UNIVERSIDAD NACIONAL DEL SUR

TESIS DE MAESTRÍA EN INGENIERÍA
DE PROCESOS PETROQUÍMICOS

Optimización Estocástica Acelerada con Aplicación a la Ingeniería de Procesos

Lucía Damiani

BAHÍA BLANCA ARGENTINA

2019

PREFACIO

Esta tesis se presenta como parte los requisitos para optar al grado académico de Magister en Ingeniería de Procesos Petroquímicos de la Universidad Nacional del Sur y no ha sido presentada previamente para la obtención de otro título similar en esta institución u otras. La misma contiene los resultados obtenidos de la investigación llevada a cabo en ámbito del Departamento de Ingeniería Química de la Universidad Nacional del Sur durante el período comprendido entre el 15 de Marzo de 2015 y el 18 de Junio de 2019, bajo la dirección de los Drs. Aníbal M. Blanco y Javier Iparraguirre.

Lucía Damiani

UNIVERSIDAD NACIONAL DEL SUR

Secretaría General de Posgrado y Educación Continua

La presente tesis ha sido aprobada el …. /..... /...., mereciendo la calificación de ….. (……)

AGRADECIMIENTOS

A mis directores, los Dres. Aníbal Blanco y Javier Iparraguirre, por la formación, guía, apoyo y recomendaciones que me brindaron para poder completar la tesis.

A los docentes del Departamento de Ingeniería Química de la Universidad Nacional del Sur por los conocimientos transferidos en el cursado de las materias del programa.

Al grupo de investigación del Dr. Javier Iparraguirre en la UTN por la buena predisposición y la colaboración.

A mis compañeros de PLAPIQUI, en particular el grupo de procesos, por las horas de trabajo y los momentos compartidos. En especial a mis compañeros de oficina Fiorella Cravero, Yamila Grassi y Franco Poggio.

A mi familia, por su acompañamiento y apoyo incondicional.

RESUMEN

Los problemas de optimización no lineal que poseen una gran cantidad de variables, ecuaciones y no linealidades, suelen presentar un considerable desafío matemático. Si bien existen numerosas plataformas de software para su formulación y resolución, muchas poseen costosas licencias propietarias. Además, aun contando con las herramientas más sofisticadas suele necesitarse un considerable esfuerzo de programación (reformulaciones, descomposiciones, etc.) para implementar y resolver este tipo de modelos.

En esta tesis se propone confeccionar una herramienta propia de optimización no lineal basada en metaheurísticas, empleando recursos de software libre, que permitan al grupo realizar proyectos de investigación y transferencia sin depender de los costos asociados a las licencias de las herramientas comerciales.

En los últimos años, las metaheurísticas basadas en poblaciones han tomado gran relevancia debido a su eficiencia, facilidad de programación, habilidad para resolver una amplia variedad de problemas y posibilidad de combinarse con otros algoritmos para mejorar sus prestaciones. En este trabajo, se implementó una de estas técnicas, la optimización por enjambre de partículas (PSO), para programar y resolver problemas de optimización no lineal. Dado que la optimización con PSO suele resultar computacionalmente costosa, se paralelizó el algoritmo sobre placas gráficas (GPU) de manera de explotar el paralelismo implícito de la técnica y aprovechar el amplio acceso a estos dispositivos de bajo costo disponibles en las computadoras de escritorio modernas.

El PSO implementado, en sus versiones serie y paralelo, se testeó con funciones benchmark de diferente dificultad, con y sin restricciones, ampliamente utilizadas en la literatura. También, se lo aplicó a modelos más complejos y de mayor escala del área de la ingeniería química. En todos los casos se observaron desempeños aceptables, tanto respecto de la calidad de las soluciones halladas como de las aceleraciones obtenidas.

ABSTRACT

Nonlinear optimization problems, with medium/large number of variables, equations and nonlinearities, usually present a significant mathematical challenge. Despite there are many technologies for their formulation and resolution, the most competitive ones, have expensive proprietary licenses. Moreover, even counting with these commercial tools, usually a considerable additional programming effort is required (reformulations, decompositions, etc.) to implement and solve this type of models.

This thesis proposes the development of a non-linear optimization tool based on metaheuristics using free software resources, to allow our group making research and transference projects without depending on the costs associated with commercial licenses.

In recent years, population based metaheuristics acquired relevance because of their efficiency, ease of programming, ability to solve a wide range of problems and possibility to combine with others algorithms to improve performance. In this work, one of these techniques, the particle swarm optimization algorithm (PSO) is implemented to program and solve non-linear optimization problems. Since optimization with PSO is often computationally expensive, the algorithm was parallelized on Graphic Processing Units (GPU) in order to exploit the implicit parallelism of this technique and take advantage of the wide access to these low-cost devices available in modern desktop computers.

The implemented PSO, in its serial and parallel versions, was tested with benchmark functions of different difficulty, with and without constraints, widely used in the optimization literature. It was also applied to more complex and larger-scale models of the chemical engineering discipline. In all cases, the optimizer provided acceptable performance regarding solution quality and speedups.

TABLA DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1

INTRODUCCIÓN

Muchos problemas de interés para las ciencias e ingenierías pueden formularse como problemas de optimización. Los más típicos son el diseño, planeamiento y operación óptima de sistemas y procesos, y el ajuste de parámetros, tanto en versiones de estado estacionario como dinámicas. Cuando se optimiza, se desea encontrar la combinación de variables que satisfagan un determinado criterio de desempeño del sistema (función objetivo) respetando una serie de restricciones de igualdad y desigualdad. Una formulación bastante general de un problema de optimización se puede expresar de la siguiente manera (Ec. 1.1):

$$\begin{aligned} & Min_{\boldsymbol{x}}\, f(\boldsymbol{x}) \\ st:\ & h(\boldsymbol{x}) = 0 \\ & g(\boldsymbol{x}) \leq 0 \\ & \boldsymbol{x^{lo}} - \boldsymbol{x} \leq 0 \\ & \boldsymbol{x} - \boldsymbol{x^{up}} \leq 0 \end{aligned} \qquad (1.1)$$

Donde $f(.)$ es la función objetivo, $h(.)$ y $g(.)$ representan las restricciones de igualdad y desigualdad respectivamente y el vector $\boldsymbol{x}$ es el conjunto de variables de optimización. Los parámetros $\boldsymbol{x^{lo}}$ y $\boldsymbol{x^{up}}$ son los límites inferiores y superiores respectivamente de cada variable.

Una versión particularmente desafiante del problema (1.1) surge cuando posee la complejidad relacionada con la no-linealidad de las expresiones involucradas, tanto en la función objetivo como en las restricciones, la cual produce no-convexidades que se manifiestan habitualmente a través de la presencia de numerosos óptimos locales. Encontrar la mejor solución de todas, o incluso hasta una solución factible es un problema de considerable dificultad. Estas características de los problemas de optimización no-

lineal los hacen de los más desafiantes dentro de la matemática aplicada moderna y, desde el punto de vista científico, el reto permanente es intentar resolver modelos de cada vez mayor escala, tanto en número de variables como de ecuaciones, en tiempos de cómputo compatibles con aplicaciones prácticas.

Actualmente existen algoritmos muy desarrollados para realizar modelado matemático y optimización. Entre las herramientas más populares se encuentran, por ejemplo, gProms[1], GAMS[2] y MATLAB[3]. Además, cabe destacar que la gran mayoría del software más competitivo es propietario y posee costos elevados en conceptos de licencias. Por ejemplo, para el caso de GAMS, en la Tabla 1.1 se detallan los valores de las licencias académicas[4] y comerciales[5] para un paquete básico.

Tabla 1.1: Precios licencias (USD) de GAMS en Junio 2019

	Tipo de modelo	Licencia Académica	Licencia Comercial
Module Base	-	640	3200
CPLEX	LP/MIP/QCP/MIQCP	1280	9600
CONOPT	NLP	640	3200
DICOPT	MINLP	320	1600
TOTAL (1 usuario)		2880	17600
TOTAL (5 usuarios)		5760	35200
TOTAL (10 usuarios)		8640	52800

Como puede observarse, los valores son significativos para los montos disponibles en los subsidios con los que cuentan los grupos de investigación para adquirir licencias académicas. Por su parte, los costos de las licencias comerciales suelen ser prohibitivos

[1] www.psenterprise.com/products/gproms
[2] www.gams.com
[3] www.mathworks.com/products/matlab.html
[4] www.gams.com/fileadmin/news-events/Pricelists/academic_price_list.pdf
[5] www.gams.com/fileadmin/news-events/Pricelists/standard_price_list.pdf

para organizaciones sin fines de lucro y pequeñas y medianas empresas. Incluso las empresas industriales de gran envergadura deben estar, en general, muy convencidas del beneficio del proyecto para realizar inversiones de estas características, a pesar de que, para sus volúmenes de negocios, estas sumas puedan parecer insignificantes.

Un caso paradigmático es el del sector de la salud pública en Argentina, donde abundan los sistemas que podrían ser gestionados de manera mucho más eficiente si se contara con herramientas de asistencia a la toma de decisiones basadas en modelos matemáticos de optimización. Entre estos se destaca el problema de planificación del uso de quirófanos en los hospitales debido a su alto costo operativo, el empleo intensivo de recurso humano que poseen y el impacto de su funcionamiento sobre la calidad de vida de la población. Sin embargo, el sector público no suele contar con los medios para adquirir este tipo de licencias de desarrollo, aunque se esté convencido, en muchos casos, de las ventajas de adoptar estas tecnologías.

Además del aspecto del costo, también es necesario tener en cuenta que, dependiendo del tamaño y tipo de problema, el uso de algoritmos de optimización numéricos (solvers) comerciales tampoco garantiza resultados satisfactorios en todas las circunstancias. El teorema de “no free lunch” en optimización establece que los optimizadores universales son imposibles, es decir, que no existe una estrategia que supere a todas las demás en todos los problemas (Ho y Pepyne, 2002). Entonces, a pesar de los numerosos esfuerzos para desarrollar solucionadores de optimización generales, sólo se espera que éstos se desempeñen eficientemente en un subconjunto del universo de problemas de optimización.

Por ejemplo, Chen y col. (2017) y Lang y Zhao (2016), plantearon modelos mixto-entero lineales de gran escala para distintas aplicaciones de interés industrial. Al intentar resolverlos con el solver CPLEX, ambos concluyeron que a medida que el modelo crecía, no se llegaba a una solución óptima o factible ya que se agotaba la memoria del sistema o el tiempo de CPU asignado. En ambos casos, los autores proponen desarrollos propios para resolver las instancias más desafiantes de sus respectivos modelos. Por su parte,

Gunnerud y col. (2012) presenta un modelo de planificación de la producción de campos petroleros. Debido a que no pudo resolver instancias realistas de este problema con el solver comercial XPRESS, los autores debieron programar estrategias de descomposición ad-hoc en la plataforma de programación adoptada (Mosel).

Asimismo, Pfetsch y col. (2015) plantearon un modelo MINLP para optimizar una red de gas en GAMS con los solvers BARON y SCIP. Debido a que se trata de un problema no convexo de gran complejidad, los autores debieron dividirlo en dos fases para poder resolverlo. En la primera, tuvieron que combinar cuatro enfoques distintos para encontrar los valores de las variables discretas, aunque en varios casos llegaron a soluciones infactibles. Luego, las soluciones factibles encontradas se utilizaron en la segunda fase para calcular las soluciones del problema NLP resultante.

También Tabkhi y col. (2010) optimizaron una red de gas natural con múltiples fuentes de suministro y múltiples puntos de entrega del producto. La inicialización del conjunto de variables binarias las proporciona el usuario, mientras que las variables continuas se inicializan automáticamente por el solucionador, ya que se encuentran bien delimitadas. Los autores debieron reformular el modelo implementando técnicas de convexificación lineal en combinación con una técnica de "Branch and Bound" para convertir el problema MINLP en subproblemas NLP más fáciles de resolver con los solvers CONOPT y SBB de GAMS.

Los ejemplos anteriores sugieren que, para muchas aplicaciones de cierta dimensión, aun contando con las plataformas de desarrollo más sofisticadas y amigables, se suele requerir de un esfuerzo de programación adicional significativo para implementar y resolver los modelos resultantes.

