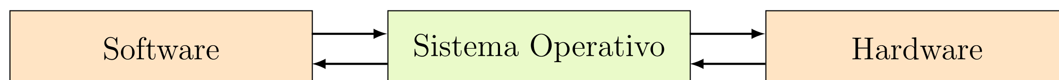


Figura 3.2: El sistema operativo es el puente entre el Software y el Hardware en un computador.

[1] En el sitio web www.top500.org se pueden encontrar diversas estadísticas en cuanto a las áreas de aplicación de la paralelización o los computadores más poderosos del mundo.

Procesos

Un programa computacional corresponde simplemente a un archivo que contiene un conjunto de instrucciones en algún orden en particular. Estas instrucciones deben estar en un lenguaje binario, el cual puede ser interpretado por el procesador de un computador. Este lenguaje binario es el resultado de “compilar” un código fuente, el cual está en un lenguaje entendible para el programador.

Al ejecutar un programa (ie, decirle a la CPU del computador que llevar a cabo su set de instrucciones) se genera una instancia del programa, a la cual se le denomina proceso. Este proceso contiene el código del programa, su estado de ejecución en la CPU, los recursos del sistema asignados y el espacio de memoria asociado. Esto último permite que puedan existir varios procesos (instancias) de un mismo programa sin mezclar ni recursos ni memoria. Por ejemplo, al navegar por internet, uno puede tener dos páginas web abiertas simultáneamente con el mismo programa. En este caso, cada una de estas páginas es resultado de un proceso diferente.

Threads (hilos)

Cada proceso está compuesto por uno o más threads, los cuales corresponden a una porción del código del programa en ejecución. Una vez que todos los threads de un proceso mueren, es decir terminen de ejecutar el código que se les asignó, entonces el proceso finaliza y tanto la memoria como los recursos asociados a dicho proceso son liberados para ser utilizados en otros procesos.

La principal característica de los threads es que son capaces de ejecutarse en paralelo. Muchas veces los procesos contienen porciones de código independientes entre sí, lo que hace que ejecutar dichas porciones en threads diferentes acelere el tiempo total de procesamiento del programa.

A diferencia de los procesos, los threads comparten la memoria y recursos asignados al proceso. Esto quiere decir que si un thread realiza algún cambio en la memoria, entonces todos los otros threads verán dicho cambio de manera instantánea. Un ejemplo de un programa con varios threads puede ser un reproductor de música. Por un lado está el thread que “escucha” todo lo que hace el usuario (apretar botones, escribir, etc.) mientras que otro thread es el que reproduce la música seleccionada.

Agendamiento

Aunque en un comienzo los computadores sólo eran capaces de computar un proceso a la vez, hoy en día la mayoría de los sistemas computacionales son capaces de ejecutar varios procesos simultáneamente. Por ejemplo es posible navegar en internet mientras se

escucha música. Es por ello que una de las tareas primordiales de los sistemas operativos tiene que ver con el agendamiento, el cual se encarga de asignar recursos de sistema a los procesos y threads.

Un proceso puede entenderse como un diagrama de estados similar al que se muestra en la figura 3.3. Al momento de crear un proceso éste debe esperar a que el kernel del sistema operativo lo deje entrar al conjunto de procesos que se están ejecutando, entrando al espacio de memoria principal del sistema. Una vez en este estado el proceso se mueve ida y vuelta hacia el estado de procesamiento, en donde una de las CPU (o core) del sistema ejecuta las instrucciones. Cada CPU no puede ejecutar más de un proceso a la vez.

Muchas veces un programa debe esperar alguna señal desde otro thread del mismo programa u otro programa completamente diferente. En este caso, pasa al estado de bloqueo.

Finalmente, cuando el proceso termina su ejecución, se mueve al estado de término en donde el proceso completo muere.

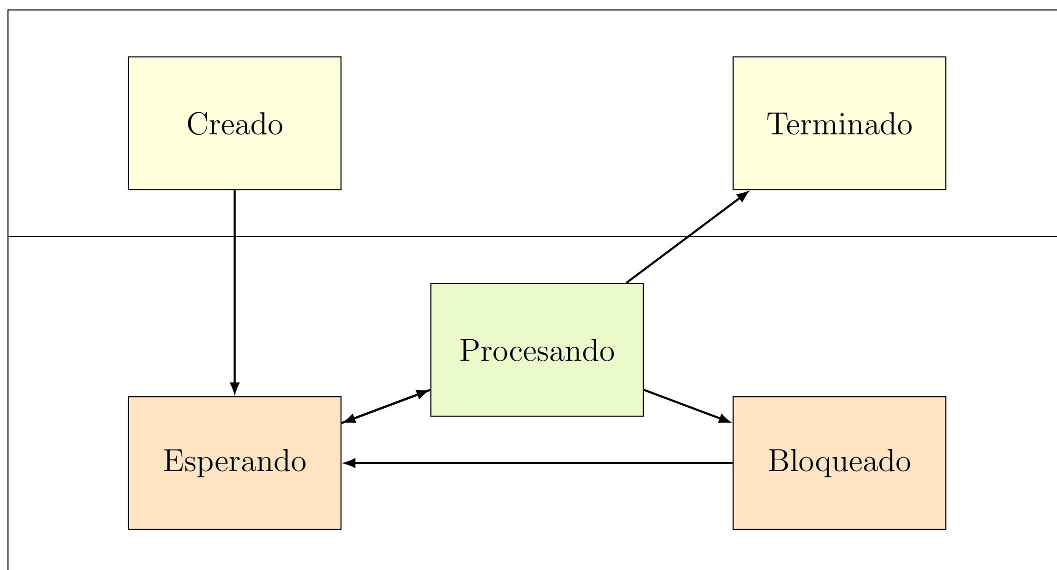


Figura 3.3: Diagrama de estados de un proceso.

3.4.2. Nivel de paralelización

Dependiendo del tipo de programa y la información con la que se cuenta se pueden lograr diferentes niveles de paralelización. Este nivel se enfoca principalmente a qué cosa, dentro de un programa, es paralelizado. Aunque existen otros niveles, para motivos de este trabajo los más relevantes corresponden a la paralelización de procesos y la

paralelización de información. Ambos niveles tienen una alta relación y muchas veces un programa se paraleliza en diferentes niveles simultáneamente.

Paralelización de procesos

La paralelización de procesos corresponde a la paralelización de threads a través de los procesadores. Cada thread puede realizar la misma o diferente función y utilizar la misma o diferente información respecto a los otros. Este nivel de paralelización se basa en la naturaleza del programa, en donde ciertas partes de código son independientes entre sí y por ende pueden ser procesadas de manera distribuida.

Debido a la naturaleza de los threads es necesario establecer métodos de coordinación entre ellos, ya que por ejemplo un thread podría estar cambiando información que otro thread necesita que no cambie. Algunas de estas metodologías se describen en la sección 3.4.3.

Paralelización de información

Por otra parte la paralelización de información se basa en la naturaleza de la data utilizada por alguna función del programa, la cual es fácilmente divisible en bloques. Esta función se procesa en threads en donde cada uno utiliza, procesa y afecta sólo el bloque de data que se le entrega. Al igual que en el caso anterior las metodologías de coordinación entre los threads es necesaria para que tanto la información que se tiene como la que se genera se mantenga íntegra.

3.4.3. Coordinación

Al paralelizar, los distintos threads deben ser capaces de seguir trabajando en conjunto. Para ello existen diferentes metodologías de coordinación que ayudan a controlar tanto el procesamiento como la información relacionada.

Sincronización

El término sincronización hace alusión a la necesidad de dos o más procesos o threads de intercambiar información acerca de lo que están haciendo cada uno por separado. En ese sentido, la sincronización se puede dividir en dos tipos: sincronización de procesos y sincronización de datos.

La sincronización de procesos se refiere a que si un thread comienza a ejecutar una porción del código que no debe ser ejecutado más de una vez simultáneamente, entonces cualquier otro thread que intente acceder al mismo código debe esperar a que el primero termine. Esto se logra mediante una señalética denominada semáforo.

Muchas veces distintos procesos trabajan con copias de la misma información. Sin embargo eventualmente se deben juntar los datos de los distintos procesos. A este proceso se le denomina sincronización de datos, en donde la idea primordial es mantener los sets de data coherentes entre ellos.

Barrera

Durante la ejecución de un programa, existen puntos en donde dos o más threads deben juntarse a "conversar" luego de cada uno haber ejecutado una porción de su código. En general en la literatura a este punto se le llama punto de sincronización, pues lo que en general ocurre es una sincronización de datos. Sin embargo también es conocido como "barrera", ya que cada proceso una vez alcanza el punto, debe detenerse a esperar a todos los otros que deben llegar.

Punto de control

El punto de control es una técnica utilizada para guardar el estado completo de un programa en ejecución, para en un tiempo futuro reanudarlo y continuar como si nunca hubiese existido dicho corte. Una de las principales ventajas de esta metodología es la capacidad de portar la aplicación a otro sistema para ser retomada.

3.4.4. Arquitectura de memoria

La memoria en computación se refiere generalmente a dispositivos semiconductores capaces de almacenar información temporal y cuya velocidad de acceso puede llegar a ser extremadamente alta. Esta información temporal es la que utiliza el procesador del computador para realizar sus tareas y almacenar los resultados de sus operaciones.

La memoria utilizada en los computadores actuales corresponde a la memoria RAM (Random Access Memory), cuya particularidad es que cualquier dirección de la memoria puede ser accedida en cualquier momento, a diferencia de por ejemplo un cassette en donde debe ser leído en secuencia independiente del contenido al que se desee obtener.

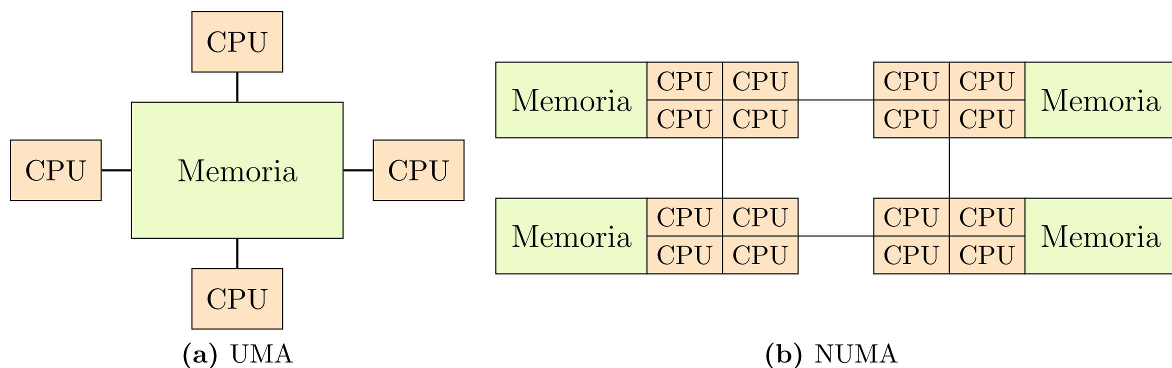
Junto a la necesidad de computadores con muchos procesadores nacieron tres principales formas de distribuir la memoria entre estos procesadores: mediante memoria compartida, mediante memoria distribuida y mediante una combinación de ambas.

Memoria compartida

En los computadores en donde la memoria es compartida, todos los procesadores son capaces de acceder a todos los espacios de memoria, mientras que su trabajo sigue siendo independiente. Esto implica que si un procesador cambia la información dentro de un espacio de memoria entonces todos los procesadores verán reflejado dicho cambio de manera automática e instantánea. La paralelización en este tipo de sistema se logra generalmente mediante la utilización de threads.

Máquinas construidas bajo este concepto pueden dividirse en dos tipos: con memoria de acceso uniforme (UMA por sus siglas en inglés) y con memoria de acceso no-uniforme (NUMA).

Para el caso de la arquitectura UMA, existe una memoria global a la que todos los procesadores son capaces de acceder con la misma prioridad. Por otra parte en la arquitectura NUMA cada procesador (o grupo de procesadores) tiene asociada una memoria local, la cual está conectada con la memoria local de los otros procesadores mediante una conexión física. Esta conexión permite el acceso de un procesador hacia la memoria local de otro pero con una prioridad disminuida respecto a la prioridad que tiene el procesador local de dicha memoria, y además a una velocidad mucho menor.



(a) UMA **(b)** NUMA

Figura 3.4: Arquitecturas de memoria compartida

Dentro de las ventajas de este tipo de diseño se encuentra la rapidez y uniformidad en el traspaso de información entre distintas tareas de los distintos procesadores. Sin embargo la escalabilidad al agregar más procesadores conectados a la misma memoria genera que haya un incremento de tráfico geométrico en el acceso a esta, lo que se traduce finalmente en que el tiempo ganado por paralelizar es opacado por el tiempo perdido en esperar el acceso a la memoria.

Memoria distribuida

En los sistemas de multiprocesamiento con memoria compartida cada procesador (o grupo de procesadores) posee su propia memoria privada. De esta forma las tareas que desee ejecutar dicho procesador sólo pueden operar con la información local y, en el caso que información remota sea requerida, la tarea debe ser capaz de establecer una comunicación con los otros procesadores. La paralelización en este caso se logra mediante un modelo de traspaso de mensajes, en donde deben existir tareas con métodos de envío de información y por otra parte tareas con métodos para recibir información.

Uno de los ejemplos más comunes dentro de esta arquitectura son los sistemas clúster. Estos sistemas se forman conectando distintos computadores mediante una red de área local. Esto hace que, a diferencia de la arquitectura memoria compartida, los computadores pueden tener tanto procesadores como memoria diferente entre ellos ya que su forma de trabajo es mucho más independiente.

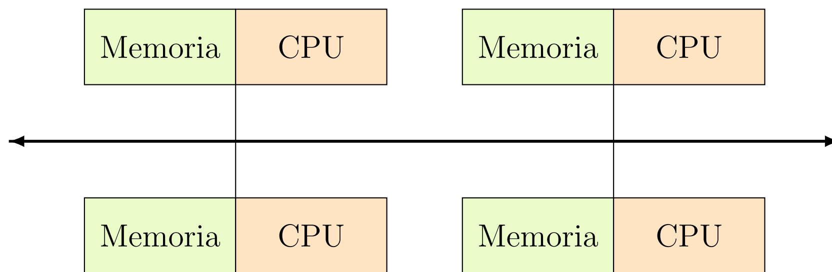


Figura 3.5: Arquitectura de memoria distribuida

Una de las principales ventajas de la memoria distribuida es la escalabilidad del sistema al aumentar el número de procesadores y memoria asociada debido a la eliminación de sobrecarga incurrida entre los procesadores al acceder al mismo espacio de memoria. Sin embargo la comunicación entre los procesadores es tarea completa del programador. Esto quiere decir que se deben establecer métodos y protocolos de comunicación entre las tareas de distintos procesadores.

3.4.5. Consideraciones al paralelizar

A continuación se explican dos conceptos relacionados con los focos con los que debe lidiar un programador al momento de paralelizar un programa: el balance de carga y la granulidad del programa.

Balance de carga

El balance de carga hace referencia a la idea de distribuir lo más uniforme posible el trabajo entre las distintas tareas, de modo que todas las tareas se mantengan ocupadas todo el tiempo. Por ejemplo, en caso de existir un punto barrera, la idea sería que todas las tareas llegasen a él simultáneamente ya que en caso contrario la última tarea que termine determinaría el tiempo total del programa.

Dependiendo de la arquitectura del sistema de multiprocesadores, el lograr un buen balance de cargas puede ser una tarea sencilla o difícil. Dado por ejemplo un sistema clúster, en donde los computadores no es necesario sean idénticos, entonces para una mejor distribución es necesario saber por ejemplo la cantidad de procesamiento por segundo de cada computador, de modo de poder tener una idea clara de cuánto puede procesar uno respecto al resto.

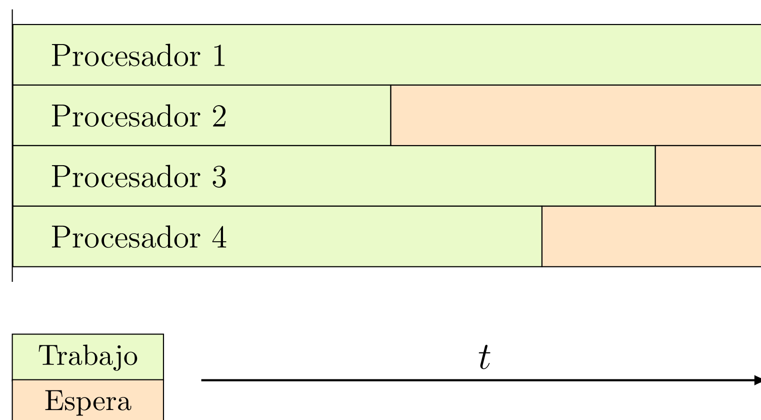


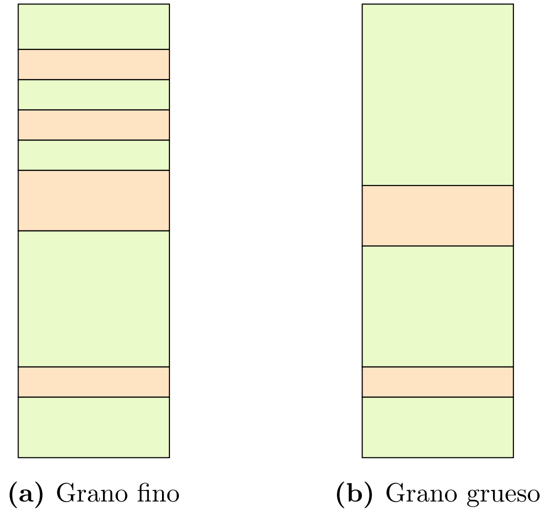
Figura 3.6: Carga desbalanceada entre los distintos procesadores

Granulidad

La granulidad de un programa es una medida cualitativa del tiempo ocupado en computar versus el tiempo ocupado en comunicar. Como se dijo en la sección 3.4.3, la comunicación entre las distintas tareas debe ser realizada con algunas precauciones, para lo cual existen distintas metodologías de coordinación.

Se dice que un programa paralelizado es de grano fino cuando existe relativamente poco trabajo entre eventos de comunicación mientras que es de grano grueso cuando los eventos de comunicación son muy pequeños en relación al trabajo desde el punto de vista tanto de cantidad como de tiempo necesario.

No existe un nivel de granulidad óptima general ya que ésta depende tanto del programa como del sistema multiprocesador en el que se ejecute. Un programa de grano fino puede generar mucha sobrecarga de comunicación (overhead) mientras que un programa de grano grueso puede incurrir en un desbalance de carga entre los procesadores.