Una opción para disponer de instrumentos accesibles para desarrollar proyectos de investigación y transferencia para el medio social y productivo nacional, evitando los costos asociados a la adquisición de licencias propietarias, es la confección de

herramientas propias de optimización empleando recursos de software libre. En esta tesis se propone el desarrollo de una herramienta de estas características.

Cabe aclarar que no se pretende desarrollar una tecnología que compita con plataformas comerciales de modelamiento y optimización de propósito general como las mencionadas anteriormente. Más bien se busca proporcionar soluciones de aceptable eficiencia, accesibles para organizaciones que no disponen de grandes presupuestos para adquirir software de soporte a la toma de decisiones y desarrollar proyectos de optimización.

Asimismo, es necesario mencionar que pueden encontrarse muchas herramientas de optimización libres desarrolladas por profesionales independientes, grupos de investigación e incluso empresas de diferentes partes del mundo, las cuales están disponibles para ser descargadas, utilizadas e incluso modificadas. Muchas de ellas son muy competitivas y bastante utilizadas y referenciadas. Sin embargo, la mayoría de estas herramientas no poseen un soporte permanente y siempre está el riesgo de que el proyecto se discontinúe. Otra desventaja es que suele ser difícil "apropiarse" de código desarrollado por terceros, aunque esté elaborado con las mejores prácticas. Finalmente, desde el punto de vista de la posible publicación de resultados en foros científicos/técnicos, contar con herramientas de desarrollo propio que puedan ir creciendo y mejorando en el tiempo, nos parece una estrategia más atractiva que utilizar software de terceros en el largo plazo.

Existen varias formas de realizar optimización numérica con la computadora. En la última década han cobrado gran importancia las estrategias estocásticas basadas en evolución de poblaciones (Rangaiah, 2010). Entre ellas se destacan los "algoritmos genéticos" y la optimización con "enjambres de partículas" (Kennedy y Eberhardt, 1995).

Conceptualmente, estas técnicas parten de un conjunto inicial de posibles soluciones en el espacio de búsqueda (población inicial), el cual va evolucionando por medio de reglas más o menos sencillas, usualmente inspiradas en procesos naturales, hasta converger a una población final que se espera contenga la solución buscada. Las principales ventajas

de este tipo de técnicas es que son relativamente simples de programar y poseen mucha flexibilidad para ser combinadas con otros algoritmos, extender sus prestaciones y adaptarlos a un gran espectro de problemas. Poseen, además, numerosas aplicaciones de éxito reportadas en prácticamente todas las áreas de la ingeniería. Por estas razones se ha preferido, en este trabajo, adoptar este tipo de metodologías para los desarrollos.

Sin embargo, una de las debilidades más importantes de estas técnicas es la carencia de un tratamiento elegante y exacto de las restricciones. Además, usualmente se requiere de un gran número de evaluaciones de las funciones involucradas para encontrar soluciones de aceptable calidad, lo que resulta habitualmente en prolongados tiempos de ejecución. Por estos motivos, en la actualidad, la investigación en optimización estocástica aborda principalmente, por un lado, el tratamiento eficiente de las restricciones y, por otro, las técnicas de aceleración del algoritmo.

Respecto del tratamiento de restricciones, las estrategias más populares son las de penalidad y las de factibilidad (Zhang y Rangaiah, 2012), cada una con ventajas y desventajas respecto de la otras y con mejores prestaciones para cierto tipo de modelos que para otros.

En lo que concierne a la aceleración, durante los últimos años el cómputo de propósito general usando Unidades de Procesamiento Gráfico (GPU por sus siglas en inglés) ha cobrado una gran relevancia (Nickolls y Dally, 2010). Las GPU son procesadores masivamente paralelos, de relativo bajo costo, que se han creado originalmente para renderizado de juegos de computadora 3D y que, actualmente, han encontrado numerosas aplicaciones en computación científica. El rápido crecimiento de este tipo de dispositivos ha permitido gran poder de cómputo en estaciones de trabajo de uso general y han contribuido a un salto de desempeño de al menos un orden de magnitud en el poder de cálculo (Papadrakakis y col., 2011). Las arquitecturas de GPUs contemporáneas disponen de cientos de procesadores con capacidad de ejecutar tareas en forma paralela permitiendo acelerar una gran cantidad de aplicaciones. Los problemas de optimización

basados en poblaciones se pueden categorizar como “masivamente paralelos” y son aptos para ser ejecutados en aceleradores modernos (Mussi y col., 2011).

Este proyecto propone básicamente implementar un optimizador estocástico, basado en enjambre de partículas, que resuelva de manera eficiente problemas complejos de la ingeniería caracterizados por ser no lineales y poseer restricciones. Se pretende desarrollar una herramienta propia, empleando software libre, que permita realizar proyectos y transferir la herramienta sin necesidad de contar con licencias de software de desarrollo comercial.

Específicamente, se decidió trabajar con el sistema operativo Linux e implementar los algoritmos en lenguaje de programación Python. Asimismo, con el fin de realizar un desarrollo colaborativo más eficiente y práctico, se utilizará la plataforma Bitbucket. A continuación, se proporciona un breve detalle de estos y otros recursos a utilizar.

1.1 GNU -Linux 

Es un sistema operativo muy robusto, estable, con gran poder de cómputo y procesamiento. Es muy dúctil, ya que puede funcionar en múltiples plataformas y dispositivos, facilitando también el uso de todo tipo de hardware. Posee una importante capacidad multitarea y multiusuario que permite aprovechar completamente las funcionalidades de las computadoras personales que cuentan con microprocesadores i386 y versiones superiores. Linux es un sistema Unix que soporta software de libre distribución junto con su código fuente mientras que garantiza un avanzado nivel de seguridad.

Se eligió trabajar con Linux principalmente por dos ventajas que se destacan: es de acceso libre y gratuito y permite una comunicación más natural entre servidores. La primera admite que cualquier persona pueda acceder a este sistema operativo sin necesidad de realizar erogaciones en conceptos de licencia. La segunda, brinda la posibilidad de utilizar

desde el lugar de trabajo, recursos que se encuentran físicamente en otro establecimiento. Particularmente, se utilizó la distribución Linux Mint 18.1.

Una de sus desventajas es que, para poder dominarlo, se requiere de un mayor conocimiento de las herramientas del sistema (en contraste con otros sistemas operativos como Windows). Sin embargo, durante la última década, se han creado interfaces que resultan más sencillas e intuitivas para el usuario, permitiéndoles realizar diferentes gestiones más fácilmente. Asimismo, cada usuario tiene la posibilidad de personalizar su interfaz, adaptando el entorno de trabajo a sus propias necesidades y requerimientos.

1.2 Python 

Existe una gran cantidad de lenguajes de programación libres entre los que se puede optar en la actualidad. Python[6] se destaca por su sencillez, legibilidad y precisión de sintaxis. Es un lenguaje de programación de alto nivel que permite la rápida implementación de aplicaciones con muy poco código. Es ideal para producir prototipos rápidamente (prototipado rápido), pero también permite el desarrollo de algoritmos complejos. El código es muy sencillo de comprender y al usar la indentación en lugar de los paréntesis permite una mayor legibilidad del mismo, logrando una gran calidad de sintaxis.

Python posee una curva de aprendizaje sencilla y amigable, su sintaxis de propósito general es muy intuitiva: cualquier desarrollador pueda dedicar poco tiempo para aprenderlo y concentrarse más en cómo crear productos innovadores con él, aprovechando sobre todo su gran flexibilidad. Cuenta con una comunidad muy activa, capaz de aportar tutoriales y respuestas a problemas concretos que facilitan la programación. Se trata de una herramienta que ahorra mucho tiempo de programación debido a su simplicidad, versatilidad, agilidad de lectura y amplia disponibilidad de librerías de todo tipo, lo que lo ha hecho crecer popularmente en las aplicaciones científicas.

[6] www.python.org

Es un lenguaje interpretado, por lo que es multiplataforma, funciona en cualquier tipo de sistema que integre su interpretador. Al ser orientado a objetos y dinámico, se le puede agregar nuevas funciones y clases a un objeto existente, incluso en tiempo de ejecución. Otra gran ventaja es que se lo puede utilizar como software "glue" para combinar distintas aplicaciones realizadas en otras plataformas.

Python cuenta con tres librerías científicas[7] muy desarrolladas y de gran capacidad, cuya descarga es gratuita, que potencian su empleo en aplicaciones de ciencia e ingeniería:

- *Numpy*: añade a Python funcionalidades para el manejo sencillo y eficiente de operaciones matriciales y vectoriales, desde las más básicas hasta las más complejas. Además, proporciona herramientas que permiten incorporar código fuente de otros lenguajes de programación como C/C++ o Fortran, lo que incrementa notablemente su compatibilidad e implementación.
- *Scipy*: es una librería de herramientas numéricas que usa Numpy como base. Posee módulos para optimización de funciones, integración, funciones especiales, y resolución de ecuaciones diferenciales ordinarias, entre otros.
- *Matplotlib*: es una biblioteca para la generación de gráficos 2D y 3D a partir de datos contenidos en listas o arreglos.

El "ecosistema" Python (Fig. 1.1) ha sido concebido como una alternativa libre y abierta al software propietario Matlab. Cabe aclarar que, además de las librerías mencionadas, existen una gran cantidad de paquetes y "toolboxes" de acceso libre para uso científico desarrollados en esta plataforma por programadores del ámbito académico y privado.

[7] www.scipy.org

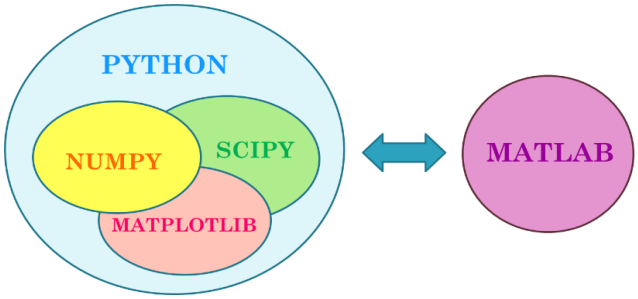

Fig. 1.1: Paquetes de Python vs MATLAB.

Este lenguaje cuenta con distintos entornos de trabajo (frameworks) de gran utilidad que permiten auxiliar desde el desarrollo de juegos hasta el de algoritmos científicos de cálculos avanzados. Por ejemplo, a través del framework Django[8], que permite el desarrollo de aplicaciones web, se cuenta con la posibilidad de poner "en línea" y con poco esfuerzo los desarrollos, potenciado así la capacidad de transferencia de los mismos.

Python se puede ejecutar en diversos entornos de desarrollo integrado (IDEs por sus siglas en inglés), desde algunos muy sencillos, como IDLE, hasta otros más completos como Spyder, Eclipse o NetBeans. En particular, en esta tesis se utiliza PyCharm[9], dado que cuenta con una comunicación más fluida con el control de versiones elegido que se explica a continuación.

Probablemente, el mayor defecto de Python es que al ser un lenguaje interpretado, no posee una gran rapidez de ejecución frente a lenguajes compilados como C y Fortran. Con el objeto de suplir esta desventaja y conservar la alta productividad de desarrollo y transferencia que brinda Python, se potenciará su uso mediante el empleo de GPUs.

1.3 Bitbucket

Bitbucket[10] es un Sistema de Control de Versiones (SCV) distribuido basado en web, para los proyectos que utilizan el sistema de control de versiones Mercurial y Git. Un SCV es un

[8] www.djangoproject.com
[9] www.jetbrains.com/pycharm
[10] www.bitbucket.org

sistema informático encargado de registrar los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que se pueden gestionar las diferentes versiones producidas: recuperar, mezclar, comparar, etc. La utilización de SCV facilita el trabajo en modo colaborativo y paralelo, mediante el uso de ramas y de funcionalidades como la capacidad de combinar o mezclar (merge) versiones diferentes de un mismo documento. En general, cualquier SCV ofrece capacidades para revertir un archivo o un proyecto entero a un estado anterior o de recuperarlos si se perdieran. Permite también comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que puede estar causando un problema, y cuándo se introdujo un error, entre muchas otras funcionalidades.