(a) Grano fino **(b)** Grano grueso

Figura 3.7: Esquema de granulidad: los cuadros verdes representan tiempo de trabajo mientras que los naranjos representan tiempo de sincronización.

3.4.6. Teoría de paralelización

Ganancia y eficiencia

La ganancia obtenida al paralelizar se refiere a cuán más rápido es el programa paralelizado respecto al mismo programa pero secuencial. Esta ganancia se define como:

$$Ganancia(n) = \frac{T(1)}{T(n)}$$

en donde n es el número de procesadores, $T(1)$ es el tiempo de ejecución del programa secuencial y $T(n)$ el tiempo de ejecución del programa con n procesadores.

A su vez, la eficiencia de la paralelización se define como:

$$Eficiencia(n) = \frac{Ganancia(n)}{n} = \frac{T(1)}{nT(n)}$$

la cual trata de estimar cuán bien utilizados están siendo los procesadores al procesar el programa, en contraste con el esfuerzo utilizado en comunicación y sincronización. En general la eficiencia es mejor indicador que la ganancia en términos de observar qué tan buena es la paralelización.

Ley de Amdahl

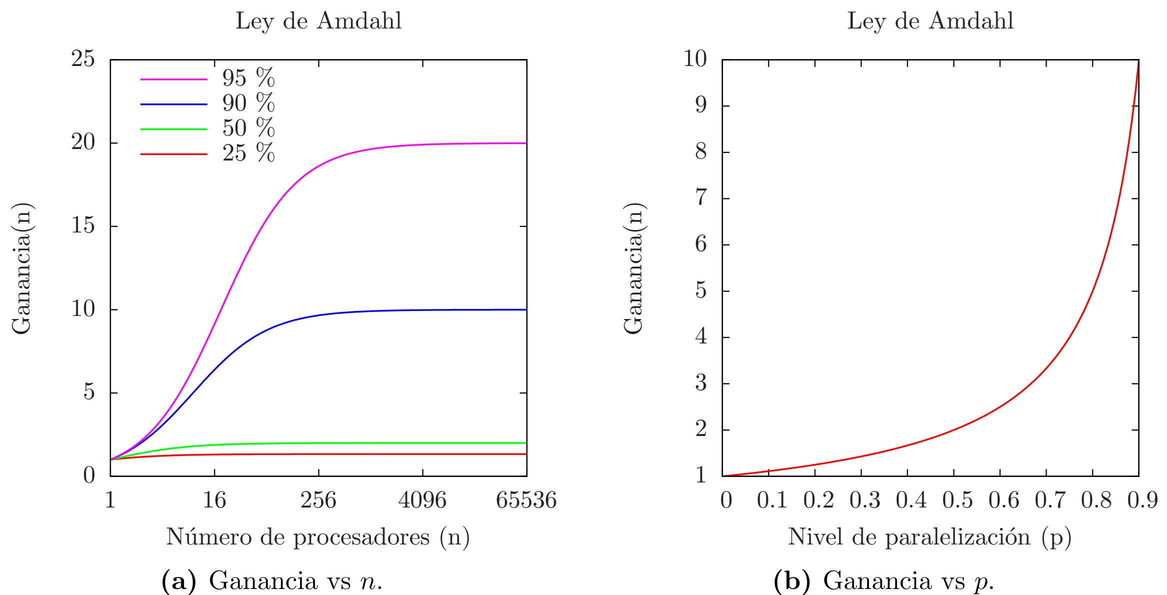
La ley de Amdahl es utilizada para encontrar la máxima ganancia esperada de un programa al ser paralelizada una parte de éste.

En general un programa no es posible de paralelizar completamente, y por ende se generan cuellos de botella que limitan la ganancia obtenida al paralelizarlo. Por ejemplo si se tiene un programa que demora 2 horas, en donde 1 hora (50 %) no es paralelizable mientras que la otra hora sí lo es, entonces independiente cuanto demore la parte paralelizable el programa entero estará sujeto a la restricción que impone la parte no paralelizable, haciendo que el programa a lo menos demore 1 hora en procesar. Esto se traduce finalmente en que la ganancia al paralelizar es a lo más de 2.

La ley de Amdahl se define como:

$$Ganancia\ maxima = \frac{1}{\frac{p}{n} + s}$$

en donde p corresponde a la fracción de programa paralelizable, s a la fracción de programa secuencial y n al número de procesadores.



(a) Ganancia vs n.

(b) Ganancia vs p.

Figura 3.8: El gráfico (a) muestra el límite de ganancia al que se puede llegar dados distintos niveles de paralelización, mientras que el gráfico (b) muestra cómo se mueve el límite de ganancia según el nivel de paralelización.

Ley de Gustafson

La ley de Amdahl se basa en la premisa de que el programa paralelizado utilizará la misma información o el mismo tamaño de problema que la versión secuencial. La ley de Gustafson por otra parte propone que dado un tiempo fijo, el problema secuencial es capaz de procesar un problema de tamaño x mientras que al paralelizarlo es capaz de procesar un problema de tamaño y, en donde el problema y es más grande que el problema x.

De este modo, la ley de Gustafson se define como:

$$Ganancia(n) = n - s * (n - 1)$$

en donde s corresponde a la fracción de programa secuencial no paralelizable y n al número de procesadores.

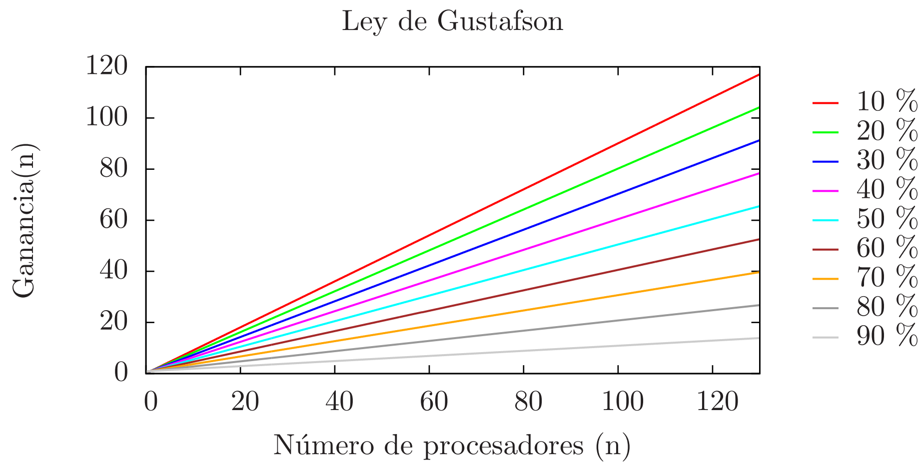


Figura 3.9: Distintas curvas de la ley de Gustafson dependiendo de la fracción no paralelizable del programa.

Una manera simple de entender la diferencia entre las leyes es que la ley de Amdahl dice que la ganancia que se puede obtener paralelizando esta limitada por la parte secuencial del programa. Por otra parte la ley de Gustafson dice que al aumentar el numero de procesadores se puede aumentar el tamaño del problema en cuanto a información, haciendo que el impacto de la parte secuencial sea reducido.

Capítulo 4

El Problema Minero

A continuación se describe el problema de planificación que enfrentan las compañías, el modelo matemático determinístico utilizado actualmente, el modelo matemático estocástico utilizado y finalmente la aplicación de PH en el modelo estocástico.

4.1. Descripción operacional de una mina

A continuación se busca explicar a grandes rasgos el proceso minero, el cual se puede desagrupar en dos grandes partes: la extracción y el procesamiento. Para mayor detalle, referirse a [19].

4.1.1. La extracción

El principal objetivo de este proceso consiste en extraer el mineral desde el macizo rocoso de la mina, para luego ser enviado a planta, todo esto de una manera eficiente de modo de obtener el mayor beneficio neto por la explotación.

Uno de los requerimientos necesarios para elaborar un buen plan de extracción, es conocer la mina. Para ello, se descompone el terreno en un cuadriculado tridimensional, al cual se le asigna unitariamente parámetros tales como tonelaje y ley, los cuales son definidos mediante métodos estadísticos de geología y resultado de la exploración mediante sondajes de la zona.

El retiro de material se realiza mediante etapas sucesivas desde la superficie hacia el fondo del rajo. Para ello, y con el objetivo de simplificar el esquema, se divide la mina en elementos de menor tamaño denominados expansiones. Geométricamente hablando, una expansión corresponde a una tajada de la pared del rajo tal y como se aprecia en la

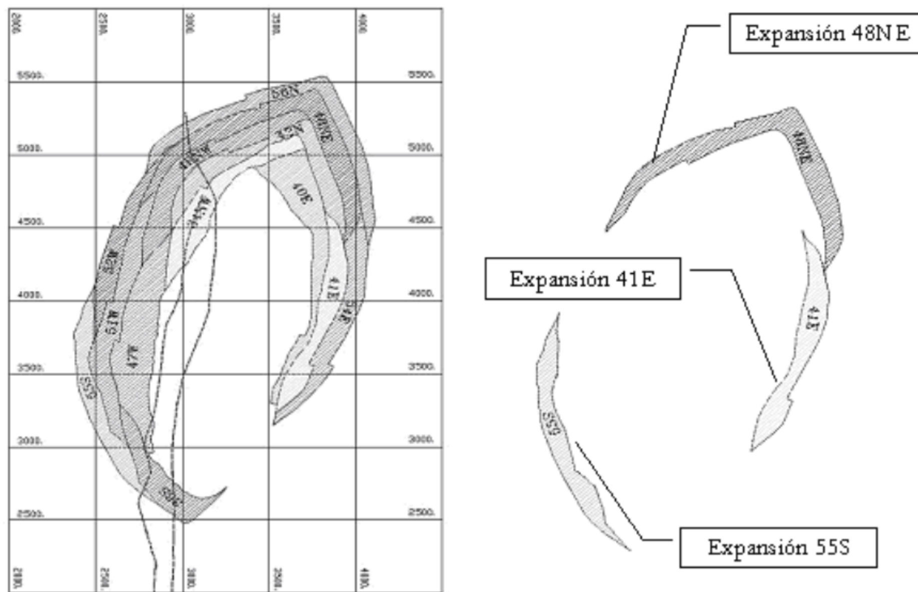


Figura 4.1: Diseño de expansiones de una mina.

figura 4.1. Luego, cada expansión, está compuesta por los cuadriculados mencionados anteriormente, a los cuáles se les denomina bancos (ver figura 4.2).

Es por esto que, la principal decisión a largo plazo en esta etapa, corresponde a definir las expansiones y el orden de extracción de cada banco, considerando las restricciones físicas y operacionales asociadas.

Finalmente, y a través de procesos de perforación, tronadura, carguío y transporte es que el material es llevado desde la mina hacia la red de procesamiento.

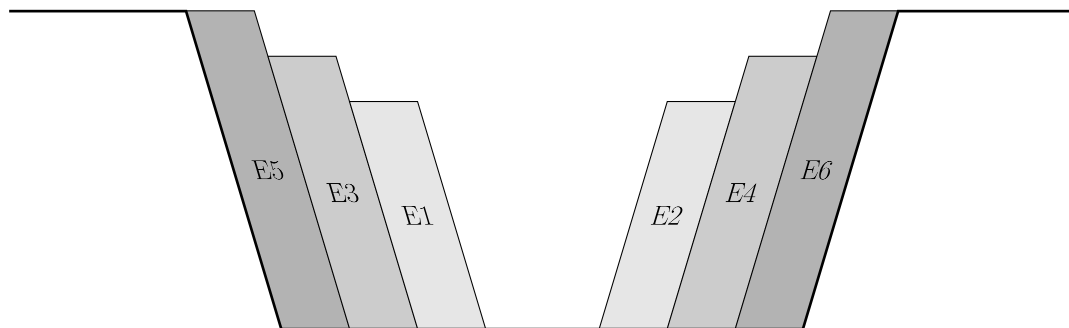


Figura 4.2: Visión esquemática de un corte transversal de una mina a cielo abierto y sus expansiones.

4.1.2. La red de procesamiento

Una vez el material ha sido extraído, éste se traslada a la red de procesamiento, la cual se compone de tres etapas: almacenamiento, chancado y procesamiento.

El almacenamiento, a su vez, se divide en dos partes: botadero y stock. El material extraído que no es económicamente rentable de acuerdo a las condiciones actuales de tecnología y mercado es enviado a la zona de botadero, el cual se podría catalogar como un "basurero". Por otra parte, el material que podría llegar a ser económicamente rentable, se envía a stock para decidir a futuro si conviene o no conviene enviar a procesar.

En la etapa de chancado, el material se ingresa a una máquina capaz de reducir el tamaño de las rocas. Cabe notar que, dependiendo del proceso posterior, puede ser necesario un chancado más fino para lo cual existen el chancado primario, secundario y terciario.

Finalmente, la roca se envía a la planta de molienda o a la planta de lixiviación, en donde en la primera se obtiene producto concentrado y molibdeno, mientras que en la segunda se obtienen láminas de alta pureza.

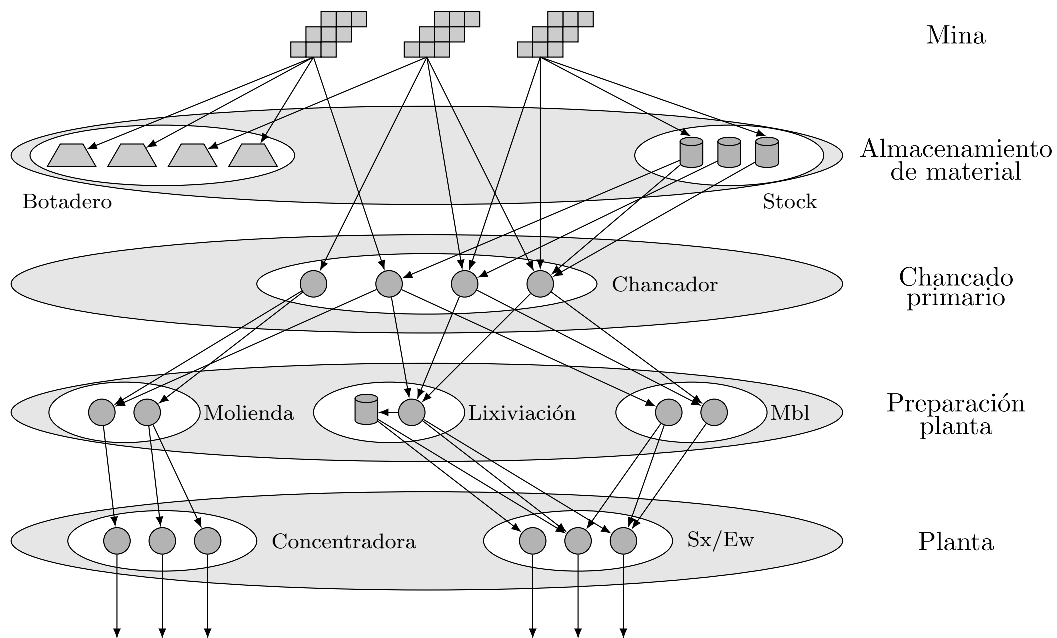


Figura 4.3: Esquema de la red de procesamiento.

4.2. Modelo determinista

El problema de la planificación minera, considerando sólo variables determinísticas, se modela matemáticamente como un problema de flujo de redes para el transporte y procesamiento del mineral, mientras que la decisión de los flujos de material se modela como un problema de extracción. Luego, y mediante la restricción de que todo lo que se extrae debe ser transportado para cada periodo, se unen ambos problemas [1].

Las principales variables de decisión del modelo se enfocan en responder las preguntas de cuándo, cuánto, dónde y con qué se extrae, transporta y procesa cada banco de material, para cada periodo de tiempo en el horizonte de evaluación. Cada una de estas decisiones en un periodo determinado influye en el estado inicial del periodo siguiente, y por consiguiente en las decisiones a tomar en dicho periodo.

Por otra parte las principales restricciones del modelo se pueden dividir en dos grupos: las asociadas a consideraciones físicas que se deben tener en cuenta en las minas de cielo abierto y las asociadas al transporte y procesamiento del mineral. Para el primero caso se encuentran la relación en la explotación entre las distintas fases además del hecho de que sólo se puede sacar el material de la tierra que está a la vista, entre otras cosas.

Por otra parte, las restricciones de transporte y procesamiento deben consideran entre otras las capacidades de las maquinarias, la conservación de los flujos entre los nodos y entre los periodos, y el almacenamiento de material en bodegas.

Finalmente la función objetivo considera los beneficios que genera la venta del mineral extraído, los costos asociados a su extracción, transporte y procesamiento, y las inversiones asociadas a la maquinara y plantas necesarias para el funcionamiento de la operación.

4.3. Modelo estocástico

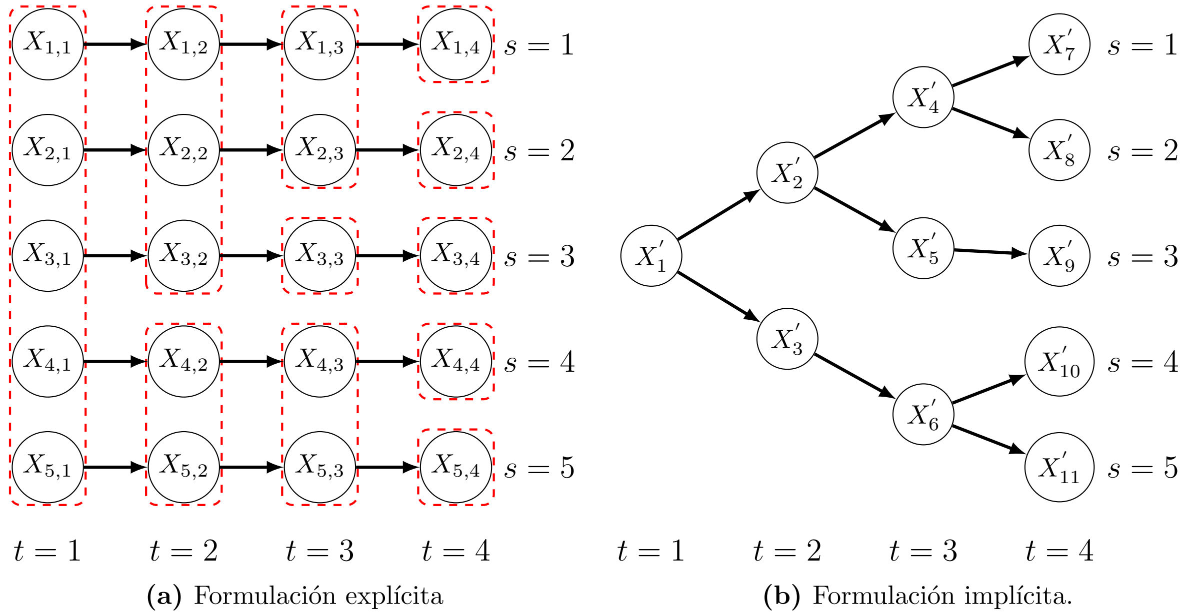
La estocasticidad en el precio del cobre, de modo de incluirla en el modelo, se representa según un árbol de escenarios (ver 3.1). La formulación estocástica para este tipo de problemas se puede llevar a cabo de dos maneras: de forma extendida (explícita) o de forma compacta (ímplícita), en donde para ambos casos se debe cumplir el principio de no anticipatividad (ver 3.2).