Debido a lo mencionado anteriormente, Bitbucket admite la revisión de código de manera eficiente, ya que todos los integrantes de un grupo de trabajo pueden analizarlo. Además, es posible realizar discusiones en el código fuente con comentarios en línea. Con esta herramienta es más sencilla la comunicación entre los programadores, facilitando la colaboración y acelerando el proceso de desarrollo.

Bitbucket ofrece opciones comerciales y gratuitas. Éstas últimas cuentan con un número limitado de repositorios privados que puede tener hasta cinco usuarios, lo cual es suficiente para los fines de este proyecto.

Principalmente se decidió utilizar este sistema de control de versiones ya que se basa en Git y debido a que facilitó la interacción con el grupo de cómputo de alto desempeño (UTN-FRBB). Se contó con su apoyo para el desarrollo de una parte de esta tesis al ser su principal línea de investigación la aceleración de algoritmos y el paralelismo empleando clusters y GPU.

1.4 CUDA/PyCUDA 

CUDA (Compute Unified Device Architecture) engloba a un conjunto de herramientas que permite el desarrollo de programas sobre arquitecturas de computación paralela. En

particular, admite el desarrollo sobre unidades de procesamiento gráfico (GPUs) del proveedor NVIDIA. Específicamente, se utilizó PyCUDA ya que permite que el código Python se ejecute en la GPU a través del entorno CUDA, el cual usa una variación del lenguaje de programación C. De esta manera, se aprovechan tanto las ventajas de Python (descritas anteriormente) como la capacidad de acelerar los tiempos de cómputo que brinda la GPU a través de CUDA. En el Capítulo 3 se detallará más sobre ambos programas y cómo funcionan.

1.5 Objetivos y Estructura de la Tesis

El objetivo principal de esta tesis es desarrollar una herramienta para resolver problemas de optimización no-lineal basada en algoritmos estocásticos empleando software libre. Para potenciar el desempeño de la herramienta se hará uso de la capacidad de cómputo paralelo disponible en computadoras personales modernas.

El optimizador, basado en la técnica estocástica de Enjambre de Partículas (Particle Swarm Optimization, PSO) permite, en muchos casos, encontrar soluciones de aceptable calidad a diferentes problemas de optimización no-lineal de interés para las ingenierías, y en particular para la Ingeniería de Procesos Petroquímicos. Dicha implementación se realizará inicialmente en serie, y luego se la paralelizará sobre placas gráficas con el fin de reducir los tiempos de cómputo involucrados en las optimizaciones.

Se pretende crear una herramienta propia utilizando los recursos de software libre descritos anteriormente, para aplicarla tanto en proyectos de investigación como de transferencia al medio social y productivo nacional, sin las restricciones y costos de licencia que impone el uso de plataformas comerciales.

A continuación, se describe brevemente la estructura de esta tesis:

En el Capítulo 2, se describe el funcionamiento y metodología de la metaheurística adoptada, la optimización por enjambre de partículas (PSO). Asimismo, se explica la

técnica de manejo de restricciones que se eligió para extender la aplicación del PSO a modelos con restricciones de igualdad y desigualdad. Por último, se detalla la implementación propia del algoritmo.

En el Capítulo 3, se realiza una introducción general a la paralelización en GPU. Se describe el lenguaje a utilizar para trabajar con la misma, PyCUDA, y se detalla el modelo de programación adoptado. Luego, se hace una revisión bibliográfica de las distintas metaheurísticas aceleradas en GPU, con énfasis en las que utilizan PSO. Finalmente, se describe la implementación propia del algoritmo en paralelo.

En el Capítulo 4, se efectúa el testeo de los algoritmos, serie y paralelo, aplicándolos a problemas benchmark cuya solución es conocida. A su vez, se describe un problema de interés industrial de reconocida dificultad, la optimización de mezclas de crudo, y se resuelven tres versiones del mismo con las implementaciones serie y paralelo del PSO propio.

En el Capítulo 5, se aplica la herramienta desarrollada a problemas de ingeniería tomados de literatura. Estos son de mayor tamaño y complejidad que los reportados en el capítulo anterior, ya que el objetivo es probar las prestaciones de los algoritmos en aplicaciones más desafiantes.

Por último, en el Capítulo 6 se presentan las conclusiones de la tesis y se plantea el trabajo a futuro según la experiencia obtenida durante el desarrollo de la misma.

CAPÍTULO 2

OPTIMIZACIÓN POR ENJAMBRE DE PARTÍCULAS

En este capítulo se presenta una descripción de la Optimización por Enjambre de Partículas (PSO por sus siglas en inglés). Se explicará el funcionamiento de este método de optimización estocástico, se describirá el significado de los distintos parámetros que posee y se detallará el enfoque elegido para el tratamiento de restricciones. Por último, se presentará la implementación propia del algoritmo.

2.1 Introducción

Una clasificación amplia de los algoritmos de optimización numérica involucra enfoques determinísticos y metaheurísticos.

La optimización determinística utiliza información de las propiedades matemáticas de la función objetivo y de las restricciones (continuidad, diferenciabilidad y convexidad) para tratar con las no-linealidades presentes (Tawarmalani y Sahinidis, 2002). Esta clase de algoritmos de optimización son matemáticamente sofisticados y las implementaciones computacionales más competitivas suelen ser productos propietarios comerciales. En modelos no lineales, los algoritmos determinísticos dependen fuertemente de un buen valor inicial de las variables para converger a una solución, la cual es habitualmente un óptimo local. Aunque la optimalidad global se puede garantizar bajo ciertas condiciones, en modelos complejos, las ejecuciones muchas veces provocan que la memoria se agote o que el proceso alcance los límites de tiempo asignados antes de lograr una convergencia, incluso para instancias de tamaño medio, en términos de variables y ecuaciones. Esta situación puede resolverse en algunos casos mediante la implementación de técnicas de descomposición de los modelos, como ser las descomposiciones de Benders y Langrangeana (Conejo y col., 2006). La plataforma de modelado GAMS posee mayormente algoritmos de tipo de determinísticos en su librería de solvers.

Por su parte, las metaheurísticas están diseñadas para resolver de manera aproximada una amplia gama de difíciles problemas de optimización sin adaptaciones específicas para cada uno. Estas técnicas funcionan mejorando un punto inicial (o población inicial de puntos) en el espacio de búsqueda de acuerdo con un conjunto de reglas normalmente sencillas generalmente inspiradas en procesos naturales, típicamente físicos y biológicos, que poseen elementos estocásticos para guiar la exploración del espacio de soluciones (Boussaid y col., 2013). Este proceso de evolución suele proporcionar soluciones cercanas en muchos casos al óptimo global sin la necesidad de explicitar un buen punto inicial, aunque es necesario establecer un equilibrio adecuado entre la exploración de la región factible y la explotación de los puntos más prometedores para lograr el desempeño eficiente de una metaheurística.

Una de sus principales ventajas es que solamente utilizan valores de función objetivo y restricciones en el procedimiento de búsqueda, sin recurrir al cálculo de derivadas, a diferencia de los métodos determinísticos, y pueden, por lo tanto, utilizarse en modelos con funciones discontinuas sin ningún esfuerzo de modelado o programación adicional.

La principal debilidad de estas técnicas es su baja performance en presencia de restricciones, en particular de igualdad, dado que, al reducirse dramáticamente la región factible del problema, se limita considerablemente la eficiencia de exploración de estos algoritmos.

Las metaheurísticas basadas en poblaciones se han tornado muy populares en los últimos años (Rangaiah, 2010). Entre ellas se destacan los “algoritmos genéticos”, la optimización con “colonias de hormigas” y la optimización con “enjambres de partículas”.

En particular, se adoptó esta última como método base para utilizar en esta tesis. Esta decisión se tomó, principalmente, debido a la sencillez de implementación que presenta y al buen desempeño reportado en diferentes aplicaciones (Marini y Walczak, 2015). Asimismo, se tuvo en cuenta su gran flexibilidad, ya que este método permite incorporar

fácilmente distintas técnicas para tratar restricciones, manipular variables binarias, considerar múltiples objetivos, así como también hibridizarlo con algoritmos de búsqueda local.

2.2 Optimización por enjambre de partículas (PSO)

PSO es una metaheurística propuesta por Kennedy y Eberhart (1995), inspirada en el comportamiento de comunidades de individuos en la naturaleza, tales como cardúmenes de peces, bandadas de aves o enjambres de insectos en busca de alimento.

Por ejemplo, un enjambre de abejas a la hora de alimentarse, se traslada hacia la región del espacio donde existe más probabilidad de encontrar polen. Inicialmente, cada abeja vuela erráticamente por la zona, intentando identificar aquellas zonas con mayor densidad de flores y recordando el mejor lugar visitado hasta entonces. A su vez, los individuos intercambian información con otros miembros del enjambre para que toda la población conozca la ubicación del sector con mayor cantidad de alimento. De esta manera, cada individuo va modificando su trayectoria sobre la base de su propia experiencia y de la información adquirida a través de sus vecinos. Si alguno de estos insectos, durante su exploración, encuentra un territorio más abastecido de flores que la conocida hasta el momento, entonces todos los individuos se dirigirán hacia ese nuevo lugar con alguna probabilidad. Esto se repetirá sucesivamente a medida que se encuentren mejores lugares para recolectar hasta que, finalmente, todas las abejas se agrupen en torno al mismo punto al final del recorrido.

Estas ideas han sido incorporadas al campo de la optimización en forma de algoritmo, el cual se emplea en la actualidad para resolver distintos tipos de sistemas. A lo largo del tiempo se lo ha ido mejorado con aportes de distintos investigadores, como por ejemplo a través de variaciones a su parametrización, combinación con otras metaheurísticas y diferentes tipos de hibridaciones.

En PSO, las partículas que pertenecen a una población de soluciones candidatas, se sitúan en el espacio de búsqueda y se mueven a través del mismo de acuerdo a ciertas

reglas. Las mismas tienen en cuenta la mejor posición local encontrada por cada individuo hasta el momento y la mejor posición global descubierta por el enjambre.

La mejor posición local se puede asemejar a la memoria autobiográfica, ya que cada individuo “recuerda” su propia experiencia. El ajuste de velocidad asociado con esta ha sido llamado "nostalgia simple" porque el individuo tiende a regresar al lugar que más satisfizo en el pasado. Por otro lado, la mejor posición global es conceptualmente similar al conocimiento publicitado, es decir, a una norma o estándar grupal que cada partícula busca alcanzar.

Mientras se descubran nuevas y mejores posiciones, éstas pasan a determinar los movimientos de los individuos. La actualización de la velocidad y la posición se repite iterativamente hasta cumplir algún criterio de convergencia, típicamente suele ser alcanzar un número máximo de iteraciones. En la Fig. 2.1 se esquematiza este proceso: los puntos representan las partículas, las flechas sus respectivas velocidades y la estrella el óptimo.

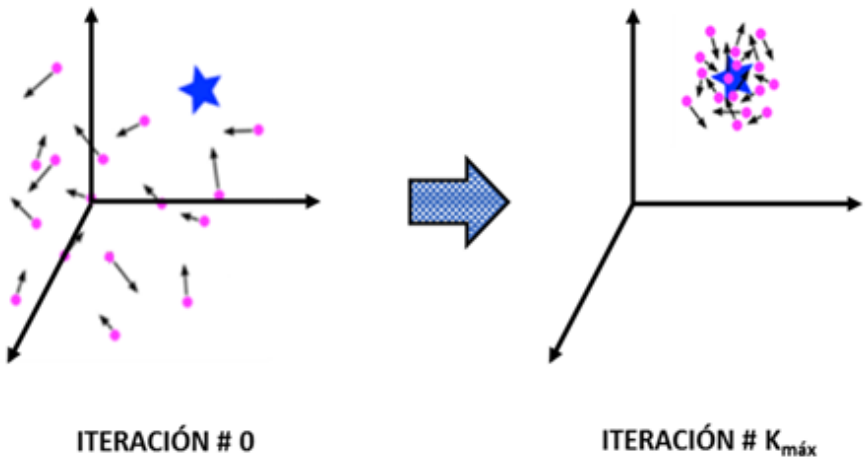

Fig. 2.1 Descripción gráfica del PSO

Un concepto clave para evaluar el desempeño en este tipo de técnicas de optimización es el balance entre explotación y exploración del espacio de búsqueda. Por un lado, la explotación guía la búsqueda teniendo en cuenta la información que se obtiene de las mejores soluciones encontradas hasta el momento. En cambio, la exploración permite

descubrir nuevas regiones, intentando evitar así una convergencia prematura. Los parámetros de este algoritmo deben ser cuidadosamente seleccionados a fin de lograr un buen balance entre esos dos objetivos y asegurar así una buena tasa de convergencia al óptimo global.