La formulación extendida de un modelo estocástico se compone de tantos modelos determinísticos como escenarios se consideren. Del mismo modo se tienen variables de

[1] Para el modelo matemático deteminístico utilizado, implementado inicialmente por Goic [19], referirse al anexo A

decisión para cada uno de estos modelos, a las cuáles por medio de restricciones se les hace cumplir el principio de no anticipatividad.

La formulación compacta por su parte considera que cada nodo del árbol de precios tiene asociado un conjunto de decisiones, las cuales son compartidas por cada uno de los escenarios que consideran un mismo nodo. De esta forma, la no anticipatividad se cumple implícitamente.



(a) Formulación explícita

(b) Formulación implícita.

Figura 4.4: Variables de decisión según su formulación estocástica.

La figura 4.4 muestra gráficamente los dos tipos de formulaciones. Para el caso de la formulación explícita, las envolturas rojas hacen referencia a las restricciones necesarias para el cumplimiento de la no anticipatividad. En el caso de la formulación implícita se aprecia como ciertas variables son compartidas por más de un escenario.

Aunque la programación de la formulación compacta es más compleja, el hecho de tener menos ecuaciones y variables en el modelo respecto de la formulación extendida hace que su aplicación sea atractiva debido a la menor carga que éste significa para un computador, lo que finalemnte se traduce en reducción de tiempos o la posibilidad de resolver problemas mucho más grandes.

En el caso particular del problema estocástico de planificación minera, y debido al gran tamaño ya del problema determinístico, se utiliza la formulación implícita del modelo.

4.4. Aplicación de PH al modelo

Gacitúa en su trabajo [18] desarrolla una implementación del algoritmo PH en el modelo estocástico de planificación minera de largo plazo. Sin embargo sus conclusiones apuntan a que el algoritmo no es capaz de entregar una solución entera factible en un tiempo sensato, por lo que optó por realizar un *crossover* entre el modelo con PH y el modelo estocástico compacto.

El concepto de *crossover* hace alusión a utilizar los resultados del primer modelo (PH) como punto de partida del segundo modelo (estocástico) mediante una estrategia de fijación de variables. El procedimiento consiste en realizar un pequeño número de iteraciones PH, revisar qué variables cumplen el principio de no anticipatividad, fijar dichas variables con los resultados obtenidos, y finalmente resolver el modelo estocástico compacto considerando esta fijación.

A continuación se describe el procedimiento completo de la implementación de PH al modelo minero:

1. **Inicialización.** Se inicializa el modelo determinístico, en donde se leen los parámetros asociados a la instancia que se resuelve. Además se establece el número de iteraciones que el algoritmo PH realizará ($iter$).

2. **PH Inicial.** Se resuelve cada escenario utilizando el modelo determinístico.

3. Si el número de iteraciones PH es cero ($iter = 0$) entonces ir al paso 7.

4. **PH Actualizaciones.** Se evalúan los resultados de todos los escenarios y se actualiza la función objetivo del modelo, estableciendo el factor de penalización del algoritmo PH.

5. **PH Iteraciones.** Se resuelve cada escenario utilizando el modelo determinístico y la función objetivo modificada.

6. Repetir desde el paso 4 $iter$ veces.

7. **Fijación.** Se revisan los resultados y se fijan las variables que cumplan con el principio de no anticipatividad.

8. **Modelo estocástico.** Se inicializa y resuelve el modelo estocástico compacto considerando la fijación de variables realizada.

Capítulo 5

Paralelización de PH

Tal como se adelantó anteriormente, el algoritmo PH tiene la característica de descomponer un problema estocástico de gran tamaño en problemas determinísticos más pequeños los cuales pueden ser resueltos de manera independiente. Esta independencia es lo que hace atractivo el resolver el algoritmo de manera paralela para cada iteración.

Según lo estudiado por Somervell [35], existen diferentes métodos de paralelizar el algoritmo. La forma más intuitiva y simple corresponde a paralelizar el procesamiento dentro de cada iteración. Sin embargo existen otras metodologías que involucran modificar el algoritmo PH. A continuación se explican las diferentes metodologías así como sus distintas ventas y desventajas.

5.1. Métodos simples

La paralelización simple de PH es una extensión en el procesamiento del algoritmo en donde se paraleliza la resolución de todos los escenarios dentro de una iteración, agregando además un punto de barrera entre iteraciones de modo de realizar las actualizaciones pertinentes.

En vez de resolver cada escenario de manera secuencial lo que se hace es enviar todos los escenarios de la iteración a los distintos procesadores de modo que la resolución se haga de modo paralela. En caso de que existan menos procesadores que escenarios (que es lo lógico) entonces debe existir un *procesador maestro* encargado de controlar el envío de los escenarios de modo que cada *procesador esclavo* tenga a lo más un escenario asignado en cada momento. Este control puede lograrse de manera síncrona o asíncrona.

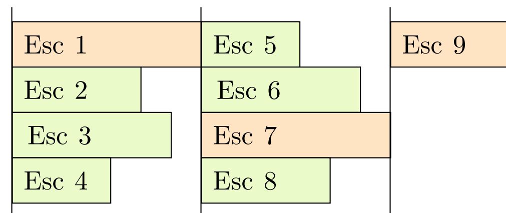
5.1.1. Método síncrono

El método síncrono se logra dividiendo el total de escenarios en grupos (batchs) que contengan tantos escenarios como procesadores esclavos existan. Luego se envía cada uno de los escenarios dentro de un grupo a cada procesador para ser resuelto. Una vez que **todos** los procesadores terminan entonces el procesador maestro puede enviar a procesar el siguiente grupo de escenarios. Esto se repite hasta que no quedan más escenarios.

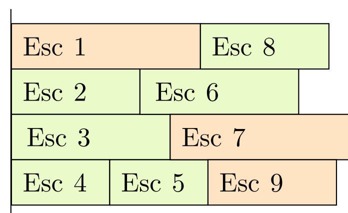
Una vez que se han procesado todos los escenarios, el procesador maestro cuenta con la solución completa de la iteración por lo que se dedica a realizar las actualizaciones necesarias por el algoritmo PH, calcula el nivel de convergencia, y de no haber convergido entonces se procesa la siguiente iteración. Esto se muestra en la figura 5.1 (a).

5.1.2. Método asíncrono

El método asíncrono es similar al síncrono, solo que en vez de que el procesador maestro en cada iteración espere que terminen todos los escenarios de un grupo para mandar los siguientes a resolver, éste está constantemente enviando y recibiendo soluciones apenas cada procesador esclavo encuentra la solución de un escenario.



(a) Método síncrono.



(b) Método asíncrono.

Figura 5.1: Arriba se muestra como se procesarían 9 escenarios de manera síncrona. Abajo se muestran los mismos 9 escenarios pero procesados de manera asíncrona.

En el caso de que el tiempo que demora cada escenario es exactamente igual para todos, entonces no existe diferencia real entre el método síncrono y el asíncrono. Sin embargo cuando el tiempo de resolución de cada escenario corresponde a una variable aleatoria entonces la versión asíncrona obtiene mejores resultados en términos de tiempo, ya que no *pierde tiempo* esperando sino que lo utiliza para seguir procesando.

Un problema que presentan los métodos simples en sistemas de memoria compartida es el overhead provocado por el acceso de los distintos procesadores al espacio de memoria. A pesar que la información se mantenga físicamente en lugares diferentes, existe una velocidad de acceso máxima permitida que, al momento de realizar multiprocesamiento, se hace notoria.

5.2. Métodos de bloques-cíclico

La idea detrás de este tipo de métodos es modificar el algoritmo PH para que acepte procesar una cantidad fija c menor al total de escenarios por iteración. En este sentido, existen dos métodos para elegir los escenarios a procesar por cada iteración: el método de bloque-cíclico ingenuo y el método de bloque-cíclico con los escenarios de menor convergencia.

5.2.1. Método de bloque-cíclico ingenuo

La idea detrás de este método es que cada iteración resuelva los siguientes c escenarios desde donde quedó la iteración anterior. Por ejemplo si se tienen 9 escenarios y se desean procesar de a 6, entonces la primera iteración procesa los escenarios 1, 2, 3, 4, 5 y 6. Luego se realizan las actualizaciones pertinentes del algoritmo y se pasa a la segunda iteración, la cual procesa los escenarios 7, 8, 9, 1, 2 y 3, y así sucesivamente hasta que se alcance la convergencia.

5.2.2. Método de bloque-cíclico con los escenarios de menor convergencia

Este método intenta mejorar el anterior decidiendo de una manera más inteligente los c escenarios a resolver por iteración. La idea es que la primera iteración resuelva el total de los escenarios. Luego se realizan las actualizaciones PH y se calcula una medida de convergencia para cada uno de los escenarios. Luego todas las iteraciones consideran los c escenarios que menos convergieron en la iteración anterior, se calculan las actualizaciones PH y se recalculan las medidas de convergencia de cada escenario.

Y así sucesivamente.

Ambos métodos descritos, según Somervell [35], son capaces de converger a la solución global del problema. Sin embargo, rara vez convergen cuando la cantidad de escenarios resueltos por iteración es menos de la mitad del total de escenarios. Es por ello que propone que para mejorar la velocidad de convergencia se tome un número cercano por arriba a la mitad de los escenarios.

5.3. Método del nodo de menor convergencia

Este método es similar al método de bloque cíclico con los escenarios de menor convergencia con la diferencia que en vez de calcular una medida de convergencia por escenario ésta se calcule para cada nodo. De este modo los escenarios a resolver en cada iteración corresponden a todos los escenarios que tengan relación con el nodo cuya medida de convergencia sea mayor.

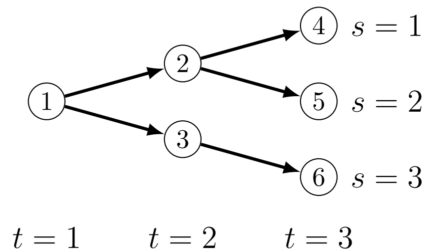


Figura 5.2: Árbol de 3 periodos y 3 escenarios.

Por ejemplo suponiendo en la figura 5.2 el nodo de menor convergencia en una iteración corresponde al nodo 1, entonces los escenarios 1, 2 y 3 serán resueltos en la siguiente iteración. Si por el contrario el nodo 3 es el de menor convergencia entonces sólo el escenario 3 será resuelto en la siguiente iteración.

Considerando el problema con los métodos de bloque-cíclicos, es posible que el algoritmo no converja con este método ya que puede ocurrir que el nodo de menor convergencia para una iteración tenga relación con menos de la mitad de los escenarios totales del problema.

5.4. Método de los escenarios que no han convergido

Este método es similar al método de bloque-cíclico con los escenarios de menor convergencia sólo que la cantidad de escenarios a resolver en cada iteración es un número variable que comienza con todos los escenarios y va cambiando para considerar sólo los escenarios que no han llegado a un nivel de convergencia igual a la tolerancia de convergencia del algoritmo dividida por el número de escenarios. Si no hay escenarios que cumplan la norma entonces el algoritmo debe haber convergido.

Dependiendo del problema este método puede llegar a obtener una ganancia de hasta 3, mientras que en otros casos la velocidad empeora.

5.5. Método completamente asíncrono

Este método intenta aprovechar la paralelización haciendo que se calculen las actualizaciones PH cada vez que un sub-problema sea resuelto. De este modo cada vez que un nodo esclavo le devuelve los resultados de un escenario al nodo maestro, éste realiza las actualizaciones y revisa el nivel de convergencia. Si aún no se alcanza, entonces se sigue iterando con el siguiente escenario.

Ya que este es un caso específico del método de bloque-cíclico ingenuo en donde la cantidad fija es 1, entonces también posee sus problemas. Dentro de éstos está el hecho que no converge incluso en problemas pequeños.

5.6. Método combinado

El método combinado es una combinación entre el método de los escenarios que no han convergido y el método asíncrono, considerando además procesar al menos la mitad de los escenarios por iteración. De este modo este método debiese converger (por procesar más de la mitad de los escenarios por iteración), debiese tomar menos tiempo en converger (al procesar sólo algunos escenarios) y debiese disminuir el tiempo en comunicación debido al procesamiento asíncrono de los escenarios.

Somervell [35] en su trabajo recomienda probar este método en problemas que utilicen Progressive Hedging.

5.7. Mejora en el tiempo

Si se quiere saber cuánto es la mejora en el tiempo del algoritmo PH en el problema minero primero se debe conocer la arquitectura del sistema en donde se procesará además de escoger la mejor metodología de paralelización.

El computador sobre el cual se trabajará corresponde a un sistema clúster de memoria distribuida en donde existen un nodo o procesador maestro (nodo central en la figura 5.3) más 8 nodos esclavos (nodos periféricos en la figura 5.3). Cada uno de estos nodos corresponde a un procesador Intel Xeon de 2,4 GHz con 8 núcleos. Además, el nodo maestro cuenta con 24 GB de memoria RAM mientras que cada nodo esclavo cuenta con 48 GB de memoria RAM. Todos los nodos se encuentran conectados mediante una red Gigabit, los cuales comparten un disco duro de 2,2 TB de capacidad.

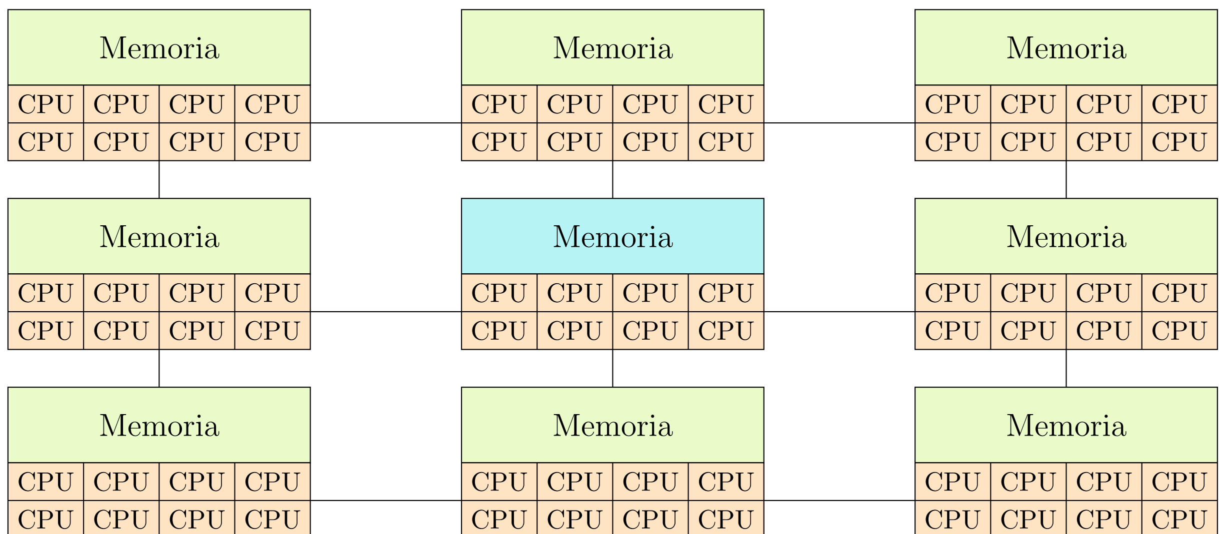


Figura 5.3: Sistema clúster con el que se cuenta. El nodo central corresponde al nodo maestro mientras que los nodos periféricos corresponden a los nodos esclavos.

El mejor método de paralelización encontrado corresponde al método combinado, en donde la cantidad óptima de escenarios a resolver por iteración depende mucho tanto del problema como de los datos asociados. Debido a esto, el estudio analítico de la ganancia se realizará sólo en función del método asíncrono (versección 5.1.2), ya que este se diferencia del primero solamente en que considera la totalidad de los escenarios.

La ganancia

Para obtener la ganancia de paralelizar el algoritmo completo para el problema de planificación minera de largo plazo es necesario estudiar cada etapa de éste. Para ello primero se introduce la nomenclatura a utilizar y luego se detalla etapa por etapa el algoritmo y su paralelización.

Cabe destacar que la implementación del algoritmo está desarrollada en el software de modelamiento GAMS y es resuelto por el *solver* CPLEX12.

5.7.1. Nomenclatura

- n: número total de núcleos.
- m: número de nodos del clúster (maestro + esclavos).
- ESC: conjunto de escenarios.
- $ITER$: conjunto de iteraciones PH.
- S: algún tipo de instancia caracterizada por los datos involucrados (períodos, fases, bancos, productos, capacidades, etc.).
- $S_{i,j}$: instancia con los datos del escenario $i \in ESC$ en la iteración $j \in ITER$.
- $g(S_{i,j})$: tiempo que toma resolver la instancia $S_{i,j}$.
- H_j: $g(S_{i^*,j})$ tal que $g(S_{i^*,j}) \geq g(S_{i,j})$ $\forall i \in ESC$, para la iteración $j \in ITER$. En palabras más simples, corresponde al tiempo máximo de resolución de un escenario en la iteración j.
- $T_j(n)$: Tiempo que demora la etapa de resolución del algoritmo paralelizado con n núcleos en la iteración j.
- A_j: Tiempo que demora la etapa de actualización del algoritmo en la iteración j.

5.7.2. Inicialización del modelo determinista

La inicialización del modelo determinista corresponde a la definición de las variables, las ecuaciones, los parámetros y la función objetivo, entre otros elementos dentro del programa de modelamiento. Esta inicialización es ínfima en términos de tiempo de procesamiento ya que toma menos del 0,01 % del tiempo total que demora el programa completo. Aunque es posible llevar a cabo una paralelización en esta etapa, lo poco que se gana no amerita el esfuerzo requerido para lograrlo.

5.7.3. Iteración inicial PH

La iteración inicial (resolución de cada escenario por separado sin forzar su convergencia) se puede descomponer en tres parte: inicialización, resolución y sincronización.

Inicialización

En esta etapa se definen todas las variables, ecuaciones, etc. asociadas al algoritmo PH. Al igual que en el caso de la inicialización del modelo determinista, el tiempo que demora esta etapa es irrelevante.

Resolución

En esta etapa es donde ocurre la primera resolución de los escenarios. Ésta es una de las etapas importantes, relevantes y en donde la paralelización debiese focalizarse. Gran parte del trabajo ya viene realizado puesto que cada escenario trabaja con sus propias variables y parámetros por lo que sólo basta distribuir el procesamiento en los distintos procesadores.

En el caso secuencial, el algoritmo demora un tiempo $T_0(1)$ equivalente a:

$$T_0(1) = \sum_{i \in ESC} g(S_{i,0}) \tag{5.1}$$

Si se paraleliza el trabajo de manera asíncrona en n núcleos, tal como se mostró en la figura 5.1 (b), entonces en el mejor caso el algoritmo demoraría idealmente un tiempo $T_0^{mejor}(n)$ equivalente a:

$$T_0^{mejor}(n) = \frac{\sum_{i \in ESC} g(S_{i,0})}{n} \tag{5.2}$$

A su vez, en el peor caso el algoritmo demoraría idealmente un tiempo $T_0^{peor}(n)$ equivalente a:

$$T_0^{peor}(n) = \frac{\sum_{i \in ESC} g(S_{i,0}) - H_0}{n} + H_0 \tag{5.3}$$

$$= \frac{\sum_{i \in ESC} g(S_{i,0}) + (n-1) * H_0}{n} \tag{5.4}$$

donde H_0 corresponde al tiempo máximo de resolución de un escenario en la iteración inicial PH.

En palabras simples, en el mejor caso la resolución de la iteración inicial en paralelo sería de forma tal que todos los núcleos del computador procesasen todo el tiempo, sin existir tiempo ocioso en ninguno de los núcleos.

Para el peor caso, el tiempo total de la iteración inicial paralelizada es igual al mejor caso del tiempo en paralelo considerando todos los escenarios menos el que demore más en resolverse, más el tiempo de resolución de este último. De esta forma se resolverían $I-1$ escenarios con todos los núcleos funcionando al 100 %, y se resolvería el escenario más largo con $n-1$ núcleos esperando a que el otro termine.

En un caso real sin embargo, es necesario agregar un término adicional para incluir los costos asociados al overhead debido al problema de escalabilidad que poseen los sistemas de memoria compartida. Este término claramente depende de cuántos procesadores intenten acceder a la misma memoria (ie, pertenecen al mismo nodo). En estricto rigor debiese existir un término asociado a cada uno de los nodos de trabajo del clúster, ya que podría darse el caso de que un nodo esté trabajando con sus 8 procesadores mientras que otro esté trabajando sólo con 1 ya que no existen más escenarios a procesar. Sin embargo, como se trata del peor caso, se considerará que todos los nodos estarán trabajando al máximo todo el tiempo.

De esta forma, la ecuación 5.4 quedaría finalmente como:

$$T_0(n) = \alpha_0(\frac{n}{m}) * \frac{\sum\limits_{i \in ESC} g(S_{i,0}) + (n-1) * H_0}{n} \tag{5.5}$$

en donde la función $\alpha_0(n/m)$ corresponde al costo de overhead y depende específicamente del sistema utilizado. En la sección 6 se implementa esta metodología con el fin de analizar el comportamiento de la curva en un sistema real.

Sincronización de datos

Finalmente en la etapa de sincronización de datos el nodo maestro debe recolectar los resultados de las resoluciones de los distintos escenarios obtenidas por los distintos procesadores. GAMS en particular lo que hace es que una vez termina de resolver un problema guarda los resultados en un archivo y le avisa al nodo maestro que ya terminó para que le sea asignado un nuevo escenario.

Para entonces llevar a cabo la sincronización de datos, primero el nodo maestro debe establecer un punto de barrera de modo que sólo se realice una vez que todos los nodos hayan terminado su trabajo.

Debido a la naturaleza de esta etapa, su paralelización es imposible. Lo que sí se podría hacer es que el nodo maestro almacene los resultados de un escenario entregados por un procesador simultáneamente con el envío del siguiente escenario hacia

dicho procesador. Sin embargo se debe tener especial cuidado pues se puede terminar perjudicando el tiempo global al aumentar el overhead de información durante el procesamiento.

5.7.4. Preparación PH

Al igual que en el caso de la inicialización del modelo determinista, en esta etapa lo que se hace es definir algunos parámetros específicos de PH y así dejar listo el modelo para su posterior resolución. Por lo mismo esta etapa también es pequeña en términos de tiempo y su paralelización traería mas complicaciones que mejoras.

5.7.5. Iteraciones PH

Al igual que la iteración inicial, la etapa de iteraciones PH se puede descomponer en 5 etapas: inicialización, actualización, resolución, sincronización y cierre. Las 3 etapas del medio se repiten tantas veces como iteraciones PH se realicen. Si por ejemplo sólo se itera 2 veces el algoritmo, entonces la secuencia de las etapas sería: inicialización $\rightarrow$ actualización $\rightarrow$ resolución $\rightarrow$ sincronización $\rightarrow$ actualización $\rightarrow$ resolución $\rightarrow$ sincronización $\rightarrow$ cierre.

Inicialización

En la etapa de inicialización lo que se hace es configurar algunas opciones del solver además de setear variables de control para las posteriores iteraciones. Al igual que en los casos anteriores, esta etapa también es pequeña en términos de tiempo de resolución por lo que paralelizarla traería una ganancia marginal respecto al tiempo total del programa.

Actualización

En la etapa de actualización de las iteraciones PH el modelo lo que hace es calcular el coeficiente de penalización de la función objetivo según los resultados obtenidos en la iteración anterior. En caso de que se logre llegar a un nivel de convergencia deseado entonces el algoritmo para y pasa directo a la etapa de cierre.

La carga de información en esta etapa es súper grande, ya que debe realizar cálculos para cada escenario y cada variable involucrada en el problema. Si se agrega un escenario adicional al problema entonces eso significa que se adiciona 1 variable por cada una de las variables del modelo al problema global. Si por otra parte se agrega una variable

al modelo entonces en el problema esto se ve reflejado como 1 nueva variable por cada escenario que lo compone.

Sin embargo su tiempo de resolución no es muy elevado puesto que, a pesar de que la cantidad de datos es importante, los cálculos realizados con dichos datos son relativamente simples. No obstante el tiempo involucrado en la etapa no es tan despreciable como en las etapas de inicialización, sincronización o cierre.

La paralelización en esta etapa es bastante compleja. Aunque se puede dividir la etapa en sub-etapas, dependiendo de los datos utilizados, cada una de ellas requiere que todas las sub-etapas anteriores hayan sido resueltas. Esto hace que la paralelización sólo pueda ser a nivel de estas sub-etapas y, por ende, su beneficio sea marginal respecto al tiempo total de la etapa.

Al tiempo que demora esta etapa se le denotará como A_j, en donde j corresponde a la iteración cuyos datos son utilizados. La primera actualización utiliza los datos de la iteración inicial, por lo que a dicho tiempo se le denotará como A_0. Para el resto de las iteraciones los resultados utilizados son los de la iteración anterior, por lo que se realizarán tantas actualizaciones como resoluciones existan en las iteraciones PH.

Resolución

La etapa de resolución de las iteraciones PH es idéntica a la etapa de resolución de la iteración inicial. La única diferencia es que, a nivel de modelo, la función objetivo tiene un término de penalización que va cambiando iteración tras iteración. Debido a esto es que se pueden considerar las mismas ecuaciones desarrolladas anteriormente.

De esta forma, la el algoritmo secuencial para la iteración j demora un tiempo $T_j(1)$ equivalente a:

$$T_j(1) = \sum_{i \in ESC} g(S_{i,j}) \tag{5.6}$$

Si por otra parte se paraleliza el trabajo de manera asíncrona entonces en el mejor caso el algoritmo demoraría idealmente un tiempo $T_j^{mejor}(n)$ equivalente a:

$$T_j^{mejor}(n) = \frac{\sum_{i \in ESC} g(S_{i,j})}{n} \tag{5.7}$$

A su vez, en el peor caso el algoritmo demoraría idealmente un tiempo $T_j^{peor}(n)$ equivalente a:

$$T_j^{peor}(n) = \alpha_j(\frac{n}{m}) * \frac{\sum_{i \in ESC} g(S_{i,j}) + (n-1) * H_j}{n} \tag{5.8}$$

Al igual que en la etapa de resolución de la iteración inicial, la función $\alpha_j(\frac{n}{m})$ depende tanto del sistema como de la iteración del algoritmo. Esto es debido a que en distintas iteraciones el término de penalización puede ser diferente tanto en tamaño como en valores, lo que hace que la cantidad de recursos utilizados por el programa de optimización sean variables entre iteraciones.

La intuición dice que este término debiese ser una función cuyo crecimiento en $\frac{n}{m}$ es mayor en contraste con la función de costos de overhead de la ecuación 5.5. Esto es debido a que la función α_j debe lidiar con un término cuadrático adicional en la función objetivo.

Sincronización

Al igual que en la etapa de sincronización de datos de la iteración inicial del algoritmo, en esta etapa se recolectan los resultados de los distintos escenarios desde los archivos que genera GAMS por cada escenario resuelto.

Dado que es necesario contar con los resultados de todos los escenarios, se debe establecer un punto de barrera antes de esta etapa para esperar que la etapa de resolución termine en su totalidad.

Como en esta etapa lo que básicamente se hace es almacenar los resultados de todos los escenarios en la matriz de resultados, su paralelización es imposible. Al igual que la etapa homóloga de la iteración inicial se podría dividir esta etapa de modo que cada vez que un escenario termine su procesamiento sus resultados sean almacenados inmediatamente. De esta forma se evitaría tener que hacer toda la sincronización al final de la iteración. El problema radica en que el overhead de información debido a esto podría resultar en un peor tiempo de resolución.

Cierre

Finalmente en la etapa de cierre se realizan los últimos cálculos de convergencia del algoritmo y la exportación de los resultados para su posterior análisis. En esta etapa la paralelización es tanto difícil como de poca utilidad.

5.7.6. Resultado

Considerando las ecuaciones 5.1 y 5.6, el tiempo relevante que demora el algoritmo completo de manera secuencial se resume en la ecuación 5.11.

$$T(1) = T_0(1) + A_0 + \sum_{(j \in ITER)} (T_j(1) + A_j) \tag{5.9}$$

$$T(1) = \sum_{(i \in ESC)} g(S_{i,0}) + A_0 + \sum_{(j \in ITER)} \left(\sum_{(i \in ESC)} g(S_{i,j}) + A_j \right) \tag{5.10}$$

$$T(1) = \sum_{(j \in \{0\} \cup ITER)} \left(\sum_{(i \in ESC)} g(S_{i,j}) + A_j \right) \tag{5.11}$$

Por otra parte si se consideran las ecuaciones 5.5 y 5.8, el tiempo relevante que demora el algoritmo completo de manera paralela según el peor caso del método asíncrono (ver sección 5.1.2) se resume en la ecuación 5.14.

$$T(n) = T_0(n) + A_0 + \sum_{(j \in ITER)} (T_j(n) + A_j) \tag{5.12}$$

$$\begin{aligned} T(n) = {} & \alpha_0(\frac{n}{m}) * \frac{\sum_{i \in ESC} g(S_{i,0}) + (n-1) * H_0}{n} + A_0 \\ & + \sum_{(j \in ITER)} \left(\alpha_j(\frac{n}{m}) * \frac{\sum_{i \in ESC} g(S_{i,j}) + (n-1) * H_j}{n} + A_j \right) \end{aligned} \tag{5.13}$$

$$T(n) = \sum_{(j \in \{0\} \cup ITER)} \left(\alpha_j(\frac{n}{m}) * \frac{\sum_{i \in ESC} g(S_{i,j}) + (n-1) * H_j}{n} + A_j \right) \tag{5.14}$$

5.7.7. Análisis

Según lo visto en la sección 3.4.6, la ganancia máxima posible de obtener con la paralelización del algoritmo viene dada por la fracción del programa paralelizable y la fracción no paralelizable. Para el caso del algoritmo PH en el problema de planificación minera de largo plazo, la parte paralelizable del programa corresponde a los términos

T_j de la ecuación 5.12 mientras que la parte no paralelizable corresponde a los términos A_j.

Al aumentar la cantidad de escenarios a considerar, los términos T_j comienzan cada vez a tomar mayor peso simplemente por el hecho de que para cada iteración se deben considerar una mayor cantidad de ellos. Adicionalmente los términos A_j también se ven incrementados pues, por cada escenario, se debe hacer un procesamiento adicional de convergencia.

Debido a esto, y considerando que el tiempo porcentual que demora cada una de estas partes está condicionado tanto por los datos como por las series de precios, dar una respuesta conclusiva de la ganancia máxima al paralelizar es imposible.

Por otra parte, la ley de Gustafson (ver sección 3.4.6) propone aumentar el tamaño del problema de modo que los términos T_j comienzen a opacar cada vez más a los términos A_j. De esta forma, la ganancia máxima posible podría mejorar considerablemente además de resolver problemas quizás mucho más atractivos.

Finalmente, el ahorro obtenido por la paralelización del algoritmo viene dado por la ecuación 5.15.

$$Ahorro(n) = \frac{T(1) - T(n)}{T(1)} \tag{5.15}$$

Capítulo 6

Implementación

A partir de lo expuesto en la sección anterior, en este capítulo se presenta una primera aproximación a la paralelización de la iteración inicial de PH para el problema de planificación minera de largo plazo.

Se considera sólo la iteración inicial del algoritmo debido a que, aunque la diferencia en su implementación es mínima, el tiempo necesario para llevar a cabo las pruebas exceden al tiempo disponible para el desarrollo de este trabajo.

6.1. GAMS

Actualmente el problema minero se encuentra desarrollado en GAMS (*General Algebraic Modeling System*), un software especialmente diseñado para modelar problemas matemáticos de optimización lineal, no lineal y entero mixto. Adicionalmente se utiliza el software de optimización CPLEX, del cual GAMS se preocupa de entregarle el problema y recuperar la solución que éste entrega.

6.1.1. Principales características

Una de las principales características que hace GAMS interesante dentro del marco de los objetivos de este trabajo es su capacidad de funcionamiento tanto en sistemas Windows como en sistemas UNIX, estos últimos utilizado en muchos de los súper-computadores de la actualidad. Adicionalmente existen dos funcionalidades que pueden ser aprovechadas para el procesamiento de un problema en paralelo: *Solving steps* y *Save & Restart*.

Solving steps

Al momento en que aparece dentro del código un llamado a resolver el problema matemático de optimización, GAMS separa la resolución de éste en tres pasos básicos: la generación, la resolución y la actualización.

1. **La generación.** El sistema inicializa el modelo utilizando las ecuaciones simbólicas definidas, considerando la base de datos del instante en que se llama la resolución. Esta instancia contiene toda la información y servicios requeridos para que el solver pueda resolverlo.
2. **La resolución.** La instancia anterior es enviada a un subsistema para que resuelva el problema.
3. **La actualización.** El subsistema de resolución entrega los resultados y estadísticas del modelo, actualizando la base de datos de GAMS.

En la mayoría de los casos, el tiempo que demoran la primera y tercera etapa son despreciables en comparación con el tiempo que demora la segunda etapa. Por otra parte, cada una de estas etapas pueden ser controladas de manera casi independiente, logrando así que el código que llama a resolver el problema pueda desligarse de la resolución de éste y así pueda continuar con el procesamiento de otros datos y eventualmente llamar a resolver otros modelos.

Save & Restart

Otra funcionalidad que incorpora GAMS, la cual es accesible desde la línea de comandos al momento de invocar el programa, corresponde a poder guardar y reanudar un programa en cualquier punto de su ejecución. Al hacer esto, la ejecución del programa no difiere ni en tiempo ni en resultados en comparación con su ejecución de modo secuencial.

6.2. Paralelización

A continuación se presentan dos formas de paralelizar el modelo estocástico con PH. La primera es utilizando sólo herramientas que posee GAMS, particularmente la característica de Solving steps, mientras que la segunda usa a Java como lenguaje de control de GAMS en conjunto con la funcionalidad de Save & Restart, accediendo así a funciones y metodologías más especializadas para el trabajo *multi-threading*.

6.2.1. Paralelización GAMS

El problema estocástico de planificación minera de largo plazo, de aquí en adelante problema estocástico, posee sólo una variable estocástica para fines de este trabajo: el precio. Si se le aplica a este problema el algoritmo PH entonces se deben resolver tantos problemas determinísticos de planificación minera de largo plazo, de aquí en adelante problema determinístico, como escenarios de precios se consideren, para cada una de las iteraciones PH.

El pseudo-código 1 muestra de manera simplificada cómo sería el código en GAMS de la iteración inicial PH para el problema estocástico. Para cada uno de los escenarios se setea el precio del modelo determinista en el precio del escenario actual y luego se envía a resolver. A continuación, y una vez terminada su resolución, se procede a guardar los resultados del modelo determinista en la columna i de la matriz de resultados.

Pseudo-código 1 PH secuencial

```
1: for all i ∈ ESC do
2:     p_modelo ← p_i                                       ▷ Setear el precio del modelo
3:     SOLVE modelo using MIQCP maximizing FObj             ▷ Mandar al solver
4:     resultados(i) ← FObj.l                               ▷ Guardar resultados
5: end for
```

Al momento de llamar a la función *SOLVE*, GAMS ejecuta los *solving steps* de manera consecutiva y sin devolverle el control al programa principal hasta que se termine. De esta forma al momento de guardar los resultados se sabe con certeza que el solver terminó y efectivamente existen datos de la solución.

Tal como se mencionó en la sección 6.1.1, GAMS permite tratar cada una de las etapas de resolución de manera independiente. En otras palabras es posible mandar a resolver todos los escenarios de manera paralela. Luego, y después que todos los escenarios terminen, proceder a recolectar sus resultados.

El pseudo-código 2 muestra cómo sería el código GAMS si lo que se quiere es separar las órdenes de envío al solver y las de recolección de resultados. Esto se logra poniendo la opción *solvelink* en 3 y teniendo un arreglo $handler(i)$ que sepa ubicar al proceso asociado al escenario i.

A continuación se escribe un loop de resolución, el cual setea el precio del modelo al del escenario que se resolverá, envía el modelo al solver y guarda el *handle* del modelo dentro del arreglo. El bloque for no se detiene hasta que termina de enviar al solver todos los escenarios dentro del conjunto ESC.

Del mismo modo se escribe un loop de recolección, el cual recupera el estado del modelo y su solución, la cual se guarda en la matriz de resultados. Dado que para

poder obtener la solución de un escenario es necesario que éste haya terminado, se deben generar métodos que prevengan intentar recolectar la solución de un modelo cuyo procesamiento aún no ha terminado. La forma más simple es mediante un $sleep(t)$, función que pausa o duerme la ejecución del programa durante t segundos.