Como se ha mencionado anteriormente, PSO y otros algoritmos de enjambre tienen una gran debilidad relacionadas con el manejo de las restricciones, en particular las de igualdad. Por esta razón, gran parte de la investigación que se desarrolla actualmente sobre estas técnicas apunta a mejorar la capacidad de convergencia del algoritmo en espacios de búsqueda complejos y restringidos. Asimismo, PSO es computacionalmente intensivo, ya que generalmente se requiere un gran número de evaluaciones de la función objetivo y de las restricciones para lograr la convergencia.

Algunos ejemplos notables de mejoras en la PSO básica son: Ali y Kaelo (2008), Kayhan y col. (2010) y Chen y Chi (2010) para abordar la convergencia global, y Yiquing y col. (2007) y Shokrian y High (2014) para manipular restricciones.

En lo que respecta a aplicaciones, este algoritmo se ha utilizado en diversos sistemas de escala realista para diferentes disciplinas. Sólo por mencionar algunos, Adams y Seider (2008) informaron sobre el uso de PSO para la optimización dinámica de procesos químicos complejos. Montain y col. (2015) lo utilizaron como un motor de optimización multi-objetivo para un modelo fisiológico de interés médico. Por otro lado, Erbeyoglu y Bilge (2016) aplicaron esta metaheurística al problema de mezclado, un sistema desafiante de la industria del petróleo. Chen y col. (2017) presentaron un modelo lineal mixto entero para optimizar la cadena de suministro de la industria de celdas solares y Lang y Zhao (2016) también propusieron un modelo basado en esta técnica para la programación de producción en la industria petrolera. Todas las referencias anteriores sugieren el potencial de PSO para abordar problemas complejos de optimización de interés académico y práctico.

2.2.1 Formulación matemática

Considere un espacio de búsqueda de dimensión D y un enjambre conformado por N partículas. La matriz $\boldsymbol{x}$ representa las posiciones de las partículas en el espacio de búsqueda (Ec. 2.1).

$$\boldsymbol{x} = x_{ij} = \left[x_{11} \dots x_{1j} \dots x_{1D} \ \dots \ x_{i1} \dots \ x_{ij} \dots x_{iD} \ \dots \ x_{N1} \dots \ x_{Nj} \dots x_{ND}\right]$$
$$\forall\, i \in [1, N], \forall\, j \in [1, D] \qquad (2.1)$$

En el proceso para encontrar la solución óptima, cada partícula actualiza su posición x_{ij} al moverse por el espacio de acuerdo a una trayectoria dada por (Ec. 2.2):

$$x_{ij}^{k+1} = x_{ij}^{k} + v_{ij}^{k+1} t \qquad \forall\, i \in [1, N], \forall\, j \in [1, D] \qquad (2.2)$$

Siendo k la iteración actual, v_{ij} la velocidad de la partícula "i" a lo largo de la dimensión "j"; t es la fracción de tiempo que arbitrariamente se asume igual a la unidad.

La velocidad de cada partícula se define según la Ec. 2.3 como:

$$v_{ij}^{k+1} = w^k v_{ij}^{k} + c_1 r_{1_{ij}} \left(p_{ij}^{k} - x_{ij}^{k}\right) + c_2 r_{2_{ij}} \left(q_j - x_{ij}^{k}\right) \qquad \forall\, i \in [1, N], \forall\, j \in [1, D] \quad (2.3)$$

Donde p_{ij} es el mejor personal, la mejor posición alcanzada por ese individuo a lo largo de su recorrido; q_j es el mejor social, la mejor solución encontrada hasta el momento por todo el enjambre o parte de él, según la topología que se decida adoptar (las cuales se detallarán en la siguiente sección). Los símbolos w, c_1, c_2, $r_{1_{ij}}$ y $r_{2_{ij}}$, cuya función se detalla más abajo, representan los parámetros aleatorios que brindan la estocasticidad al método de búsqueda. Conceptualmente, los tres términos que componen la velocidad representan:

1. *Inercia o impulso*: previene que la partícula cambie drásticamente de dirección, ya que mantiene un registro de la orientación anterior $\left(wv_{ij}\right)$.

2. *Componente cognitivo*: tiene en cuenta la tendencia de la partícula para volver a la mejor posición previamente encontrada por ella misma $\left(c_1 r_{1_{ij}} \left(p_{ij} - x_{ij}\right)\right)$.

3. *Componente social*: determina la capacidad de la partícula de moverse hacia la mejor posición encontrada por todo el enjambre $\left(c_2 r_{2_{ij}} \left(q_j - x_{ij}\right)\right)$.

En la Fig 2.2 se representa esquemáticamente el cálculo de la nueva posición de una partícula en función de estos tres componentes.

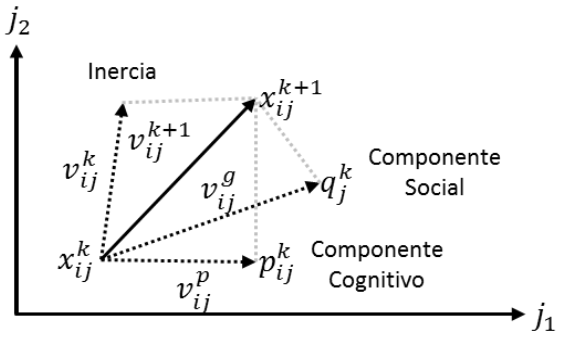

Fig. 2.2 Descomposición de la velocidad

2.2.2 Inicialización y parametrización del algoritmo

El PSO posee parámetros que rigen el comportamiento de las partículas y condicionan la eficiencia que tendrá el algoritmo para resolver un cierto problema de optimización. Existe una abundante bibliografía relacionada con la parametrización del PSO. En Engelbrecht y Cleghorn (2018), por ejemplo, se presenta un estudio reciente sobre muchos aspectos prácticos asociados a la selección de los parámetros. A continuación, se describen los mismos, junto con los criterios adoptados para su selección en este trabajo:

Tamaño del enjambre (N)

Es un factor que puede tener un gran impacto en la robustez y el costo computacional del algoritmo. Un enjambre muy grande suele provocar una mayor capacidad de exploración, pero al costo de una convergencia más lenta y un mayor esfuerzo computacional. En cambio, una población excesivamente pequeña puede conducir a convergencia local.

Es necesario definir un tamaño de población adecuado para mantener la eficacia del algoritmo y, como se mencionó, lograr un buen balance entre la exploración y explotación del mismo. Algunos autores (Mussi y col., 2011) han sugerido un tamaño de población en función del número de variables, por ejemplo, $N = 10 + \sqrt{D}$.

Otros trabajos sugieren que el desempeño del algoritmo es insensible a la cantidad de partículas para N mayor a 50 para modelos de diversa dimensión (Marini y Walczak, 2015). Por esta razón, en esta tesis se adoptó $N = 50$ para todos los experimentos realizados.

Topologías PSO

La topología, estructura social, o vecindad, establece la forma en la que las partículas se comunicarán. Las diferentes topologías afectan la velocidad y el grado en el que la población es atraída hacia una cierta región, influyendo en el rendimiento del algoritmo. A continuación, se describen las dos estrategias más utilizadas según Kennedy (1999):

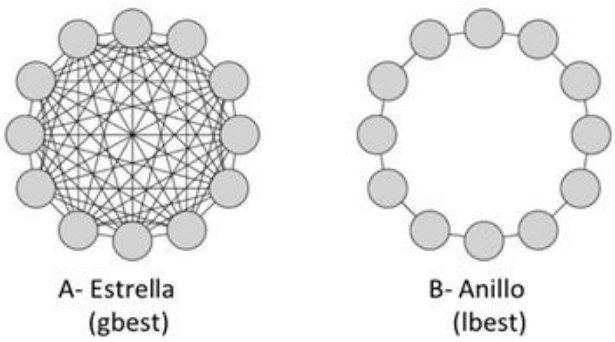

Fig. 2.3 Topologías PSO: a) Estrella, b) Anillo

Estrella: conocida también como “gbest” o mejor global. Debido a su estructura social, todas las partículas están interconectadas, comunicándose entre sí (Fig. 2.3.a). En este caso, cada individuo es atraído hacia la mejor solución encontrada por todo el enjambre. Por lo tanto, cada partícula imita la mejor solución general. Se ha demostrado que, con esta topología, el PSO converge más rápido que con otras estructuras de red, con el riesgo de quedar atrapado en los mínimos locales.

Anillo: conocida también como “lbest” o mejor local. En esta topología, cada partícula es afectada por el mejor desempeño de sus "n" vecinos inmediatos de la población. En el caso más común, $n = 2$, (Fig. 2.3.b) el individuo es afectado sólo por sus vecinos inmediatamente adyacentes. Por lo tanto, es posible que un segmento de la población pueda converger en un óptimo local, mientras que otro segmento de la población pueda hacerlo en un punto diferente o seguir buscando. Presenta en general una convergencia lenta.

No existe una topología específica que supere la performance del resto en todos los casos de estudio. Es decir, la elección de la misma depende del modelo que se desea optimizar. En el presente trabajo se optó por la topología estrella o “gbest”, ya que existe evidencia en la literatura de que presenta buen desempeño para una amplia gama de modelos.

Actualización de las partículas: PSO sincrónico y asincrónico

Como se explicó anteriormente, en el PSO las partículas actualizan sus vectores de posición y velocidad de acuerdo con su inercia, su memoria y el conocimiento social. En función del instante dentro del proceso iterativo en el que se realiza la actualización de la memoria local y grupal, se puede distinguir entre PSOs con actualizaciones sincrónica y asincrónica de la población. En la primera, todas las partículas se mueven en paralelo y comparten información. En cada iteración se evalúa el la función objetivo de cada individuo y se actualiza el mejor local y global, luego, todo el enjambre se desplaza teniendo en cuenta esta nueva información.

En cambio, en el modelo asincrónico, en cada iteración la partícula se desplaza con la información actualizada por sus inmediatos predecesores, es decir, no espera a que se determine el mejor global entre todas las partículas, sino que utiliza el valor calculado por las partículas previas. Es decir, en cada iteración k, la partícula i-ésima se desplaza hacia un nuevo punto x_{ij}^{k+1} utilizando la información del vector q_j^k, actualizado por las $i-1$ partículas previas. Luego, ese individuo evalúa la calidad del nuevo punto encontrado y actualiza el mejor local y global, que será transmitido a las restantes partículas.

De acuerdo con un estudio reciente (Englebrecht y Cleghorn, 2018), si se analiza la performance general de los dos tipos de modelos, no se encuentran diferencias significativas entre ellos. Por dicha razón y considerando que la naturaleza del PSO sincrónico lo hace susceptible de ser ejecutado en paralelo sobre múltiples procesadores, se optó por programar un PSO sincrónico en esta tesis.

Criterios de terminación

Los criterios de terminación establecen un punto en el tiempo para finalizar un ciclo de optimización. Son muy importantes dado que se busca evitar una terminación prematura y a la vez un cómputo excesivo. Existen diversos criterios de terminación, tal como describe Jain y col. (2001). Los más comunes e intuitivos son:

- *Máximo número de iteraciones*: el algoritmo se ejecuta hasta que supera el número máximo de iteraciones especificado.

- *Máximo tiempo de ejecución*: consiste en finalizar la búsqueda del algoritmo si éste supera el tiempo máximo establecido por el usuario.

- *Grado de avance del algoritmo*: el algoritmo se ejecuta hasta que no existe una mejoría notoria en la solución (Ec. 2.4):

$$\left| fg - \left(\frac{1}{RG}\right) fgs \right| \leq \varepsilon \qquad (2.4)$$

Dónde fg es la mejor solución hallada hasta el momento, fgs es la suma de las mejores soluciones encontradas en las últimas RG iteraciones y ε es el valor mínimo de aproximación entre la solución actual (fg) y el promedio de las anteriores (fgs).