Pseudo-código 2 PH en paralelo (GAMS)

1: $modelo.solvelink \leftarrow 3$ ▷ Opción para resolver en paralelo
2:
3: **for all** $i \in ESC$ **do** ▷ Loop de resolución
4: $p_{modelo} \leftarrow p_i$
5: SOLVE *modelo* using MIQCP maximizing $FObj$
6: $handler(i) \leftarrow modelo.handle$
7: **end for**
8:
9: $sleep(t)$ ▷ Esperar t segundos
10:
11: **for all** $i \in ESC$ **do** ▷ Loop de recolección
12: $modelo.handle \leftarrow handler(i)$
13: $execute_loadhandle\ modelo;$
14: $resultados(i) \leftarrow FObj.l$
15: **end for**

GAMS provee funciones adicionales para verificar el estado de resolución de un modelo. De esta forma se puede mejorar el código anterior haciendo que en vez de sólo hacer un *sleep* largo se hagan varios cortos, y entre cada uno se le pregunte a cada modelo si ha terminado y así saber si se puede o no recuperar los resultados.

La paralelización propuesta se puede apreciar en el pseudo-código 3. La idea general es mantener el ciclo del *repeat* hasta que se terminen de resolver y guardar los resultados de todos los escenarios.

El loop de resolución debe enviar tantos escenarios como threads n se deseen. En caso de que se tengan todos los threads funcionando entonces se debe detener el loop de resolución ya que no pueden enviarse más escenarios a resolver hasta que el loop de recolección guarde la solución de al menos un modelo más. Como tampoco se deben enviar a resolver modelos que ya fueron enviados, la línea 13 hace que se salten dichos escenarios.

El loop de recolección por su parte debe primero que todo buscar los modelos de los escenarios que ya hayan sido enviados a resolver ($ordinal(i) \leq n_ultimoenviado$) pero que aún no se hayan recuperado sus resultados ($Omitir(i) = False$). En el caso que el escenario haya terminado de procesar ($(handler(i)) = 2$) se debe guardar la solución en la matriz de resultados y actualizar las variables de control de la paralelización.

Pseudo-código 3 PH en paralelo (GAMS)

```
 1: modelo.solvelink ← 3                                ▷ Opción para resolver en paralelo
 2:
 3: n                                                   ▷ Número de threads
 4: n_listos ← 0                                        ▷ Número de escenarios listos
 5: n_procesando ← 0                                    ▷ Número de escenarios procesándose
 6: n_ultimoenviado ← 0                                 ▷ Último escenario enviado
 7: Omitir(i) ← False                                   ▷ True cuando se recupere su solución
 8:
 9: repeat
10:     for i ∈ ESC do                                  ▷ Loop de resolución
11:         if n_procesando ≥ n then
12:             end for
13:         else if ordinal(i) ≤ n_ultimoenviado then
14:             continue
15:         else
16:             n_procesando ← n_procesando + 1
17:             n_ultimoenviado ← n_ultimoenviado + 1
18:
19:             p_modelo ← p_i
20:             SOLVE modelo using MIQCP maximizing FObj
21:             handler(i) ← modelo.handle
22:         end if
23:     end for
24:
25:     sleep(5)                                        ▷ Esperar 5 segundos
26:
27:     for i ∈ ESC do                                  ▷ Loop de recolección
28:         if ordinal(i) ≤ n_ultimoenviado and !Omitir(i) then
29:             if handlestatus(handler(i)) = 2 then
30:                 modelo.handle ← handler(i)
31:                 execute_loadhandle modelo
32:                 resultados(i) ← FObj.l
33:
34:                 Omitir(i) ← True
35:                 n_listos ← n_listos + 1
36:                 n_procesando ← n_procesando − 1
37:             end if
38:         end if
39:     end for
40: until k = cardinal(ESC)
```

6.2.2. Paralelización Java

Otra forma de abordar la paralelización del modelo estocástico con PH consiste en aprovechar la mayor cantidad de métodos y funciones que entrega el lenguaje Java para el trabajo con programas *multi-threading*. La idea central es que un programa Java sea quien controle qué hace GAMS, diciéndole qué escenarios se resuelven y cuándo. Esto se logra utilizando la funcionalidad de Save & Restart.

Tal como se explicó en la sección 5.7.3, la iteración inicial se compone de una primera etapa de inicialización, una segunda etapa de resolución y una tercera etapa de sincronización. Para hacer que Java sea capaz de paralelizar se deben trabajar de forma separada cada una de estas etapas.

Para comenzar Java debe invocar al programa GAMS asociado a la inicialización y pedirle que una vez que finalice su ejecución este guarde el estado de dicho programa. De esta forma se puede reanudar este estado para continuar con la segunda etapa de resolución.

En la segunda etapa debe existir un cambio estructural en el programa GAMS: cada vez que se invoque su ejecución este debe resolver sólo 1 escenario, que está determinado de antemano, para así darle pie a Java para que sea quien controle qué escenario se resuelve y cuándo.

Finalmente se lleva a cabo la etapa de sincronización, en donde la tarea fundamental corresponde a guardar las soluciones de todos los escenarios dentro de la matriz de resultados. Para ello se debe leer por cada escenario un archivo que contiene su solución, el cual se genera en la etapa de resolución al finalizar la ejecución del solver para cada escenario.

El programa Java por su parte se compone de dos partes: el programa principal (pseudo-código 4) y los threads que éste crea (pseudo-código 5).

El programa principal comienza llamando, mediante la consola de comandos del sistema operativo, al programa *gams.exe* para que ejecute el archivo *inicializacion.gms* y guarde su estado al finalizar con el nombre *inicializacion*.

A continuación se crean tantos threads como se requieran y se comienza con la resolución de los escenarios. El objeto *CountDownLatch* lo que hace es establecer un contador equivalente al número de threads que se hayan creado. Luego, con la función *latch.await()*, se pausa el programa principal hasta que el contador del *latch* llegue a cero. Una vez que esto ocurra el programa principal se reanuda y se procede a ejecutar la etapa de sincronización. El parámetro *r=inicializacion* le dice a GAMS que debe reanudar el programa desde el estado que dejó guardado la etapa de inicialización.

Pseudo-código 4 Programa Java: Programa principal

```
1:                                          ▷ Ejecutar la etapa de inicialización
2: exec gams.exe inicializacion.gms s=inicializacion
3:
4: for i ← 0, N_threads − 1 do              ▷ Mandar a ejecutar N_threads threads
5:     new Thread().run()
6: end for
7:                                          ▷ Esperar que todos los threads terminen
8: latch ← new CountDownLatch(N_nucleos)
9: latch.await()
10:                                         ▷ Ejecutar la etapa de sincronización
11: exec gams.exe sincronizacion.gms r=inicializacion
```

Cada thread de java por su parte comienza consiguiendo el siguiente escenario que será resuelto mediante una función *sincronizada* (ver sección 3.4.3). Esto último es necesario ya que se debe evitar que varios threads eventualmente accedan a esta función de manera simultánea, ya que podrían ocurrir errores tales como que se resuelva un escenario más de una vez o que simplemente no se resuelva.

Una vez el thread tiene el siguiente escenario que le corresponde resolver procede a llamar al programa GAMS encargado de la resolución, diciéndole que comience desde el estado guardado por la etapa de inicialización y entregándole el número del escenario que debe resolver. Esto se repite tantas veces cómo sea necesario hasta que no existan más escenarios por resolver. Una vez esto ocurra, se le debe restar uno al contador *latch* del programa principal.

Pseudo-código 5 Programa Java: Thread

```
1: i ← sinchronize siguiente_escenario()    ▷ Obtener siguiente escenario
2:
3: while i < N_escenarios do                ▷ Loop de resolución
4:     exec gams.exe resolucion.gms r=inicializacion –escenario=i    ▷ Resolver escenario
5:     i ← sinchronize siguiente_escenario()    ▷ Obtener siguiente escenario
6: end while
7:
8: latch.countDown()                        ▷ Avisar fin del thread
```

6.3. Resultados numéricos

A continuación se presenta una comparación entre la paralelización gams y la paralelización java, utilizando una instancia relativamente pequeña con 16 escenarios de precios, en donde cada uno de estos demora menos de 60 segundos en resolverse, y un computador con Windows 7, 12 Gb de memoria RAM y un procesador Intel Core i7 920 de 2,67 GHz con 8 núcleos. Cabe recordar que la implementación expuesta considera sólo la iteración inicial del algoritmo PH.

Adicionalmente se muestra el comportamiento de la curva de variación en los tiempos de procesamiento de los escenarios cuando se tienen más de un thread funcionando simultáneamente. En palabras más simples, ésta representa el costo que considera tener varios threads funcionando paralelamente debido a la arquitectura de memoria compartida, ya que como se dijo en la sección 3.4.4 la velocidad de acceso a la memoria disminuye acorde la cantidad de threads que acceden a ella se incrementa, más aún cuando cada thread considera una gran cantidad de datos en su procesamiento. A esta curva se le denominará *curva de overhead*.

Para los gráficos que se muestran a continuación, las curvas se nombran según la forma de paralelización utilizada (Gams o Java) y del número de threads con los que se procesa separados por un guión. El * por su parte denota el no uso de hotstart.

En la sección 6.4 se muestra qué ocurriría si se escalara el problema tanto en el número de escenarios a resolver como en la cantidad de procesadores del computador.

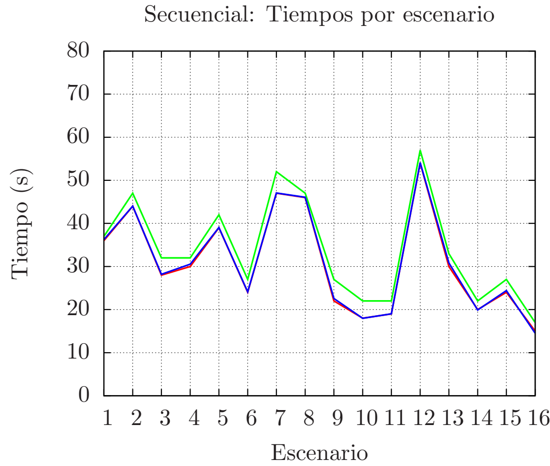
6.3.1. GAMS vs Java

Antes de revisar qué tan buena es la paralelización es necesario saber qué tan eficiente son ambas metodologías en contraste con el programa secuencial original.

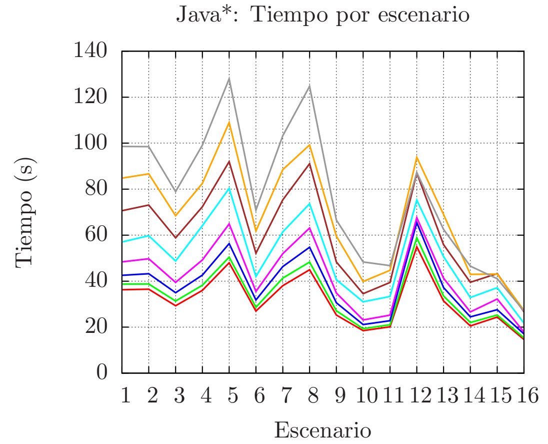
La figura 6.1a muestra qué tan diferentes son los tiempos de resolución para cada escenario según el programa secuencial original, el programa GAMS paralelo con 1 thread y el programa Java paralelo con 1 thread. Para los 3 casos se consideran las mismas condiciones: se resuelven los escenarios en el mismo orden y se utiliza el resultado de un escenario como punto de partida para el siguiente escenario (hotstart).

Para el caso del programa GAMS paralelo con 1 thread, los tiempos de resolución de cada escenario son mayores a los tiempos entregados por el programa secuencial original en aproximadamente un 9 %. Esto se debe principalmente a dos factores: el *repeat* y el *sleep* dentro del código.

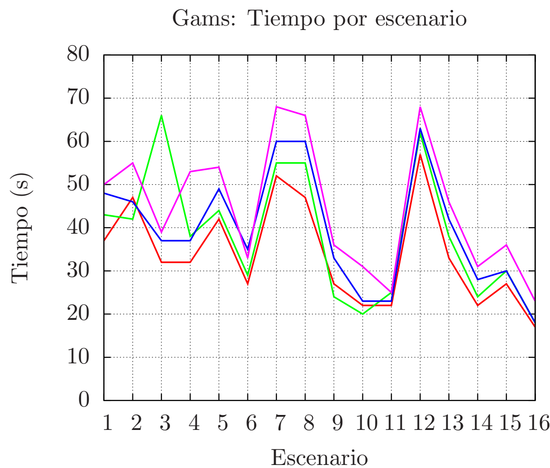
Debido a que GAMS no entrega ningún método que avise cuándo termina de resolver un modelo, es necesario incluir un *repeat* en el código que provoca que el procesador



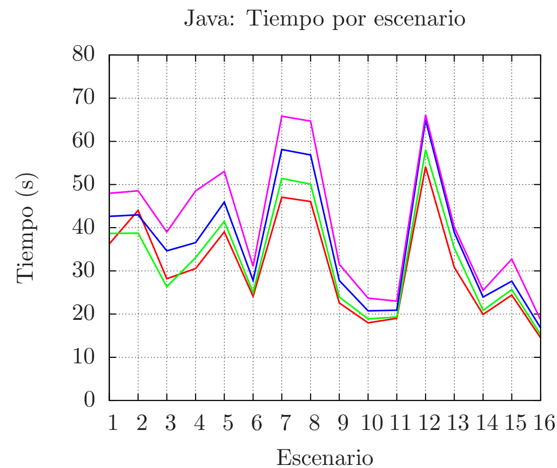
(a)



(b)



(c)



(d)

Figura 6.1: Comparación en los tiempos de resolución de cada escenario según el programa utilizado.
(a) Programa secuencial original vs GAMS paralelo 1 thread vs Java paralelo 1 thread.
(b) Java paralelo *sin* hotstart para distinto número de threads.
(c) GAMS paralelo *con* hotstart para distinto número de threads.
(d) Java paralelo *con* hotstart para distinto número de threads.

no sólo se tenga que preocupar de resolver el problema sino que también de estar constantemente revisando si la resolución ha terminado. De modo de evitar que esta revisión se haga molesta, se incluye un *sleep*. Sin embargo esto también genera un posible problema. Suponiendo se entra en el *sleep* en el instante en que se termina de resolver un escenario, entonces no se recuperará la solución sino hasta que transcurra el tiempo de pausa, lo que finalmente se traduce en que el tiempo de resolución del escenario es incrementado por este tiempo perdido.

Por otra parte, para el caso del programa Java paralelo con 1 thread los tiempos de resolución de los escenarios son prácticamente idénticos al tiempo que demoran en el programa secuencial original. En este caso el problema del *repeat* y el *sleep* es resuelto con el uso de threads y un objeto *latch*. Esto demuestra la mayor eficiencia que entrega tener funcionalidades especializadas para el procesamiento *multi-threading*.

A continuación las figuras 6.1c y 6.1d muestran cómo varían los tiempos de resolución de cada escenario para distinto número de hilos de procesamiento utilizando el programa GAMS paralelo y el programa Java paralelo respectivamente. En ambos casos se utiliza *hotstart*, en donde el punto de partida de un escenario es el resultado del escenario anterior más cercano y que haya terminado. Por ejemplo, si ya se resolvieron los escenarios 1, 2, 3 y 5, y se manda a resolver el escenario 8, entonces éste utilizará como punto de partida la solución del escenario 5.

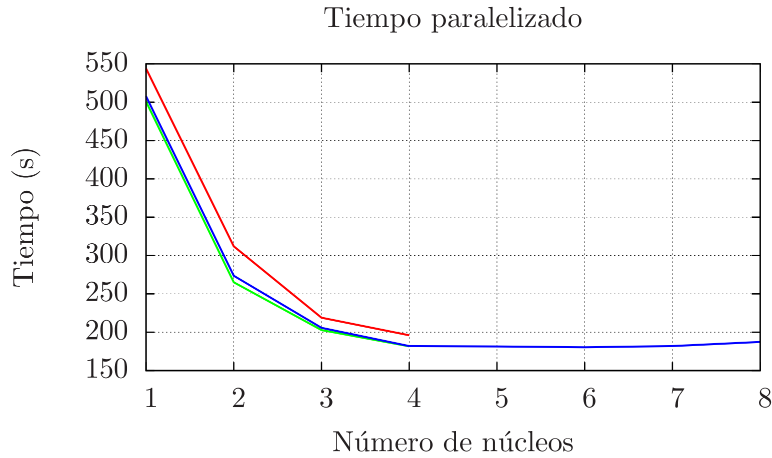
Debido al efecto que puede tener comenzar desde una solución u otra, el tiempo que toma resolver cada escenario puede variar bastante dependiendo del número de *threads* que se utilizan. En la figura 6.1d se puede apreciar que el segundo escenario demora 44 segundos en resolverse para el caso de 1 thread, lo que significa que comienza desde la solución del primer escenario. Sin embargo, cuando se usan 2 o más threads, no existe este hotstart. De hecho, al usar sólo 2 threads el tiempo de resolución del segundo escenario baja a 39 segundos.

Sin embargo, intentando omitir el efecto que produce el hotstart sobre las figuras, los tiempos de resolución para el programa GAMS paralelo son ligeramente superiores a los tiempos de resolución del programa Java paralelo.

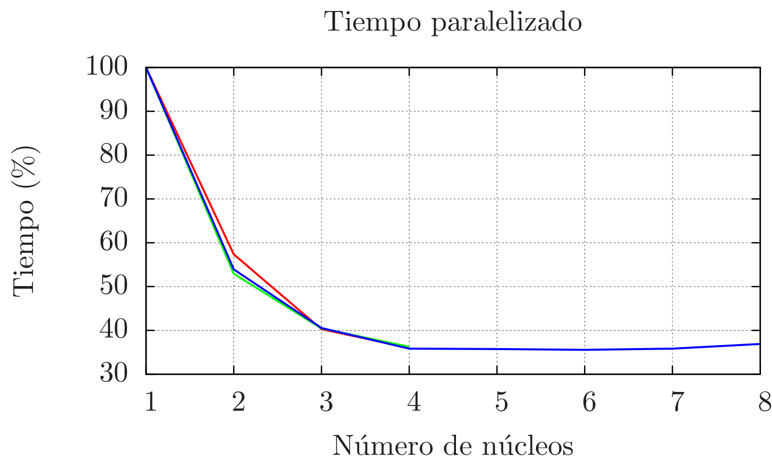
Por otra parte la figura 6.2 muestra cómo varían los tiempos totales de la iteración inicial PH sobre el algoritmo estocástico. En la figura 6.2a se aprecia de mejor manera que el programa GAMS paralelo es más lento que el programa Java paralelo dadas las mismas condiciones. Más aún, el tiempo total de la iteración inicial PH en el programa GAMS paralelo es aproximadamente 9 % mayor al tiempo que toma el programa Java paralelo debido al tiempo adicional que toma cada escenario en resolverse.