Habitualmente se combinan dos o más criterios de terminación para asegurar una convergencia apropiada y un tiempo de ejecución razonable. En este trabajo se adoptó el máximo número de iteraciones para finalizar la ejecución de los experimentos, a fin de poder comparar de una manera más justa los tiempos computacionales de las versiones serie y paralelo del PSO.

Posición inicial (x_{ij}^0)

La posición inicial de cada partícula (Ec. 2.5) se generará aleatoriamente de acuerdo a una distribución uniforme (U) comprendida entre el límite inferior (x_{ij}^{lo}) y superior (x_{ij}^{up}) de cada variable.

$$\boldsymbol{x} = x_{ij} = U\left(x_{ij}^{lo}, x_{ij}^{up}\right) \qquad \forall\, i \in [1, N], \forall\, j \in [1, D] \qquad (2.5)$$

De esta manera, se logra que la población cubra, al inicio, lo más uniformemente posible toda la región de búsqueda y fomente una mayor exploración, lo que favorecería la convergencia del algoritmo.

Velocidad inicial (v_{ij}^0)

Existen varias estrategias para inicializar la velocidad de las partículas. Una opción es un valor aleatorio comprendido entre los límites superiores e inferiores de la misma. Sin embargo, en este trabajo se adopta que la velocidad inicial de cada partícula sea igual a

cero, ya que se cree que con la Ec. 2.5 se logra una conveniente distribución de la población inicial en la caja.

Velocidad máxima (v_j^{max})

Para evitar que las partículas crucen los límites del espacio determinado por la caja ($x_{ij}^{lo} \leq x_{ij} \leq x_{ij}^{up}$), se introducen las Ec. 2.6 y Ec. 2.7 para restringir la velocidad:

$$Si\ v_{ij}^{k+1} > v_j^{max}\ entonces: v_{ij}^{k+1} = v_j^{max} \qquad \forall\, i \in [1, N], \forall\, j \in [1, D] \qquad (2.6)$$

$$Si\ v_{ij}^{k+1} < -v_j^{max}\ entonces: v_{ij}^{k+1} = -v_j^{max} \qquad \forall\, i \in [1, N], \forall\, j \in [1, D] \qquad (2.7)$$

Donde v_j^{max} es la velocidad máxima de la dimensión "j".

Estos límites contribuyen a suavizar las oscilaciones y a que cada miembro del enjambre no se mueva más allá del espacio establecido para cada variable, facilitando así la convergencia al evitar explorar zonas muy alejadas de la región factible. Si bien definir el parámetro v_j^{max} no es simple, muchos autores coinciden en calcularlo como sigue (Ec. 2.8):

$$v_{ij}^{max} = 0.90 \frac{\left(x_{ij}^{up} - x_{ij}^{lo}\right)}{2} \qquad \forall\, j \in [1, D] \qquad (2.8)$$

Donde x_{ij}^{up} es el límite superior de la partícula y x_{ij}^{lo} el inferior.

Peso de inercia (w)

Este factor, junto con las Ec. 2.6 a 2.8, evita la divergencia del enjambre. Si el peso de inercia toma un valor superior a uno, favorece la exploración global, mientras que, si es menor a uno, domina la explotación local. En particular, el peso de inercia controla el momento de la partícula, ya que modula el efecto de la velocidad previamente calculada que establece, en parte, el valor de la nueva velocidad (Ec. 2.3).

Se han propuesto muchas formas de calcular el peso de inercia. Marini y Walzac (2015), por ejemplo, revisan seis modelos diferentes. Cada variante presenta diversas prestaciones.

Por ejemplo, varios autores han implementado un peso de inercia que disminuye linealmente desde un máximo de 0.9 hasta un mínimo de 0.4, con lo que han logrado muy buenos resultados en varias aplicaciones. Por otra parte, Bansal y col. (2011) demostraron que un peso de inercia aleatorio es mejor si se desea una convergencia más rápida, mientras que uno linealmente decreciente conduciría aproximadamente a la misma solución en todas las evaluaciones. De todas maneras, las estrategias de peso de inercia constante y linealmente decreciente fueron las que alcanzaron, en líneas generales, mejores resultados.

Si bien en el transcurso de este es trabajo se programaron varias de las formas de calcular el factor de inercia, se optó por utilizar el modelo más simple, determinando que *w* sea un parámetro constante e igual a 0.75 para todos los modelos evaluados. Este valor es ampliamente utilizado en experimentos con PSO y muchos autores reportan muy buenos resultados en una amplia gama de problemas.

Constantes de aceleración (c_1 y c_2)

Controlan la extensión de desplazamiento de las partículas, ya que modulan las contribuciones relativas del componente social y cognitivo. Diversos autores concuerdan en que cuando éstas constantes son grandes, las partículas se acercan más al óptimo, mientras que valores más pequeños resultan en patrones sinusoidales. En general, se ha demostrado que el algoritmo funciona bien para la mayoría de las aplicaciones si se especifica c_1 y c_2 tal como se describe en la Ec. 2.9:

$$c_1 = c_2 = 1.5 \tag{2.9}$$

Factores $r_{1_{ij}}$ y $r_{2_{ij}}$

Son dos matrices de números aleatorios con una distribución uniforme entre 0 y 1, provocando así que tanto el componente cognitivo como el social tengan cierta influencia estocástica cuando se actualiza la velocidad.

2.2.3 Tratamiento de restricciones

El PSO descripto por las Ecs. (2.2) y (2.3) no propone un tratamiento explícito de las restricciones dado que $\boldsymbol{q}$ y $\boldsymbol{p}$ hacen referencia implícita a soluciones factibles. Debido que prácticamente todos los problemas de interés en ciencia e ingeniería no sólo cuentan con una función objetivo y restricciones de caja (representadas por los límites superiores e inferiores de las variables), sino que también poseen restricciones de igualdad y desigualdad que limitan la región factible, es necesario implementar un método que permita incluirlas en la optimización.

Las restricciones de igualdad son las más difíciles de manejar debido a que generan una región factible más pequeña que las de desigualdad. En principio, pueden ser utilizadas para reducir el número de variables en el problema de optimización a través de un proceso de eliminación. Sin embargo, este proceso requiere de la reformulación del problema original.

Los métodos para manejar las restricciones en optimización global estocástica han recibido mucha atención en los últimos años. Coello (2002) proporciona una revisión de las técnicas más populares que se utilizan actualmente en los algoritmos evolutivos. Entre ellos se encuentran las funciones de penalización, las que separan los objetivos de las restricciones, aquellas que poseen representaciones y operadores especiales, algoritmos de reparación y determinados métodos híbridos, entre otros. Muchas de estas técnicas pueden incorporarse sin dificultades al PSO.

Cada método cuenta con ventajas y desventajas que lo hacen más o menos eficiente para determinados problemas, por lo que resulta un desafío optar por un enfoque

general que maneje todas las restricciones que pueden existir con sus diversas complejidades.

Por ejemplo, el enfoque tal vez más popular es aquel que considera una función de penalidad, ya que es intuitivo y muy fácil de implementar. La idea es transformar un problema de optimización restringido en uno no restringido agregando cierto valor a la función objetivo, según la cantidad y magnitud de las violaciones a las restricciones presentes. El inconveniente de esta técnica consiste en la dificultad de elegir un "peso" conveniente para asignar a las penalizaciones, dado que no existen parametrizaciones universales y por lo tanto su elección "óptima" suele ser problema-dependiente.

En este trabajo se adoptó el método para el manejo de restricciones propuesto en Zhang y Rangaiah (2012) en el contexto de un algoritmo de evolución diferencial. Se trata de una técnica bastante práctica, ya que no requiere de la reformulación o reducción de restricciones y es menos dependiente de parámetros si se la compara, por ejemplo, con las funciones con penalidades.

El procedimiento consiste en relajar las restricciones de igualdad y desigualdad para aumentar temporalmente la región factible. A medida que los individuos de la población van ingresando a la región relajada, ésta se va reduciendo hasta que recupera la región factible original.

El total de violaciones de cada partícula a las restricciones se contabilizan con la variable $\boldsymbol{TAV}$ (Violación Absoluta Total, por sus siglas en inglés) definida en Ec. 2.10. Cabe aclarar que la transgresión a los límites máximos y mínimos de cada dimensión también se tienen en cuenta en el cálculo de $\boldsymbol{TAV}$, ya que se las trata como restricciones de desigualdad:

$$TAV_i = \sum_{s=1}^{m1} |h_s(\boldsymbol{x})| + \sum_{s=1}^{m2} max\big(0, g_s(\boldsymbol{x})\big) + \sum_{j=1}^{D} max(0, (x_j^{lo} - x_{ij}^k)) + \sum_{j=1}^{D} max(0, (x_{ij}^k - x_j^{up})) \qquad \forall\, i \in [1, N] \qquad (2.10)$$

Donde $h_s(x)$ representa las restricciones de igualdad; m_1 es la cantidad total de restricciones de igualdad; $g_s(x)$ son las restricciones de desigualdad; m_2 es la cantidad total de restricciones de desigualdad; x_j^{lo} y x_j^{up} son el límite inferior y superior respectivamente, en la dimensión "j".

Si el TAV de cada miembro de la población es menor que el valor de relajación de las restricciones (μ) impuesto en esa iteración ($TAV_i < \mu^k$), se lo considera una solución factible, caso contrario, resulta infactible (Fig. 2.4).

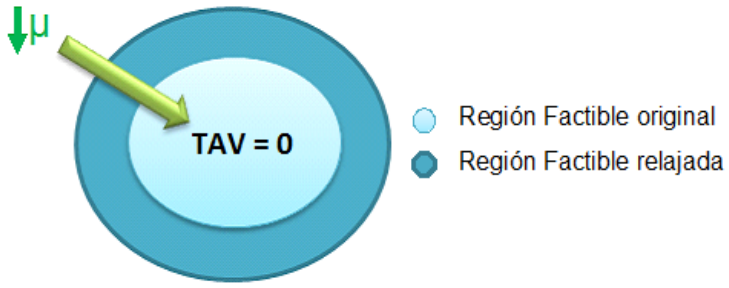

Fig. 2.4 Relajación de la región factible para el tratamiento de restricciones

El parámetro μ se reduce de acuerdo a alguna ley apropiada. En este trabajo se utiliza una versión modificada de la fórmula utilizada en Zhang y Rangaiah (2012). En concreto, μ se reduce dinámicamente basándose en el número de partículas factibles (F_F), según la Ec. 2.11:

$$\mu^{k+1} = \mu^k \left(1 - \frac{F_F}{N}\right) \tag{2.11}$$

De acuerdo a este enfoque de factibilidad, para poder evaluar qué partícula elegir para reemplazar el mejor global y el mejor individual en cada iteración del algoritmo, se considera que:

- Una solución factible es preferida a una infactible.
- Entre dos soluciones factibles, aquella que alcanzó un mejor valor de la función objetivo es preferida.

- Entre dos soluciones infactibles, es escogida la que posee menor TAV.

Esta manera de tratar las restricciones se prefirió por su generalidad e independencia de parámetros de ajuste. Además, el hecho de forzar gradualmente a la población hacia la región factible, permitiendo violaciones transitorias de las restricciones, tiende a favorecer una exploración conveniente del espacio de búsqueda. A pesar de que esta estrategia fue originalmente propuesta para algoritmos de "evolución diferencial", es directamente implementable bajo PSO y cualquier otra técnica evolutiva.

2.2.4 Implementación propia

El PSO planteado por Marini y Walczak (2015), se tomó como base para la implementación propia del algoritmo. El mismo se modificó de acuerdo a la técnica de manejo de restricciones ya comentada de Zhang y Rangaiah (2012) con algunas modificaciones. En la Fig. 2.5 se presenta el diagrama de flujos correspondiente.