Sin embargo la figura 6.2b muestra otro detalle. Tanto en el programa GAMS paralelo como en el programa Java paralelo la reducción en tiempo respecto a su misma versión secuencial es prácticamente idéntica. Esto indicaría que independiente de los

costos constantes que se incurran durante la paralelización del algoritmo, la mejora porcentual en tiempo en relación a su misma versión secuencial será idéntica. De esta forma, la curva de la la figura 6.2a sólo se movería hacia arriba o hacia abajo pero sin cambiar en su forma.



(a)



(b)

Figura 6.2: Comparación en los tiempos de resolución de la iteración inicial según el programa utilizado (GAMS *con* hotstart, Java *con* hotstart y Java *sin* hotstart).
(a) Tiempo total en segundos.
(b) Porcentaje del tiempo respecto al tiempo total secuencial de cada programa.

Para el caso particular de la instancia considerada, el ahorro de tiempo obtenido con la implementación de la paralelización alcanza un 64,45 %. Este resultado sin embargo se ve afectado por el pequeño número de escenarios respecto a la cantidad de threads considerados, debido a que existe una alta probabilidad de que algún núcleo posea un

alto porcentaje de tiempo ocioso. De este modo sería lógico esperar que, al aumentar la cantidad de escenarios considerados por el algoritmo, el ahorro que sería capaz de entregar la paralelización fuese mucho mayor.

6.3.2. La función de overhead

La curva de overhead representa cuánto tiempo porcentual adicional demora la resolución de un escenario dependiendo de la cantidad de threads ejecutándose simultáneamente. Tal y como ya se había adelantado, este *costo de paralelización* se debe principalmente a que los diferentes threads comparten el mismo espacio de memoria (en un sistema de memoria compartida), lo que se traduce en un aumento en el tiempo que demora el acceso a la información en memoria.

Cabe recordar que se utilizó, para todas las pruebas, un computador con un procesador de 8 núcleos y memoria compartida entre ellos.

Para el cálculo de esta función se utilizan los datos del programa Java paralelo *sin* hotstart (ver figura 6.1b). La tabla 6.1 muestra cómo varían porcentualmente los tiempos de resolución de cada escenario cuando se paraleliza desde 1 hasta en 8 threads.

Esc.	Java*-1	Java*-2	Java*-3	Java*-4	Java*-5	Java*-6	Java*-7	Java*-8
1	0,00 %	6,70 %	17,30 %	33,42 %	57,23 %	94,66 %	133,67 %	171,81 %
2	0,00 %	6,11 %	18,40 %	36,24 %	63,69 %	100,20 %	137,25 %	169,75 %
3	0,00 %	6,93 %	19,40 %	34,66 %	66,36 %	100,84 %	133,65 %	169,39 %
4	0,00 %	6,16 %	18,43 %	36,73 %	78,09 %	100,98 %	129,19 %	175,38 %
5	0,00 %	5,20 %	17,43 %	35,21 %	67,51 %	91,78 %	127,23 %	166,86 %
6	0,00 %	6,31 %	17,95 %	31,83 %	55,87 %	93,43 %	129,32 %	162,97 %
7	0,00 %	8,66 %	21,86 %	36,61 %	61,54 %	98,23 %	132,77 %	171,53 %
8	0,00 %	7,50 %	21,84 %	40,49 %	63,90 %	102,67 %	120,87 %	177,58 %
9	0,00 %	7,20 %	21,01 %	37,79 %	60,74 %	91,16 %	135,64 %	163,42 %
10	0,00 %	4,42 %	14,22 %	25,08 %	67,78 %	86,96 %	115,51 %	161,49 %
11	0,00 %	4,43 %	13,35 %	25,29 %	65,72 %	95,69 %	122,10 %	132,37 %
12	0,00 %	7,24 %	19,24 %	23,16 %	37,06 %	58,46 %	70,91 %	58,86 %
13	0,00 %	6,76 %	18,44 %	31,57 %	61,33 %	78,01 %	119,83 %	99,05 %
14	0,00 %	7,25 %	19,12 %	29,47 %	60,03 %	92,41 %	109,22 %	126,50 %
15	0,00 %	3,67 %	13,15 %	32,32 %	52,49 %	77,74 %	76,93 %	68,55 %
16	0,00 %	4,79 %	16,46 %	21,78 %	48,70 %	82,20 %	86,80 %	84,77 %
Prom,	0,00 %	6,21 %	17,97 %	31,98 %	60,50 %	90,34 %	117,56 %	141,27 %
Prom',	0,00 %	6,70 %	19,08 %	35,65 %	64,27 %	97,85 %	130,49 %	170,66 %

Tabla 6.1: Tiempo porcentual adicional que demora cada escenario respecto al tiempo que demora de manera secuencial para el programa Java paralelo *sin* hotstart (Java*-1). El promedio considera los 16 escenarios mientras que el promedio' considera sólo los primeros 8.

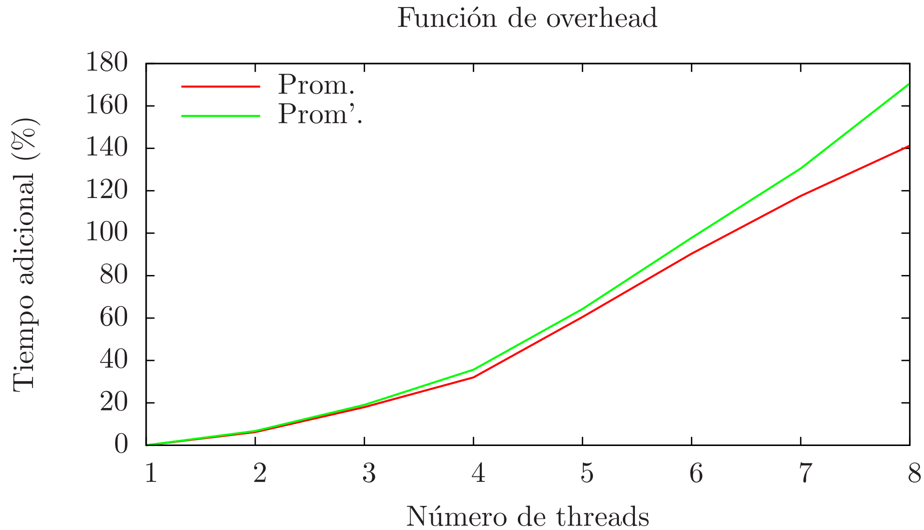


Figura 6.3: Tiempo porcentual promedio adicional que demoran un escenario respecto al tiempo que demora de manera secuencial para el programa Java paralelo *sin* hotstart. El promedio considera los 16 escenarios medidos mientras que el promedio' considera sólo los primeros 8.

Al analizar los datos se puede observar que el tiempo porcentual adicional que toman los primeros escenarios tiende a ser mayor que el que toman los últimos escenarios. Esto se debe al pequeño número de escenarios considerados (16) respecto a la cantidad de núcleos disponibles (8), haciendo que los últimos escenarios se resuelvan en paralelo menos escenarios que threads disponibles. Es por ello que para cada ejecución del programa Java paralelo se calcularon tanto los promedios de todos los escenarios (Prom.) como los promedios de los primeros 8 escenarios (Prom'.). La figura 6.3 muestra el crecimiento que poseen estos porcentajes.

Para la sección de simulación de escenarios (sección 6.4) se desea utilizar una curva de tendencia a partir de los datos anteriores, con el fin de incorporar en el modelo los costos de overhead asociados.

La tabla 6.2 por su parte muestra las curvas de tendencia que siguen tanto la curva Prom. como la curva Prom*., tanto para el caso que el intercepto es cero como para el caso general. Para las cuatro curvas de tendencias generadas se obtiene que el coeficiente de determinación R^2 está sobre el 99 %, siendo mayor para las curvas que consideran sólo los primeros 8 escenarios.

	Curva-1	**Curva-2**	**Curva-3**	**Curva-4**
Curva	Prom.	Prom.	Prom*.	Prom*.
a	0,0233	0,0194	0,0299	0,0287
b	-0,0017	0,0390	-0,0235	-0,0104
c	0	-0,0869	0	-0,0278
R^2	99,15 %	99,35 %	99,88 %	99,89 %

Tabla 6.2: Curvas de tendencia del estilo $ax^2 + bx + c$ que siguen las curvas de la figura 6.3, con x el número de threads menos 1 que se ejecutan en un procesador.

6.4. Simulación de escenarios

Hasta el momento todo el análisis realizado se ha centrado en datos reales en un ejemplo pequeño. La idea ahora es, con dichos datos, simular y evaluar el impacto que tendría la paralelización de la iteración inicial PH en un problema más grande considerando lo expuesto en la sección 5.7 y la ecuación 5.5.

Además, y de modo de evaluar el impacto que genera el balance de cargas en la paralelización del algoritmo (ver 3.4.5) se realizará un análisis adicional para el caso en que se procesan los escenarios ordenadamente según su tiempo de procesamiento. De esta forma se debiese obtener un balance de cargas con una desviación estándar muy pequeña entre cada uno de los núcleos considerados.

Los gráficos que se presentan a continuación se nombran según la notación “xP-yT”, en donde x denota el número de procesadores e y la cantidad de threads por procesador utilizados.

6.4.1. Índices a medir

- Ahorro porcentual obtenido por la paralelización (ahorro).
- Porcentaje de tiempo que cada thread está trabajando (carga).
- La desviación estándar de la cantidad de escenarios que resuelve cada thread (stdev ExN).

6.4.2. Parámetros de entrada

- Número de escenarios (ESC).
- Número de repeticiones para conjunto de parámetros ($REPET$).
- Tiempo medio que toma resolver cada escenario (μ).

- Desviación estándar en el tiempo que toma resolver cada escenario (σ).
- Número de procesadores (p).
- Número de threads por procesador (n).
- Función de overhead ($\alpha(n)$).

El tiempo que demora un escenario en resolverse estará representado por una distribución normal de media μ y varianza σ^2, omitiendo valores de tiempo que sean menores a cero. Además se utilizará la **Curva-3** de la tabla 6.2 para emular el overhead generado por la paralelización, ya que su R^2 es casi igual al mejor caso posible y además considera que en la realidad el intercepto debe valer cero.

De esta forma, el tiempo que toma en resolverse un escenario i cuando se procesan n threads paralelamente (ecuación 6.1) equivale al tiempo aleatorio obtenido por la distribución Normal más un porcentaje adicional descrito por la función de overhead (ecuación 6.2).

$$t_{i,n} = N(\mu, \sigma^2) * (1 + \alpha(n)) \tag{6.1}$$

$$\alpha(n) = 0,0299 * (n-1)^2 - 0,0235 * (n-1) \tag{6.2}$$

6.4.3. Casos de estudio

La tabla 6.3 muestra los distintos casos de estudios considerados en la simulación de escenarios. El caso base considera un número de escenarios equivalente a lo que se espera lograr con la implementación del algoritmo PH en el modelo estocástico. El resto de los casos consideran variaciones unidimensionales de los parámetros del caso base, con el fin de estudiar cómo afectan cada uno de ellos en los resultados finales.

Caso	ESC	μ (s)	σ (s)	Curva de overhead		
				a	b	c
Base	1.500	1.000	250	0.0299	-0.0235	0.0000
100-escenarios	100					
200-escenarios	200					
Media-baja		800				
Media-alta		1.200				
Desviación-baja			50			
Desviación-alta			450			
Curva-baja				0.0075	-0.0059	
Curva-alta				0.1196	-0.0940	

Tabla 6.3: Casos de estudio considerados. Para los casos distintos al base sólo se muestran los parámetros que varían.

Para el caso curva-alta, cada factor de la curva de overhead base (a, b y c) se multiplica por cuatro. Por el contrario, para el caso curva-baja se divide cada factor por 4.

6.4.4. Procedimiento

A continuación se explica a grandes rasgos el procedimiento que sigue el simulador de escenarios.

1. Generar los tiempos de resolución de cada escenario.
2. Se aumentan los tiempos generados según la función de overhead y la cantidad de threads por procesador del sistema.
3. Distribuir cada escenario en cada thread de cada procesador.
4. Calcular índices asociados.
5. Ordenar los escenarios de mayor a menor según su tiempo de resolución.
6. Distribuir cada escenario, según nuevo orden, en cada thread de cada procesador.
7. Calcular índices asociados.
8. Repetir $REPET$ veces desde el punto 1, de modo de obtener promedios estadísticamente representativos.

Este procedimiento se repite variando la cantidad de threads desde 1 hasta 10 y manteniendo fijo los procesadores en 1. Luego se fija la cantidad de threads en la cantidad óptima encontrada (entre 1 y 8) y se varía el número de procesadores desde 1 hasta 9. De esta forma se obtienen resultados acerca de qué ocurre al variar la cantidad de threads por procesador y al variar la cantidad de procesadores.

6.4.5. Resultados

Caso base

La figura 6.4 muestra un resumen de los resultados entregados por el simulador para el caso base. La curva *ideal* considera que todos los procesadores tienen una carga del 100 % todo el tiempo y que no existe una curva de overhead. La curva *paralelo* representa el caso real entregado por el simulador, la curva **Paralelo* muestra qué pasaría si se ordenasen los escenarios en orden decreciente según su tiempo de resolución y luego se procesaran considerando dicho orden. Finalmente la curva ***Paralelo* muestra el peor caso posible, curva descrita por la ecuación 5.5.

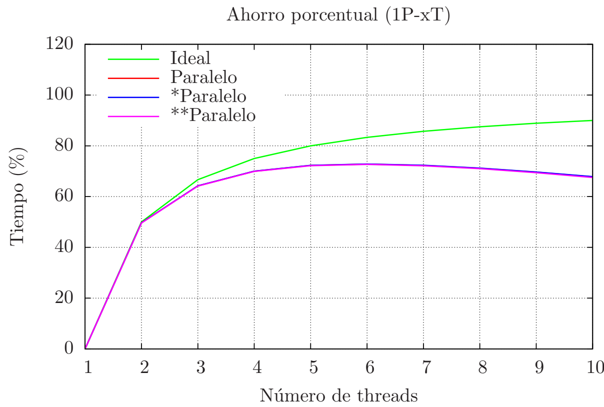
La figura 6.4a muestra que a medida que se aumentan la cantidad de threads, la curva real y la curva ideal se van alejando cada vez más. Esto es lógico considerando que la primera considera un factor que aumenta cuadráticamente y que disminuye el ahorro total entregado por la paralelización. Por otra parte, la figura 6.4b muestra que una vez se fija la cantidad de threads, y con ello el factor cuadrático de costo de overhead, la curva real se comienza a acercar a la curva ideal a medida que se aumentan la cantidad de procesadores.

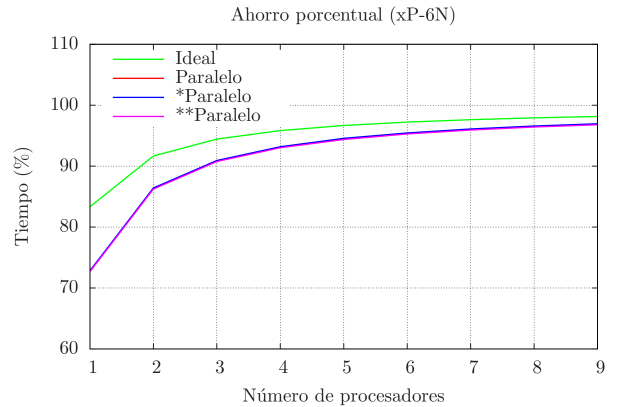
Las figuras 6.4c y 6.4d muestran que a medida que se aumentan la cantidad de threads y procesadores la carga promedio de los threads comienza a disminuir si se considera la curva real y se mantiene relativamente constante para el caso en que se ordenan los escenarios. Este efecto lo explican muy bien las figuras 6.4e y 6.4f puesto que dan cuenta que si se ordenan los escenarios entonces cada thread no varía mucho respecto a los otros.

Sin embargo, al analizar las curvas de ahorro, se puede observar que cualquier esfuerzo que se haga en pos de ordenar los escenarios de manera de mantener una carga balanceada entre los distintos threads no se traduce en ningún beneficio sustancial. La tabla 6.4 muestra los ahorros porcentuales que se obtienen considerando 6 threads por procesador y 9 procesadores, ya que son los óptimos.

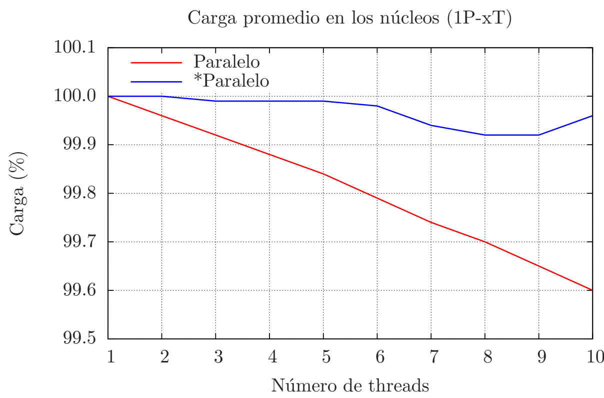
Curva	Ahorro
Ideal	98,15 %
Paralelo	96,89 %
*Paralelo	96,96 %
**Paralelo	96,79 %

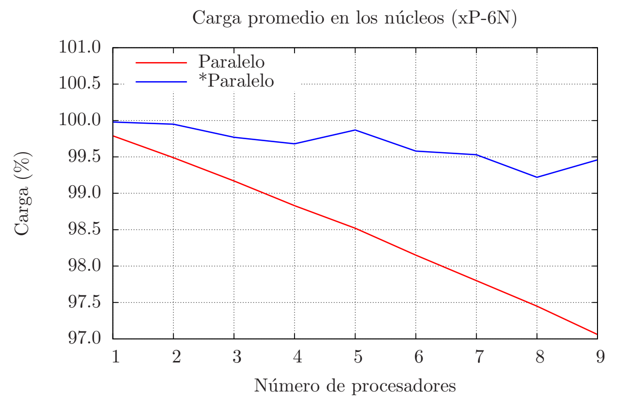
Tabla 6.4: Ahorros obtenidos en la simulación del caso base, para 6 threads por procesador y 9 procesadores.