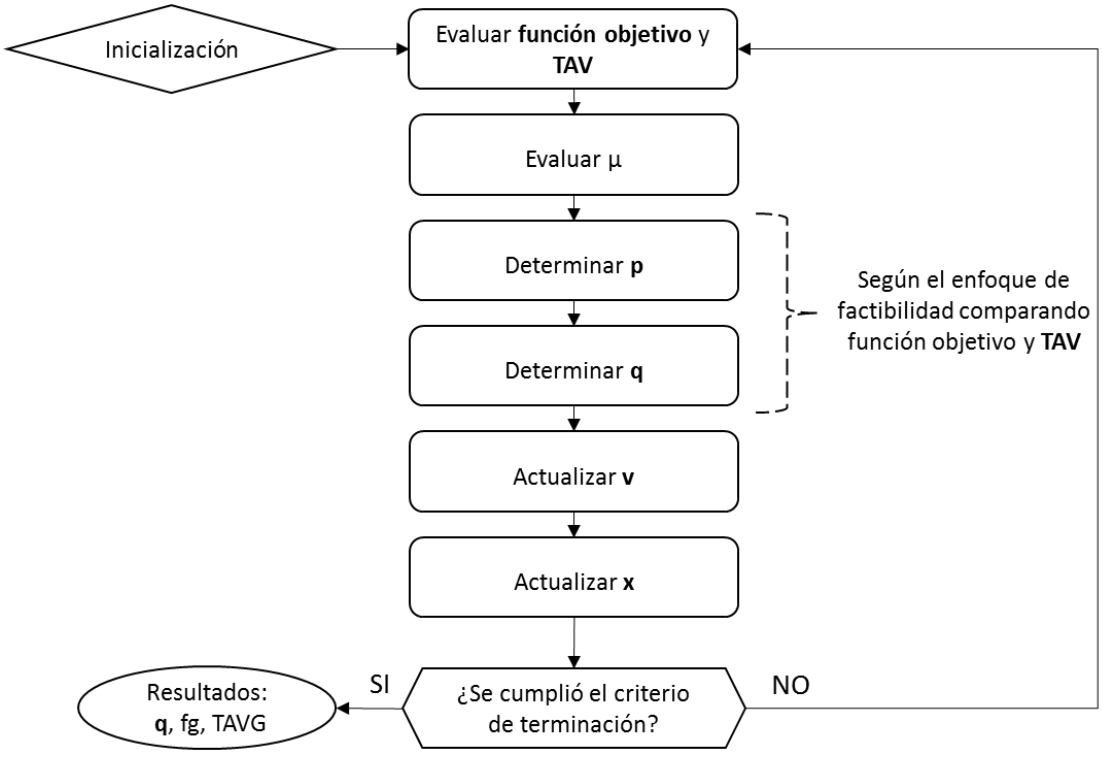

Fig. 2.5 Esquema de representación de la implementación propia del PSO

La metodología descrita fue programada en el lenguaje de programación Python, haciendo uso de las capacidades proporcionadas por la biblioteca NumPy[1] para producir una versión en serie competitiva del algoritmo. La Fig. 2.6 proporciona en formato de pseudocódigo el algoritmo implementado.

1. Inicialización de cada partícula del enjambre:
   a. Inicializar la posición aleatoriamente: $x_{ij}^0 = U\left(x_{ij}^{lo}, x_{ij}^{up}\right)$
   b. Inicializar la velocidad en cero: $v_{ij}^0 = 0$
   c. Inicializar la mejor posición local: $p_{ij}^0 = x_{ij}^0$.
   d. Calcular el valor de la función f_i y de TAV_i. Establecer como mejor global $\boldsymbol{q}$ la partícula que cumple con el criterio de factibilidad.
   e. Calcular el valor inicial μ^0 como un promedio de los $\boldsymbol{TAV}$ del enjambre.
2. Repetir los siguientes pasos hasta que se cumpla alguno de los criterios de terminación elegidos:
   a. Actualizar la velocidad de cada partícula:
   $$v_{ij}^{k+1} = w^k v_{ij}^k + c_1 r_{1\,ij}\left(p_{ij}^k - x_{ij}^k\right) + c_2 r_{2\,ij}\left(q_j - x_{ij}^k\right)$$
   b. Actualizar la posición de cada partícula:
   $$x_{ij}^{k+1} = x_{ij}^k + v_{ij}^{k+1}$$
   c. Evaluar el valor de la función $f\left(x_i^{k+1}\right)$ y de $TAV\left(x_i^{k+1}\right)$ para cada partícula en su nueva posición.
   d. Evaluar el valor de relajación: $\mu^{k+1} = \mu^k\left(1 - \frac{F_F}{N}\right)$
   e. Actualizar el mejor local: $p_{ij} = x_{ij}^{k+1}$, de acuerdo al criterio de factibilidad.
   f. Actualizar el mejor global: $q_j = x_{ij}^{k+1}$, de acuerdo al criterio de factibilidad.
3. Al finalizar el proceso iterativo, la mejor solución estará representada por el mejor global (q_j).

Fig. 2.6. Pseudocódigo para PSO

[1] www.numpy.org

El algoritmo propuesto se testeó en diversas funciones benchmark con diferente grado de dificultad, así como en distintas aplicaciones más complejas de ingeniería química (Capítulos 4 y 5).

Es necesario aclarar que no es el propósito de este trabajo presentar mejoras en el algoritmo básico del PSO ni tampoco incorporar una técnica de manejo de restricciones novedosa. El objetivo de esta tesis, en cuanto a este aspecto, fue poder implementar una versión propia de esta técnica metaheurística, adoptando una parametrización estándar, que permita la identificación de buenas soluciones factibles en los problemas que optimice.

CAPÍTULO 3

PARALELIZACIÓN EN GPU

En este capítulo se explica la estrategia de paralelización adoptada del algoritmo de optimización detallado en el Capítulo 2. Específicamente se hace uso de una Unidad de Procesamiento Gráfico (GPU por sus siglas en inglés), la cual es un dispositivo común de las computadoras personales modernas. Asimismo, se comenta el funcionamiento de PyCUDA, que es el modelo de programación elegido, el cual permite utilizar la GPU aprovechando los beneficios del lenguaje de programación Python, descritos anteriormente. Finalmente, se detalla la implementación propia del PSO con todas sus consideraciones y se describe la métrica seleccionada para probar la eficiencia del algoritmo paralelizado.

3.1 Introducción

En los últimos años se han comenzado a desarrollar arquitecturas de procesamiento que dejan atrás el enfoque clásico de programación secuencial. Gracias a los avances tecnológicos, actualmente existen otros métodos para reducir el costo computacional de un procedimiento. En el nivel más alto de abstracción se encuentra la optimización del código del algoritmo, ya que modificando su diseño es posible lograr un mayor rendimiento aprovechando al máximo los recursos disponibles. En un nivel más bajo de abstracción, se puede optar por la programación paralela. La misma consiste en dividir una tarea en varias más pequeñas e independientes entre sí, y ejecutarlas de forma concurrente a través de varios procesadores, a diferencia del procesamiento secuencial en el que cada tarea se ejecuta una después de la otra, resultando en un proceso en general más lento (Fig. 3.1).

Durante la última década, el desempeño de los procesadores tradicionales se ha estancado (Patterson, 2010). Debido a dicha realidad, los arquitectos de computadoras se han inclinado hacia los procesadores multi-núcleo. Este tipo de arquitecturas son

naturalmente paralelas. Además de ese cambio radical en los procesadores de propósito general (CPUs), se ha podido observar la introducción disruptiva de los aceleradores. En particular, las GPUs muestran excelentes rendimientos en cómputos matriciales (Nickolls y Dally, 2010).

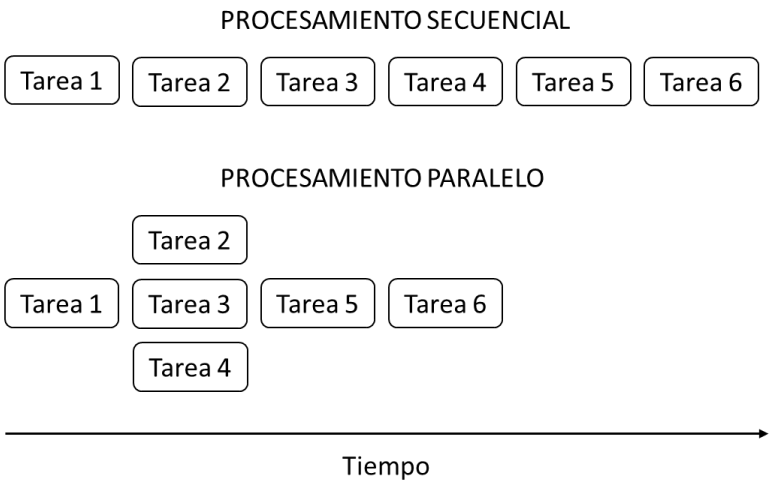

Fig. 3.1: Ejecución procesamiento secuencial versus paralelo

Debido a su arquitectura "masivamente paralela", las GPUs pueden incrementar el rendimiento de algoritmos de propósito general y no sólo de aquellos orientados a visualización gráfica. Esta técnica de usar la GPU para aplicaciones que tradicionalmente se ejecutaban en la CPU se denomina GPGPU (del inglés General Purpose Computing on GPUs).

La capacidad de cómputo de las GPU que permiten la implementación de cálculos en paralelo, ha alcanzado un desarrollo notable en los últimos años. Estos componentes han crecido de la mano de un fuerte incremento en la producción y demanda de dispositivos que las integran, tales como smartphones, tablets, computadoras personales, y, en particular, de la industria de los juegos.

La arquitectura de las GPU ha llamado la atención de la comunidad científica como una gran oportunidad para mejorar notablemente los tiempos de cómputo de algunos

algoritmos secuenciales que no logran resolver ciertos problemas en tiempos aceptables.

Las metaheurísticas de enjambre de partículas usualmente requieren un gran número de evaluaciones de funciones para lograr la convergencia, incluso en problemas de tamaño mediano o pequeño. Dado que esta clase de optimizadores basados en poblaciones son altamente paralelizables, la mayoría de estos algoritmos fueron programados casi inmediatamente para explotar todo tipo de arquitecturas paralelas disponibles. En particular, el desarrollo de la tecnología de GPU proporciona una forma práctica y de bajo costo para implementar tal paralelismo a escala de escritorio.

3.2 Unidad de Procesamiento Gráfico (GPU)

Una GPU, es un coprocesador que se dedica exclusivamente al procesamiento de gráficos y operaciones de "punto flotante". El diseño orientado a operaciones matriciales, le permite resolver problemas de álgebra lineal de forma muy eficiente. En las situaciones en que el tipo de problema lo justifica, las operaciones con mayor carga computacional se realizan en la GPU, mientras que el resto del código se ejecuta en la CPU (Fig. 3.2).

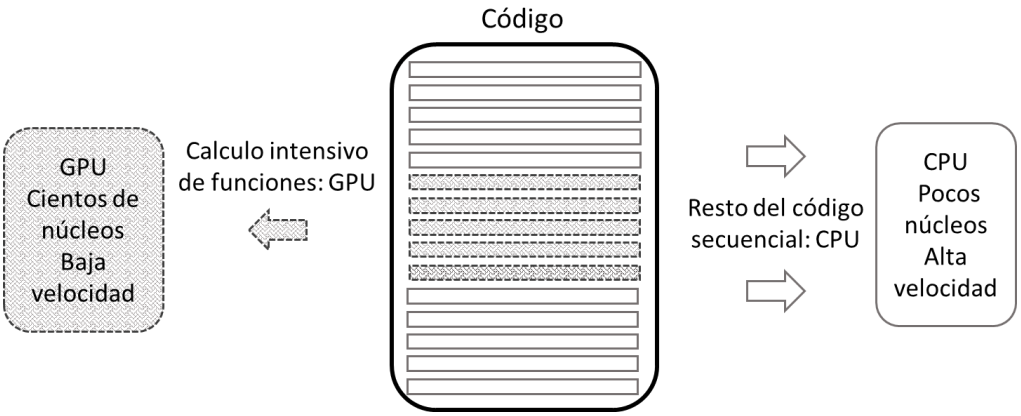

Fig. 3.2: Operaciones en GPU y CPU

Por su parte, la Fig. 3.3 muestra las principales diferencias en la arquitectura de la CPU y de la GPU, en donde “Control” se refiere a la unidad que busca las instrucciones en la

memoria principal, las decodifica y las ejecuta con la unidad de proceso, “DRAM” es la memoria dinámica de acceso aleatorio, “CACHE” es la memoria de acceso rápido y “ALU” es la unidad aritmética lógica. Como puede observarse, la GPU cuenta con más unidades de procesamiento que la CPU, aunque éstas son más sencillas.

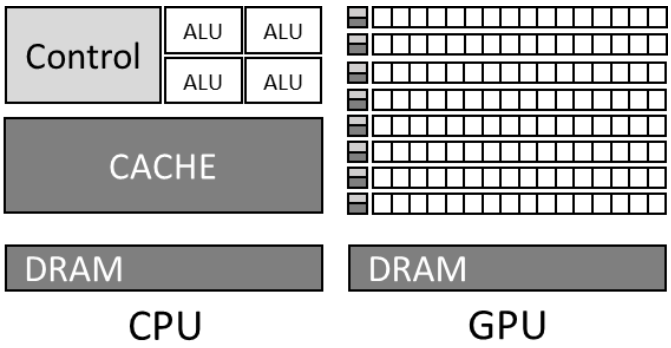

Fig. 3.3: Comparación entre la arquitectura de una CPU y una GPU

La CPU tiene un propósito más general, además del cómputo su diseño se concentra en el manejo de datos de entrada y salida. Los núcleos se focalizan a resolver instrucciones generales en el menor tiempo posible siguiendo el modelo de Secuencia de Instrucción Única, Flujo de Datos Único (SISD por sus siglas en inglés).

Por el contrario, la GPU está optimizada para trabajar con grandes cantidades de datos sobre los cuales se realizan las mismas operaciones de forma concurrente. Dispone de una gran cantidad de núcleos de procesamiento más pequeños que, a velocidades bajas, pueden operar en paralelo permitiendo acelerar una gran cantidad de aplicaciones. Sigue el modelo de Secuencia de Instrucción Única, Flujo de Datos Múltiples (SIMD por sus siglas en inglés). Por estas razones, la GPU puede aumentar enormemente su rendimiento en comparación con la CPU. El diseño orientado a vectores (o matrices) demanda al programador escribir programas en formato específico.

Los componentes de la GPU que pueden ser programados por el usuario son principalmente hilos, bloques de hilos y grillas de bloques. Los hilos de la GPU son los elementos básicos: todos ellos ejecutan el mismo código. Una cierta cantidad de éstos,

que varía según el modelo de la GPU, se agrupan en bloques, los cuales se organizan, a su vez, en una grilla de bloques.

Cada hilo se identifica por un vector de tres componentes (x, y, z) denominado threadIdx. Entonces, los hilos pueden ser identificados por un índice threadIdx unidimensional, bidimensional o tridimensional, formando a su vez, un bloque unidimensional, bidimensional o tridimensional identificado por la variable blockIdx. Por ejemplo, en la Fig. 3.4 se muestra un arreglo bidimensional de hilos y bloques.

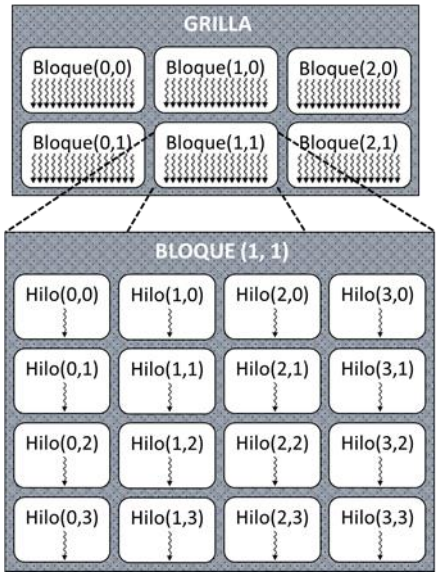

Fig. 3.4: Grilla 2D de bloques de hilos 2D

El Kernel describe las operaciones que ejecuta la GPU. Al invocarse, el programador asigna explícitamente recursos de cómputo en formato de hilos, bloques y grilla.

Por otro lado, la GPU consta de Procesadores Escalares (SP) y de Multiprocesadores de "Streaming" (SM). Los primeros son los que ejecutan los hilos y son el tipo más simple de procesador, donde cada instrucción se opera sobre un dato cada vez. Los segundos son los encargados de realizar toda la computación, su cantidad difiere según el modelo de la GPU. Cada SM puede ejecutar hasta ocho bloques de hilos en serie, los cuales admiten paralelismo a nivel de instrucción. Estos procesadores de propósito general poseen baja velocidad de transferencia de dato, contienen sus propios registros,

unidades de control, memorias caché y memoria compartida. En cuanto se haya finalizado la ejecución de un bloque de hilos, se comienza con el siguiente bloque en serie. En la Fig. 3.5 se representa cómo los SP ejecutan los hilos y los SM los bloques de hilos.

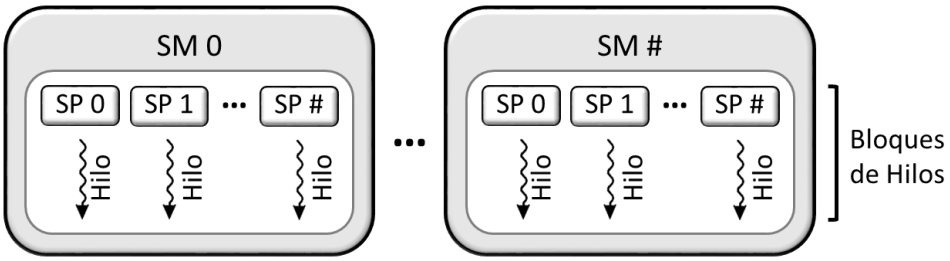

Fig. 3.5: Representación de SM y SP

Cada bloque de hilos puede ser ejecutado en el SM que esté disponible, en cualquier orden, concurrente o secuencialmente. De esta manera, un programa CUDA ya compilado puede ejecutarse en sistemas con distinto número de SM y solo el sistema de tiempo de ejecución debe conocer cuántos de ellos existen físicamente. Por ejemplo, en la Fig. 3.6 se puede observar que, en el caso del sistema que cuenta con dos SM, la velocidad de cómputo será menor que el que posee cuatro SM. Esto sucede porque la primer GPU debe distribuir la cantidad total de bloques solo entre dos SM, resultando en que cada uno de ellos ejecute cuatro bloques secuencialmente, mientras que el otro sistema ejecuta sólo dos bloques por SM.

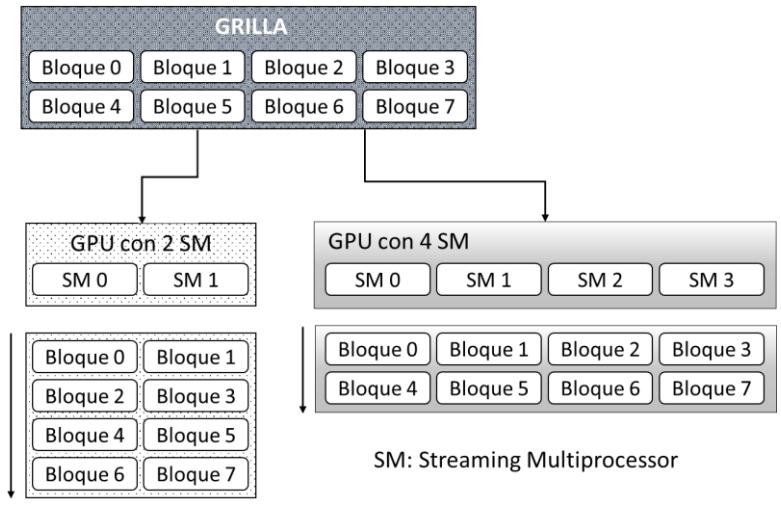

Fig. 3.6: Escalado GPU

En la Tabla 3.1 se detallan algunas características del hardware y software de las tres GPU marca NVIDIA utilizadas en esta tesis. Una buena placa de video que permita realizar este tipo de cómputos paralelos, como lo es la GTX 1060 utilizada, resulta relativamente accesible ya que su costo actualmente es de USD299 o, su versión más reciente (GTX 1080) tiene un valor de USD800.

Tabla 3.1. Características GPUs GeForce

Modelo	GTX TITAN X	GTX 1060	GTX 480
RAM	12GB	3GB	1.5GB
SM	24	10	15
Bloques	192	80	60
Hilos por Bloque	1024	1024	512
Arquitectura	Kepler	Pascal	Fermi

3.3 Modelo CUDA/PyCUDA

La empresa NVIDIA[1] es uno de los fabricantes especializados que más ha apostado por el enfoque GPGPU (Armas y col., 2011), desarrollando un modelo de programación denominado CUDA[2] (Compute Unified Device Architecture), que permite ejecutar algoritmos en sus GPU. El lenguaje de programación correspondiente es una variación del lenguaje C que contiene extensiones para trabajar con la GPU.

CUDA es una plataforma diseñada conjuntamente a nivel software y hardware para aprovechar la potencia de una GPU en aplicaciones de propósito general a tres niveles:

- *Software*: Permite programar la GPU en lenguaje C con mínimas pero potentes extensiones SIMD (Single Instruction Multiple Data) para lograr una ejecución eficiente y escalable.

[1] www.nvidia.com
[2] www.developer.nvidia.com/cuda

● *Firmware*: Ofrece un driver para la programación GPGPU que es compatible con el que utiliza para renderizar (generar una imagen o video). Sencilla interfaz de programación de aplicaciones (APIs por sus siglas en inglés) manejan los dispositivos, la memoria, etc.

● *Hardware*: Habilita el paralelismo de la GPU para programación a través de un número de multiprocesadores dotados de un conjunto de núcleos computacionales agrupados por una jerarquía de memoria.

Mientras el lenguaje de programación C es compilado, Python es un lenguaje de programación interpretado. Esta característica, le permite al programador mayor flexibilidad en la sintaxis, facilitando la productividad. PyCUDA permite programar un ambiente CUDA desde un lenguaje de programación orientado a la productividad como Python. Al utilizar PyCUDA se aprovecha, en teoría, tanto la productividad, dinamismo y simplicidad de Python, como la velocidad que CUDA permite obtener de una GPU.

Cabe aclarar que existen GPUs de otras marcas que admiten la programación en otros lenguajes (como por ejemplo OpenCL[3]). En esta tesis se adoptó la tecnología NVIDIA/CUDA por ser NVIDIA el líder en el desarrollo de GPUs en este momento y por disponer de GPUs de esta marca en el grupo de trabajo.

En la Fig. 3.7 se presenta la diferencia en el código para efectuar la suma de dos matrices en Python y en PyCUDA. En este último lenguaje, mediante la sentencia "__global__" se define el kernel, función que será invocada por cada hilo. Para distinguir e identificar cada hilo se definen los componentes "i" y "j", que indican la posición del hilo dentro del bloque, permitiendo realizar cálculos sobre la matriz de una manera más natural. Cada hilo realiza la suma de un elemento de la matriz "a" con la del elemento en "b" para guardar el resultado en "c". Este subproceso se realiza simultáneamente

[3] www.khronos.org/opencl

para todos los elementos de la matriz, evitando de esta manera la necesidad de utilizar lazos para recorrerlos disminuyendo así el tiempo de cómputo requerido.