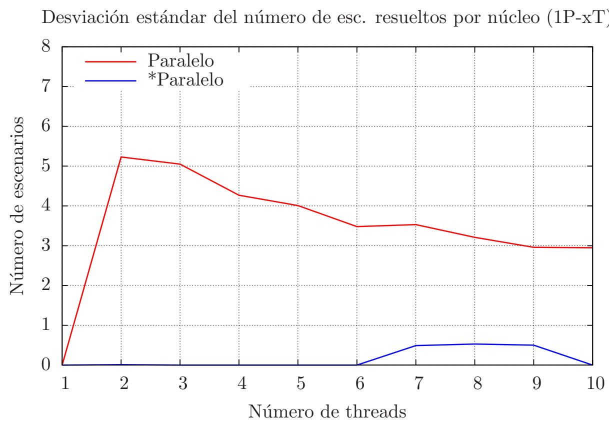


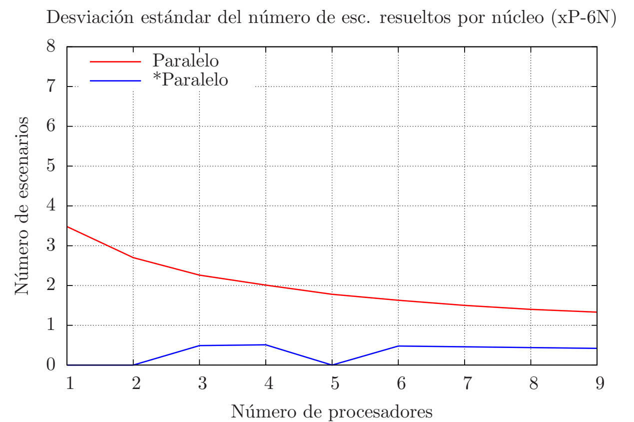
(a) **(b)**





(c) **(d)**





(e) **(f)**

Figura 6.4: Resultados de la simulación del caso base de paralelización considerando el caso ideal (Ideal), el caso en paralelo (Paralelo), el caso paralelo ordenando los escenarios en función de su tiempo de procesamiento (*Paralelo) y el peor caso en paralelo (**Paralelo).

Casos con sensibilidad en la cantidad de escenarios

Si se realiza una simulación considerando una pequeña variación en la cantidad de escenarios respecto al caso base, los resultados no son considerablemente diferentes. Es por ello que se realizó una simulación considerando relativamente pocos escenarios respecto a la cantidad máxima de threads con los que se cuenta para procesar.

Los resultados para el caso 100-escenarios resultaron ser no muy diferentes del caso 200-escenarios por lo que sólo se muestra el ahorro para el primer caso en la figura 6.5. En este caso se puede observar una mayor diferencia entre el peor caso (94,36 %), el caso real (95,73 %) y el caso con los escenarios ordenados (96,77 %). Sin embargo, incluso para un caso extremo como éste, queda en duda si realmente vale la pena invertir esfuerzos en intentar ordenar los escenarios.

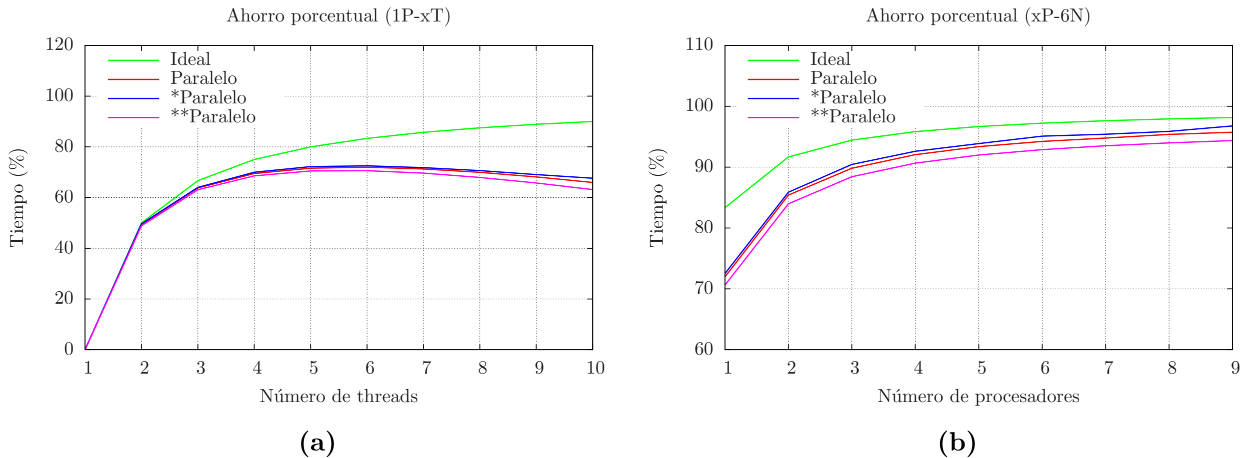


Figura 6.5: Resultados de la simulación del caso 100-escenarios considerando el caso ideal (Ideal), el caso en paralelo (Paralelo), el caso paralelo con los escenarios ordenados (*Paralelo) y el peor caso en paralelo (**Paralelo).

Casos con sensibilidad en el tiempo de resolución

Los resultados entregados por el simulador para los casos media-baja y media-alta fueron prácticamente idénticos a los entregados para el caso base. Este es un resultado esperado pues el valor medio del tiempo de resolución de cada escenario en lo único afecta es en el ahorro absoluto del algoritmo en paralelo y no en su ahorro porcentual.

Si por el contrario lo que se varía es la desviación estándar de los tiempos de resolución, los resultados perciben una pequeña variación. Para el caso desviación-baja lo que ocurre es que la brecha entre el ahorro ideal, el ahorro con los escenarios ordenados y el peor ahorro posible disminuye. Esto es lógico pues, en el caso límite en que todos

los escenarios demoran lo mismo ($\sigma = 0$), la distribución de cargas es equitativa tanto para el caso real como para el caso en que los escenarios se ordenan.

Para el caso desviación-alta se produce el efecto contrario, lo que se traduce en una diferencia un poco mayor entre los ahorros de las distintas curvas. Sin embargo en este caso en particular la diferencia llega a ser casi despreciable (ver tabla 6.5).

Curva	Desviación-baja	Caso base	Desviación-alta
Ideal	98,15 %	98,15 %	98,15 %
Paralelo	96,93 %	96,89 %	96,86 %
*Paralelo	96,96 %	96,96 %	96,97 %
**Paralelo	96,86 %	96,79 %	96,72 %

Tabla 6.5: Comparación del ahorro obtenido para casos con sensibilidad en la desviación estándar, para 6 threads por procesador y 9 procesadores.

Casos con sensibilidad en la curva de overhead

Tal y como se había adelantado, para los casos con sensibilidad en la curva de overhead se variaron los parámetros de la curva (a, b y c) de modo tal que en la "curva-alta" se multiplican los parámetros base por 4, mientras que en la "curva-baja" éstos se dividen por 4 (ver figura 6.6).

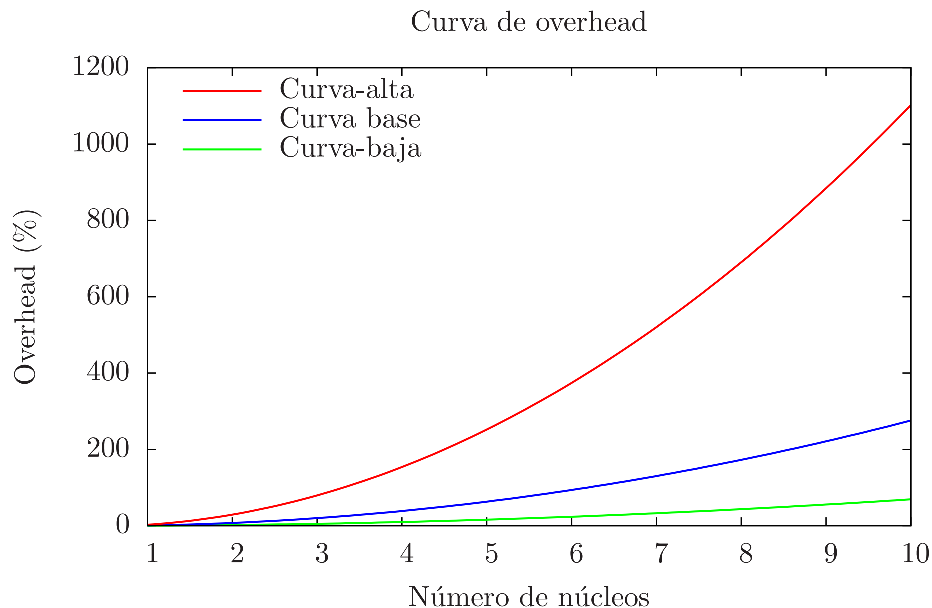


Figura 6.6: Sensibilidad en la curva de overhead.

Al realizar la simulación de escenarios considerando estos cambios en la curva de overhead se obtienen los gráficos que muestra la figura 6.7. En este caso se puede observar que la curva de overhead afecta notablemente el comportamiento de los gráficos.

Para el caso curva-baja se observa que el ahorro porcentual que se puede alcanzar paralelizando el algoritmo se aleja mucho menos del ahorro ideal, en donde el costo de overhead es inexistente. Debido a esto el número óptimo de threads para este caso son 8 (en vez de 6 como en el caso base). Es así como el ahorro porcentual que se puede obtener con 9 procesadores y 8 threads por procesador es de 98,08 %.

Para el caso contrario curva-alta el ahorro porcentual que se puede alcanzar paralelizando se aleja bastante más del ahorro ideal que en el caso base. Su efecto es tan negativo que la cantidad óptima de threads es de sólo 3. Sin embargo, y a pesar de esto, el ahorro porcentual que se obtiene con 9 procesadores y 3 threads por procesador es 95,16 %.

La figura 6.7c muestra cuán relevante puede llegar a ser el efecto de la curva de overhead en el ahorro generado por la paralelización. En este caso, y asumiendo el procesador tuviese más de 10 núcleos, si se paraleliza con más de 10 threads el tiempo total de la iteración inicial PH demoraría más que si se considerase el algoritmo secuencial.

Curva	Curva-baja	Caso base	Curva-alta
Ideal	98,61 %	98,15 %	96,30 %
Paralelo	98,08 %	96,89 %	95,16 %
*Paralelo	98,15 %	96,96 %	95,20 %
**Paralelo	98,00 %	96,79 %	95,07 %

Tabla 6.6: Comparación del ahorro obtenido para casos con sensibilidad en la curva de overhead.

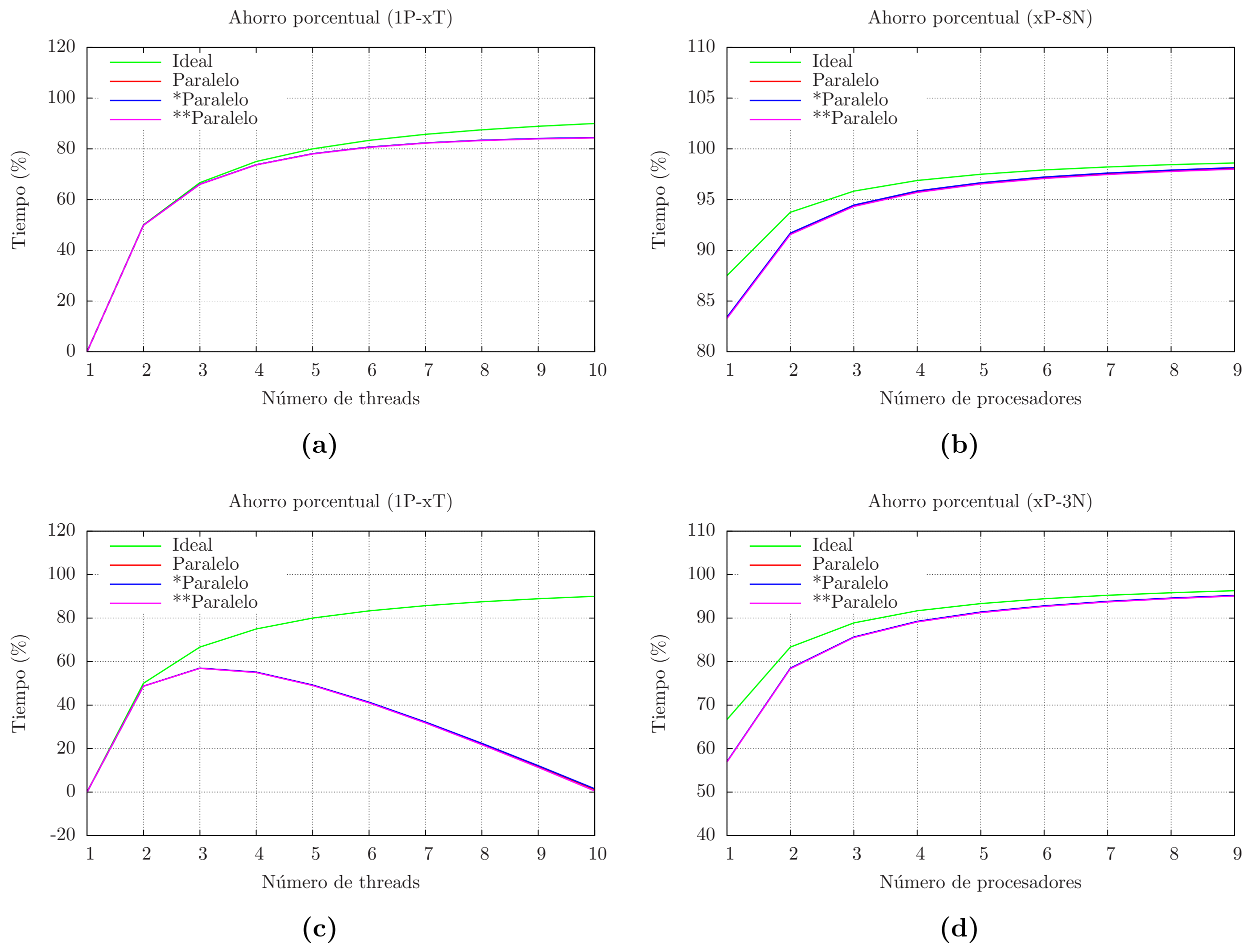


Figura 6.7: Resultados de la simulación de paralelización con sensibilidad en la curva de overhead considerando el caso ideal (Ideal), el caso en paralelo (Paralelo), el caso paralelo ordenando los escenarios en función de su tiempo de procesamiento (*Paralelo) y el peor caso en paralelo (**Paralelo). Las figuras **(a)** y **(b)** corresponden al caso curva-baja mientras que las figuras **(c)** y **(d)** corresponden al caso curva-alta.

Capítulo 7

Conclusiones

La inclusión de estocasticidad en el precio del cobre en el modelamiento del problema de planificación minera de largo plazo hace que la resolución de éste sea virtualmente imposible, incluso considerando poderosos sistemas computacionales, debido al gran tamaño del problema. Es por esto que se considera la utilización de un algoritmo que reduce el problema estocástico en varios problemas determinísticos de modo que el procesamiento del problema global sea más abordable.

Una de las principales bondades que entrega el algoritmo PH (Progressive Hedging) es la independencia que existe en la resolución de los distintos sub-problemas determinísticos (escenarios). De esta forma es posible pensar en su resolución de manera paralela, utilizando sistemas computacionales designados para ello.

A pesar de que cada uno de los escenarios corresponde a un problema relativamente pequeño en comparación con el problema estocástico, es necesario un alto nivel de procesamiento a nivel computacional, lo que en palabras simples significa que cada núcleo genera un alto tráfico de comunicación entre él y su espacio de memoria asignado. Esto se traduce en que existe un alto nivel de overhead asociado a la paralelización en sistemas donde la memoria es compartida entre varios núcleos de procesamiento, alcanzando niveles de un 170 % de tiempo adicional por escenario en las pruebas realizadas cuando se paraleliza con 8 threads. Es por ello que para la resolución del problema estocástico con PH conviene utilizar sistemas de memoria distribuída tales como clúster.

Respecto a los distintos métodos de paralelización del algoritmo PH destaca la resolución asíncrona de cada uno de los escenarios, debido a las diferencias que existen en los tiempos de procesamiento entre un escenario y otro.

Por otra parte, al estudiar la implementacioń del algoritmo PH en el problema minero se pudo notar que las partes no paralelizables del programa son muy pequeñas en comparación con las que sí lo son, lo que se traduce en que la ganancia teórica en

tiempo obtenida debiese ser sobre el 90 % respecto al tiempo que demoraría el mismo problema en ser resuelto de manera secuencial.

Al realizar una simulación de los efectos de la paralelización sobre la iteración inicial del algoritmo PH se pudo apreciar que la curva de beneficios que entrega la cantidad de threads por procesador tiene una forma cóncava. Esto significa que un thread adicional de procesamiento es bueno sólo hasta cierto punto debido a la curva de overhead generada por el acceso al mismo espacio de memoria compartida. Por el contrario, la curva de beneficios que entrega la cantidad de procesadores tiene una forma logarítmica, lo que significa que un procesador adicional siempre es bueno, pero no tanto como el anterior.

Los resultados obtenidos con la simulación indican que el ahorro obtenido al resolver un problema con 9 procesadores y un número óptimo de threads por procesador es del orden el 97 %, valor que puede variar entre un 95 % y un 98 % si lo que se hace es variar la curva de overhead, principal factor dentro de paralelización.

Adicionalmente se observa que no vale la pena ningún esfuerzo por intentar procesar los escenarios de manera ordenada, según su tiempo de resolución, de modo de equilibrar la carga entre todos los núcleos de procesamiento, ya que el beneficio en el ahorro porcentual entregado es menos de un 0,1 %. Del mismo modo si se considera el peor caso paralelizado (peor orden), el ahorro porcentual disminuye en un orden del 0,1 % respecto al ahorro esperado.

Tanto el análisis analítico como los resultados de la simulación apuntan a que la ganancia de la paralelización es indiscutible. La cantidad de series de precios que se esperan considerar en futuras planificaciones son más de 1000, lo que significa que los ahorros no debiesen ser muy diferentes a los obtenidos mediante la simulación, en términos de porcentaje. Más aún, la paralelización entrega la posibilidad de aumentar el tamaño de los problemas determinísticos incluyendo más bancos, periodos o decisiones, entre otras cosas.

Dentro de las principales dificultades encontradas en la formulación de la pequeña implementación, se encuentra la forma en que el thread encargado de controlar la paralelización se comunica con los otros threads para saber cuándo han terminado. Para el caso en que la memoria es compartida el trabajo comunicacional puede ser trivial, pero para un caso real en donde se tienen procesadores con memoria distribuída es necesario contar con un sistema eficiente de modo que no haya un overhead adicional debido a esta comunicación.

Finalmente queda decir que, a pesar de lo simple de la paralelización implementada en este trabajo, el resultado que ésta entrega es muy similar a lo que se esperaría en una paralelización más compleja. Un ahorro en tiempo de sobre el 90 %, llegando incluso a 95 % para el caso estudiado, no deja duda de que la mejor forma de aprovechar el algoritmo Progressive Hedging es paralelizándolo.

7.1. Trabajo futuro

En esta tesis se implementa una paralelización simple sólo de la iteración inicial PH. El siguiente paso sería llevar a cabo la paralelización en las siguientes iteraciones PH, en donde realmente se puede apreciar en completo el beneficio de la paralelización. Esta etapa debiese ser directa pues, la única diferencia sustancial entre la iteración inicial y el resto corresponde a un término adicional dentro de la función objetivo para forzar la convergencia, lo que se traduce sólo en que puede demorar más tiempo cada escenario y no en la forma de paralelización.

En caso de que los niveles de ahorro o ganancia no sean los esperados, existe la posibilidad de utilizar un lenguaje más sofisticado para el trabajo tanto con modelos matemáticos de optimización como con procesamiento multi-threading, tales como python y su extensión a pyomo. Más aún, se propone la utilización de sistemas computacionales con un sistema operativo del tipo Linux, debido principalmente a dos cosas. La idea conceptual de los sistemas Linux es la eficiencia en el trabajo con súper computadores, entregando buenos soportes y utilidades a lo que es la comunicación inter-procesador. De esta forma se puede escalar un problema en un procesador a varios procesadores de manera mucho más sencilla y con menos overhead asociado.

Por otra parte linux posee una mejor programación en la parte del kernel del sistema que tiene que ver con el agendamiento y distribución de los procesos en los distintos núcleos y procesadores, lo que también se traduce en una menor carga de overhead en el sistema en general y, por consiguiente, se pueden obtener mejores ahorros.

Finalmente se propone el análisis algorítmico de la programación del algoritmo PH de modo de reducir toda carga redundante en memoria.

Adicionalmente se podrían probar las metodologías de paralelización del algoritmo expuestas en la sección 5 de modo de sacarle el mayor provecho al hecho de que se resuelven muchos problemas pequeños en vez de uno muy grande.

Capítulo 8

Bibliografía

[1] Jung Ho Ahn, Mattan Erez, and William J. Dally. Tradeoff between data-, instruction-, and thread-level parallelism in stream processors. In *Proceedings of the 21st annual international conference on Supercomputing (21st, 2007, Washington, USA)*, pages 126–137, 2007.

[2] P. Arbenz and A. Adelmann. Parallel numerical computing. http://www.inf.ethz.ch/personal/iyves/pnc11/.

[3] Fernando Badilla. Problema de planificación forestal estocástico resuelto a través del algoritmo progressive hedging. Tesis (magíster en gestión de operaciones), Universidad de Chile, Facultad de Ciencias Físicas y Matemáticas, Santiago, Chile, 2010.

[4] Holger Blaar, Matthias Legeler, and Thomas Rauber. Efficiency of thread-parallel java programs from scientific computing. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (16th, 2002, Florida, USA)*, pages 131–, Abril 2002.

[5] Livermore Computing Center. High performance computing: Training. https://computing.llnl.gov/?set=training&page=index.

[6] Sin Man Cheang, Kwong Sak Leung, and Kin Hong Lee. Genetic parallel programming - design and implementation. *Evolutionary Computation*, 14(2):129 – 156, Junio 2006.

[7] COCHILCO. Informe tendencias del mercado del cobre enero-mayo 2010. Technical report, Comisión Chilena del Cobre, Santiago, Chile, Junio 2010.

[8] COCHILCO. Informe tendencias del mercado del cobre 2011-2012. Technical report, Comisión Chilena del Cobre, Santiago, Chile, Enero 2011.

[9] CODELCO. Memoria anual, 2008.

[10] CODELCO. Memoria anual, 2009.

[11] CODELCO. Memoria anual, 2010.

[12] Marco Colombo and Andreas Grothey. Ergo 09-008. Technical report, School of Mathematics, The University of Edinburgh, Edimburgo, Escocia, 2010.

[13] Sociedad Nacional de Minería. Pib minería. http://www.sonami.cl/index.php?option=com_content&view=article&id=221&Itemid=109.

[14] A. De Silva and D. Abramson. Computational experience with the parallel progressive hedging algorithm for stochastic linear programs. In *Proceedings of 1993 Parallel Computing and Transputers Conference (17th, 1993, Brisbane, Australia)*, pages 164–174, Noviembre 1993.

[15] N. Demirovic, S. Tesnjak, and A. Tokic. Hot start and warm start in lp based interior point method and it's application to multiperiod optimal power flows. In *Proceedings of the 2006 IEEE PES Power Systems Conference and Exposition (2006, Georgia, USA)*, pages 699–704, Octubre 2006.

[16] EDITEC. Codelco potencia su cartera de proyectos. *MCH Mineria Chilena*, 30(349):1–166, Julio 2010.

[17] EDITEC. Codelco a 40 años de la nacionalización del cobre. *MCH Mineria Chilena*, 31(361):1–246, Julio 2011.

[18] Jaime Gacitúa. Aplicación de una heurística escalable para resolver un problema estocático de planificación minera. Tesis (magíster en gestión de operaciones), Universidad de Chile, Facultad de Ciencias Físicas y Matemáticas, Santiago, Chile, 2010.

[19] Marcel GOIC. Formulación e implementación de un modelo de programación matemática para la planificación de largo plazo en minería a cielo abierto. Tesis (magíster en gestión de operaciones), Universidad de Chile, Facultad de Ciencias Físicas y Matemáticas, Santiago, Chile, 2003.

[20] Raphael Gonçalves, Erlon Finardi, and Edson Luiz Da Silva. Exploring the progressive hedging characteristics in the solution of the medium-term operation planning problem. In *Power Systems Computation Conference (17th, 2011, Estocolmo, Suecia)*, Agosto 2011.

[21] Raphael Gonçalves, Erlon Finardi, Edson Luiz Da Silva, and Marcelo dos Santos. Solving the short term operating planning problem of hydrothermal systems by using the progressive hedging method. In *Power Systems Computation Conference (16th, 2008, Glasgow, Escocia)*, Julio 2008.

[22] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. *SIGMETRICS Perform. Eval. Rev.*, 19(1):120–132, Abril 1991.

[23] Diego Hernández. Resultados codelco enero – septiembre 2011. In *Conferencia de prensa*, Santiago, Chile, Noviembre 2011.

[24] K. Hoyland and S. Wallace. Generating scenario trees for multistage decision problems. *Management Science*, 47(2):295–307, Febrero 2001.

[25] IBM. Help - cluster products information center. http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp.

[26] IBM. *Users Manual for CPLEX*. IBM Corporation, 2009.

[27] Elizabeth John and E. Alper Yildirim. Implementation of warm-start strategies in interior-point methods for linear programming in fixed dimension. *Computational Optimization and Applications*, 41(2):151–183, Noviembre 2008.

[28] Manu Konchady. Parallel computing using linux. *Linux J.*, 1998(45es), Enero 1998.

[29] Martin McCarthy. What is multi-threading? *Linux J.*, 1997(34es), Febrero 1997.

[30] Dorit Naishlos, Joseph Nuzman, Chau-Wen Tseng, and Uzi Vishkin. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures (13th, 2001, Crete Island, Grecia)*, pages 93–102, 2001.

[31] Carlos Queiroz, Marco A. S. Netto, and Rajkumar Buyya. Message passing over windows-based desktop grids. In *Proceedings of the 4th international workshop on Middleware for grid computing (4th, 2006, Melbourne, Australia)*, pages 15–, Noviembre 2006.

[32] R. T. Rockafellar and Roger J.-B. Wets. Scenarios and policy aggregation in optimization under uncertainty. *Mathematics of Operations Research*, 16(1):119–147, Febrero 1991.

[33] Richard E. Rosenthal. *GAMS - A Users Guide*. GAMS Development Corporation, Washington, DC, USA, 2010.

[34] Caitlin Sadowski and Andrew Shewmaker. The last mile: parallel programming and usability. In *Proceedings of the FSE/SDP workshop on Future of software engineering research (18th, 2010, New Mexico, USA)*, pages 309–314, Noviembre 2010.

[35] Michael Somervell. Progressive hedging in parallel. Technical report, Department of Engineering Science, University of Auckland, Auckland, New Zeland, 1998.

[36] Oliver Trachsel and Thomas R. Gross. Variant-based competitive parallel execution of sequential programs. In *Proceedings of the 7th ACM international conference on Computing frontiers (7th, 2010, Bertinoro, Italia)*, pages 197–206, 2010.

[37] David L. Woodruff and Jean-Paul Watson. *Progressive Hedging for Multi-stage Stochastic Optimization Problems*.

Anexo A

Modelo matemático de planificación minera de largo plazo

A.1. Conjuntos

t, u: Período del horizonde de planificación.
k: Producto.
a: Expansión.
i, j: Bancos de expansión.
m, n: Nodo de la red de procesos.
e: Equipos de extracción.

A.2. Conjuntos adicionales

$P(i)$: Conjunto de bancos de expansiones que deben ser explotados antes que i por encontrarse en la misma expansión que i, pero en una cota superior a él.

$R(i)$: Conjunto de bancos de expansiones que deben ser explotados antes que i por razones de seguridad.

CON: Productos contaminantes.

NT: Nodos terminales donde pueden comercializarse los productos.

V: Nodos de depósito permanente de producto.

A.3. Parámetros

TON_{ik}: Toneladas del producto k en el banco i.
P_{kt}: Precio del producto k en el periodo t.
$\delta(t)$: Factor de descuento para el periodo t.
$CT_{nkk'}$: Coeficiente de transformación de producto k en producto k' en el nodo n.
$CAPP_{nt}$: Capacidad de proceso en el nodo n en el periodo t.
$CAPS_{nt}$: Capacidad de almacenamiento en el nodo n en el periodo t.
$CAPT_{nmt}$: Capacidad de transporte entre el nodo n y el nodo m en el periodo t.
$CAPE_{egt}$: Capacidad de equipo de tipo e en la mina g en el periodo t.
$CAPV_{nt}$: Capacidad de almacenamiento en el nodo de depósito terminal n en todo el horizonte de planificación.
$DuraE_{eg}$: Vida útil del equipo tipo e de la mina g.
$MaxC_k$: Máxima cantidad a emitir para el contaminante k.
cz_{it}: Costo [$/ton] de explotar el banco i en el periodo t.
$c\alpha_{nt}$: Costo [$] de habilitar nodo de proceso n en el periodo t.
cx_{nt}: Costo [$] fijo del nodo de proceso n en el periodo t.
cp_{nt}: Costo [$/ton] variable de procesamiento del nodo de proceso n en el periodo t.
cf_{mnt}: Costo [$/ton] de enviar producto desde nodo m hasta nodo n en el periodo t.

A.4. Variables de decisión

$z_{it} = 1$: Si el banco de expansión i es explotado en el periodo t.
$y_{nkt} = 1$: Inventario de producto k en el nodo n al final del periodo t.
$f_{mnkt} =$: Cantidad de producto k [ton] enviado desde nodo m hasta n en el periodo t.
α_{nt}: Si se habilita el nodo de proceso n en el periodo t.
α_{mnt}: Si se habilita el arco que conecta a los procesos m y n en el periodo t.
x_{nt}: Si está habilitado el nodo de proceso n en el periodo t.
x_{nmt}: Si está habilitado el arco que conecta a los procesos n y m en el periodo t.
HrR_{egt}: Horas requeridas del equipo tipo e en el periodo t en la mina g.
HrD_{egt}: Horas disponibles del equipo tipo e en el periodo t en la mina g.
HrC_{egt}: Horas compradas del equipo tipo e en el periodo t en la mina g.

A.5. Restricciones

A.5.1. Extracción

Un banco de una expansión se explota sólo una vez en el horizonte de planificación.

$$\sum_t z_{it} = 1 \quad \forall i \tag{A.1}$$

Precedencia entre los abcnos de uan misma expansión, de modo que los bancos superiores se exploten antes que los inferiores.

$$\sum_{u \leq t} z_{iu} \leq \sum_{u \leq t} z_{ju} \quad \forall i, \forall j \in P(i), \forall t \tag{A.2}$$

Precedencia entre los bancos de expansiones ubicadas en la misma pared del rajo, de modo de respetar las condiciones de seguridad.

$$\sum_{u \leq t} z_{iu} \geq \sum_{u \leq t} z_{ju} \quad \forall i, \forall j \in R(i), \forall t \tag{A.3}$$

A.5.2. Disponibilidad de nodos y arcos

Un nodo estará disponible en un período si en alguno de los períodos anteriores se ha habilitado.

$$x_{nt} = \sum_{u \leq t} \alpha_{nu} \quad \forall n, t \tag{A.4}$$

Un arco estará disponible en un período si en alguno de los períodos anteriores se ha habilitado.

$$x_{nmt} = \sum_{u \leq t} \alpha_{nmu} \quad \forall n, m, t \tag{A.5}$$

A.5.3. Conservación de flujos

Conservación de flujo en la extracción de los bancos.

$$TON_{ik} z_{it} = \sum_n f_{inkt} \quad \forall i, k, t \tag{A.6}$$

Conservación de flujo en los nodos de proceso.

$$\sum_{m}\sum kCT_{nkk'}f_{mnkt} = \sum_{m} f_{nmk't} \quad \forall n, k', t \tag{A.7}$$

Conservación de flujo en los stocks.

$$\sum_{m}\sum_{k} CT_{nkk'}f_{mnkt} + y_{nk'(t-1)} = \sum_{m} f_{nmk't} + y_{nk't} \quad \forall n, k', t \tag{A.8}$$

A.5.4. Capacidades

Capacidad de proceso en los nodos de la red.

$$\sum_{m}\sum_{k} f_{mnkt} \leq CAPP_{nt}x_{nt} \quad \forall n, t \tag{A.9}$$

Capacidad de almacenamiento en los nodos de la red.

$$\sum_{k} y_{nkt} \leq CAPS_{nt}x_{nt} \quad \forall n, t \tag{A.10}$$

Capacidad de almacenamiento en los nodos de almacenamiento permanente de la red.

$$\sum_{m}\sum_{k}\sum_{t} f_{mnkt} \leq CAPV_{nt}x_{nt} \quad \forall n \in V \tag{A.11}$$

Capacidad de transporte entre los nodos de proceso.

$$\sum_{k} f_{nmkt} \leq CAPT_{nm}x_{nmt} \quad \forall n, m, t \tag{A.12}$$

A.5.5. Uso de equipos

Horas de equipo tipo e requeridas para soportar la extracción de materiales.

$$HrR_{et} = \sum_{i}\sum_{k} z_{it}TON_{ik}/CAPE_{et} \quad \forall e, t \tag{A.13}$$

Conservación de flujo en las horas del equipo tipo e disponibles en cada período.

$$HrD_{et} = HrD_{e(t-1)} + HrC_{et} - HrC_{e(t-DuraE_e)} \quad \forall e, t \tag{A.14}$$

Las horas requeridas cada período debe ser menor o igual que las horas disponibles.

$$HrR_{et} \leq HrD_{et} \quad \forall e, t \tag{A.15}$$

A.5.6. Contaminantes

La cantidad de contaminantes debe ser menor que lo máximo permitido.

$$\sum_{n \in NT} \sum_{m} \sum_{k} CT_{nkk'} f_{nmkt} \leq MaxC_{k'} \quad \forall k' \in CON, \forall t \tag{A.16}$$

A.6. Función objetivo

A.6.1. Costos de inversión

$$CI = \sum_{t} \delta(t) \left(\sum_{n} c\alpha_{nt} \alpha_{nt} + \sum_{n} \sum_{m} c\alpha_{nmt} \alpha_{nmt} + \sum_{e} \sum_{g} CInv_{eg} HrC_{eg} \right) \tag{A.17}$$

A.6.2. Costos fijos

$$CF = \sum_{t} \delta(t) \left(\sum_{n} cx_{nt} x_{nt} + \sum_{n} \sum_{m} cx_{nmt} x_{nmt} \right) \tag{A.18}$$

A.6.3. Costos variables

$$CV = \sum_{t} \delta(t) \left[\sum_{i} cz_{it} z_{it} + \sum_{n} cp_{nt} \sum_{m} \sum_{k} f_{nmkt} + \sum_{n} \sum_{m} \sum_{k} cf_{nmkt} f_{nmkt} \right] \tag{A.19}$$

A.6.4. Beneficios

$$B = \sum_{t} \delta(t) \sum_{k} \left[(P_{kt} - D_{kt}) \sum_{n \in NT} \sum_{m} f_{nmkt} \right] \tag{A.20}$$

A.6.5. Función objetivo

$$FO = B - CI - CF - CV \tag{A.21}$$

A.7. Naturaleza de las variables

$$CI, CF, CV, B, FO, HrR, HrD, HrC, f, y \geq 0 \quad \text{(A.22)}$$

$$z, \alpha, x \in \{0, 1\} \quad \text{(A.23)}$$

Anexo B

Resultados de la simulación para el caso 200-escenarios

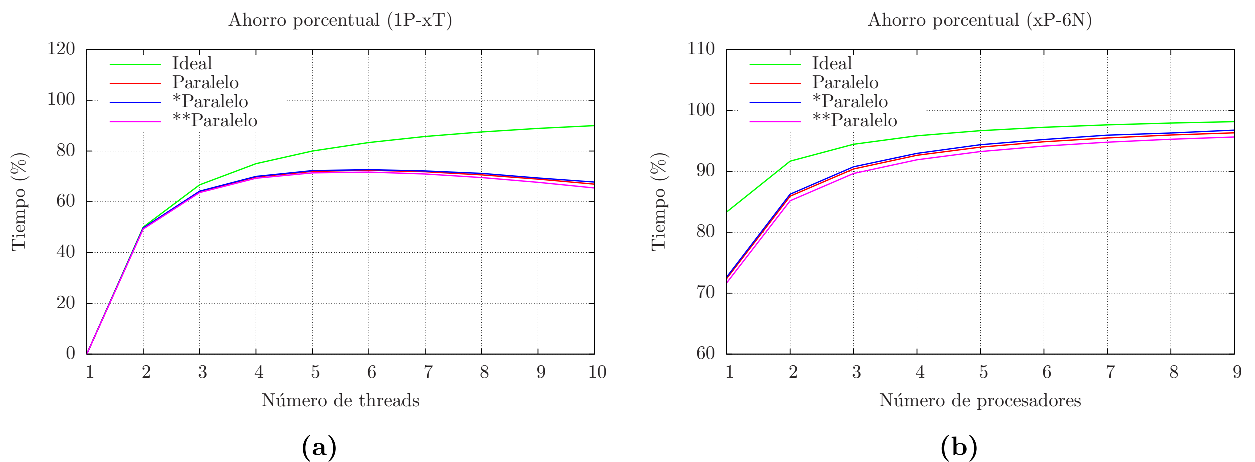


Figura B.1: Resultados de la simulación del caso 200-escenarios considerando el caso ideal (Ideal), el caso en paralelo (Paralelo), el caso paralelo con los escenarios ordenados (*Paralelo) y el peor caso en paralelo (**Paralelo).