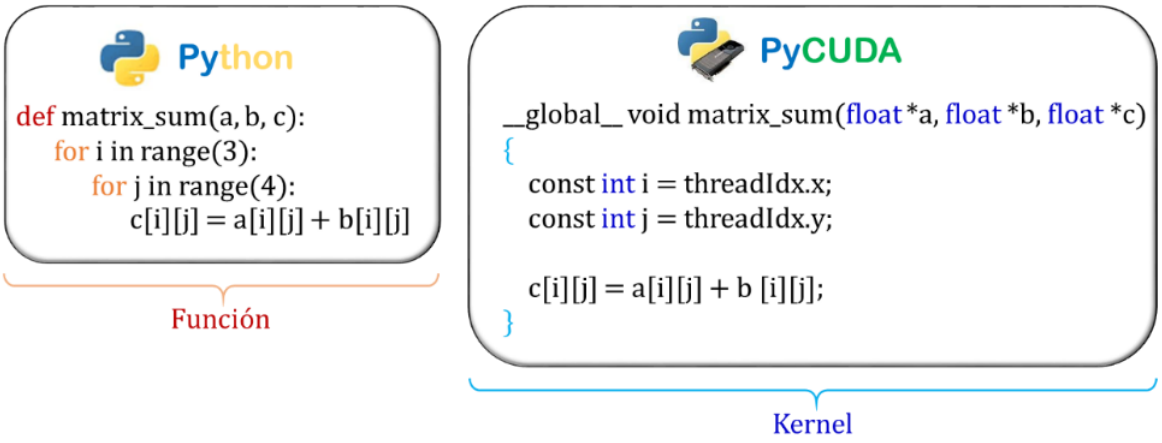

Fig. 3.7: Suma de dos matrices en Python y PyCUDA

El diseño de CUDA posee como objetivo el desarrollo de software que, de manera transparente, escale el paralelismo con un conjunto reducido de instrucciones a fin de aprovechar el incremento del número de procesadores mientras mantiene una baja curva de aprendizaje para los programadores familiarizados con lenguajes estándar como C. Según Armas y col. (2011), para cumplir con este desafío se cuenta con una jerarquía de hilos y sincronizaciones por barrera y una jerarquía de memoria, las cuales se describen a continuación:

• *Jerarquía de hilos y sincronizaciones por barrera*

La jerarquía de hilos se organiza en base a los tres elementos ya definidos: hilo (thread), bloque (block) y grilla (grid). Los mismos permiten un granulado fino del paralelismo de los datos. Esta estructura preserva la expresividad del lenguaje, logrando que los hilos cooperen en la resolución de cada subproblema, y al mismo tiempo permite la escalabilidad.

Como los distintos hilos colaboran entre ellos y pueden compartir datos, se requieren directivas de sincronización. En CUDA se puede especificar una sincronización del tipo barrera, en la que todos los hilos esperan a que los demás lleguen al mismo punto.

• *Jerarquía de memoria*

Para poder ejecutar un trabajo en una GPU se deben transferir los datos que se van a procesar desde la memoria principal de la CPU a la del dispositivo. Para ello, CUDA dispone de métodos que permiten al programador reservar memoria y copiar datos desde la CPU. Debido al elevado costo de tiempo que suponen estas transferencias de datos entre el Host y la GPU es recomendable, sino imperativo, que se reutilice la información ya copiada en la GPU y se evite, al máximo, el número de transferencias.

Es importante mantener los datos del problema en la GPU durante la ejecución. El tamaño de la memoria principal varía según los modelos, pero aun aquellos que cuentan con mayor memoria poseen una capacidad que sigue siendo limitada, por lo que es fundamental conocer cuánta memoria se está ocupando y tratar de optimizarla.

En la Fig. 3.8, puede observarse la relación que existe entre el software, las memorias y el hardware de la GPU. En primer lugar, se tiene la memoria local privada de cada hilo, los cuales son ejecutados por los procesadores escalares. Cada bloque de hilos, ejecutados en los SM, posee memoria compartida visible solo por los hilos del bloque, que pueden leer y escribir en la misma. Esta memoria es pequeña, muy rápida y posee el mismo tiempo de vida del bloque.

Por su parte, cada hilo en cada bloque de cada grilla puede acceder a la memoria global, que es la más grande, pero también la más lenta. Puede ser leída y escrita por la CPU y por los hilos de la GPU, permitiendo la comunicación de datos entre ellos.

La transferencia de datos entre las memorias cache y compartida es mucho más rápida que entre la memoria global y la memoria compartida. Está en manos del programador

aprovechar las ventajas que proporcionan las distintas memorias, ya que posee el control total sobre ellas, siendo un recurso muy importante a la hora de implementar un algoritmo.

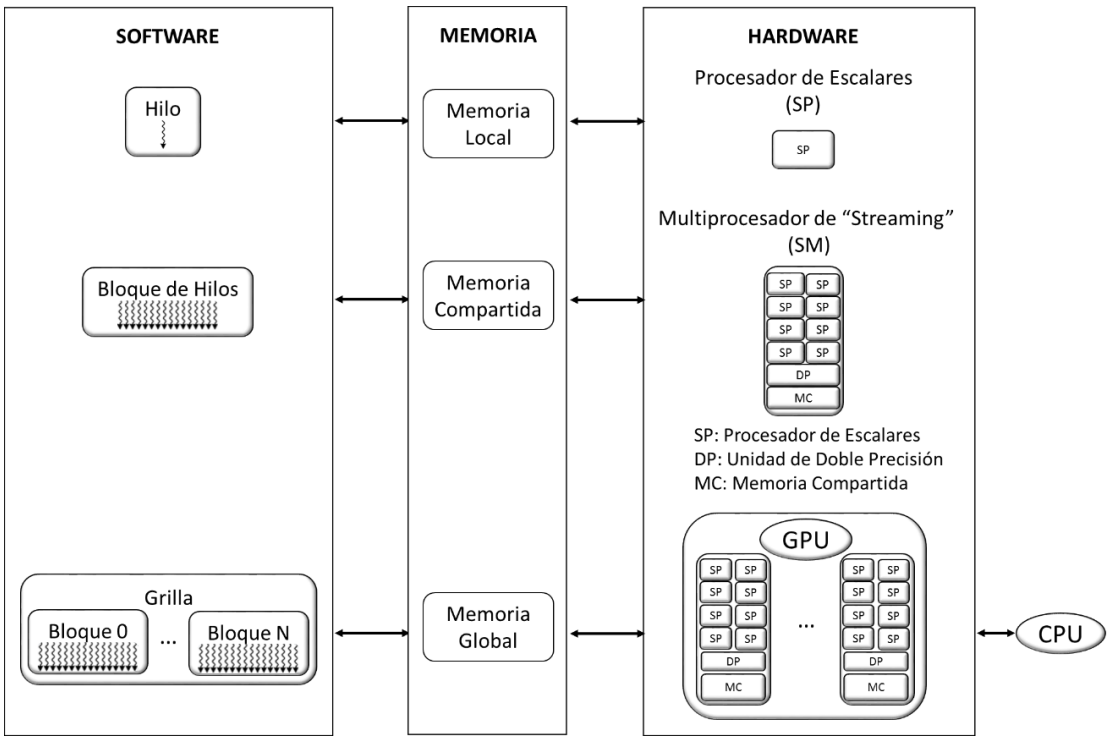

Fig. 3.8: Relación software/hardware/memoria en la GPU

3.4 Metaheurísticas en GPUs/ PSO en GPUs

Actualmente se encuentran muchas metaheurísticas paralelizadas en diferentes arquitecturas. En Alba y col. (2013) se realiza una revisión detallada de la literatura hasta el año 2013. Asimismo, Tan y Ding (2016) examinan varias publicaciones sobre algoritmos de inteligencia de enjambres que se implementaron en GPU, entre las que incluyen la optimización de colonia de hormigas, la evolución diferencial y, por supuesto, el PSO. En su trabajo proponen una taxonomía para clasificar los diversos enfoques, comentan los desafíos de implementación y critican las métricas de desempeño que suelen utilizarse.

A su vez, existen diversas maneras de utilizar los recursos que brinda la GPU, aunque no todas los explotan completamente. En el PSO, el enfoque más utilizado es asignar una partícula a un hilo. En este caso, las partículas se pueden comunicar entre sí a través de la memoria compartida y el número de individuos del enjambre podría ser la cantidad de hilos disponible en la GPU (1024 hilos por bloque). Esta configuración necesita de un lazo para recorrer las variables del problema, no tiene un alto grado de aprovechamiento de los recursos de la placa ni permite un escalado en la arquitectura de la misma.

Otra posibilidad es asignar un bloque a cada partícula y un hilo por dimensión. Esta estructura es posible para implementar un PSO asincrónico, en cambio, en el caso del algoritmo sincrónico este diseño limita su implementación a CUDA 10, ya que es la única versión que permite la sincronización entre bloques (recién a partir de fines de 2018) para que las partículas puedan establecer el mejor global, aunque la otra opción sería implementar un PSO asincrónico en vez del sincrónico. Con esta configuración es posible ejecutar tantas partículas en el enjambre como cantidad de bloques soporte la GPU (80 bloques usualmente), no requiere de lazos y es la que mejor aprovecha los recursos de la placa.

Otra forma es asignar todo el enjambre a un solo bloque y ejecutar un hilo por dimensión, donde cada D dimensiones se representa a cada partícula. De esta manera se pueden comunicar todos los individuos entre sí (tal como en el caso anterior), no se precisa de ningún lazo para recorrer las dimensiones, pero la población del enjambre se encuentra limitada porque la cantidad de partículas por la dimensión del problema no debe superar a la cantidad de hilos de la GPU ($N * D \leq 1024$). Con esta distribución es posible, más adelante, lograr un escalado al uso de multibloques o, incluso, de multiplacas.

Para proporcionar un panorama sobre la programación de metaheurísticas en GPUs, se revisaron los trabajos que abordan específicamente la aceleración de PSO empleando CUDA.

Por ejemplo, Laguna Sanchez y col. (2009), Zhou y Tan (2009) y Kolodziejczyk y col. (2017) implementaron el PSO con distinto grado de paralelismo en la placa, asignando una partícula por hilo. Estos autores testearon el algoritmo con distintas funciones objetivo con restricciones de caja y diversos tamaños de enjambre, resultando que en todos los casos las aceleraciones mejoraban a medida que el número de partículas del enjambre aumentaba, independientemente de la dimensión del problema.

Por su parte, Roberge y Tarbouchi (2012) y Hung y Wang (2012) se enfocaron en realizar una implementación completamente paralelizada del PSO, donde cada paso del algoritmo se optimizará en la GPU, también determinando un hilo por partícula. Al igual que otros autores, aplicaron el PSO a funciones objetivo con restricciones de caja, llegando a la misma conclusión de que la aceleración aumenta a medida que el tamaño del enjambre crece.

Asimismo, Mussi y col. (2011) desarrollaron un PSO sincrónico ("SyncPSO") que se caracteriza por no tener acceso a la memoria global, ya que en esta implementación asignaron todo el enjambre a un solo bloque, donde cada partícula es representada por un hilo al igual que en los trabajos anteriores. De esta manera, cada individuo almacena sus datos dentro de los registros locales del hilo correspondiente y se comunican con los otros a través de la memoria compartida. Esto resulta eficiente desde el punto de vista del uso de la memoria, pero no aprovecha completamente el potencial de la GPU. Para aprovechar mejor los recursos de la placa, estos autores también propusieron un PSO asincrónico ("RingPSO"), en donde cada partícula se asigna a un bloque y cada variable del problema es representada por un hilo, compartiendo información a través de la memoria global. Así, el nuevo modelo explota más de un SM al mismo tiempo, aunque exige más intercambio de información a nivel memoria. En este trabajo, también se aplicaron ambos algoritmos a funciones objetivo sujetas a restricciones de caja, con diferentes tamaños de enjambre, logrando aceleraciones superiores a otras paralelizaciones.

Calazan y col. (2013) proponen tres maneras para paralelizar el PSO: la primera establece una partícula por hilo; la segunda divide en regiones más pequeñas el espacio de búsqueda y asigna un enjambre a cada una ellas, que se ejecutan en paralelo; y la tercera implementa una partícula por bloque de hilos y cada hilo representa la dimensión del problema. Ellos analizaron sólo tres funciones objetivo con restricciones de caja, concluyendo que un aumento en la cantidad de partículas o de dimensiones incrementa el Speed-Up.

Basándose en la revisión anterior, se puede concluir que la mayoría de las implementaciones de PSO paralelizados en GPUs se probaron sólo en modelos conformados por funciones objetivo sujetas a restricciones de caja. Esto es esperable, ya que las funciones objetivo que cuentan únicamente con límites sobre las variables son la opción más sencilla para probar el rendimiento de los algoritmos metaheurísticos. Sin embargo, aunque este tipo de problemas son muy importantes para algunas disciplinas, no reflejan la complejidad de la mayoría de los modelos reales que poseen restricciones de diferentes tipos. De hecho, según nuestro conocimiento, sólo Souza y col. (2012) hace un estudio explícito de modelos con restricciones en una implementación paralela de PSO. Ellos presentaron una versión de PSO hibridizado con mecanismos de estrategias evolutivas, que también explota la cooperación entre múltiples enjambres, siguiendo un modelo de paralelismo maestro-esclavo. Este desarrollo se aplica a varios problemas pequeños del campo de la ingeniería mecánica que poseen restricciones de igualdad y desigualdad, las cuales se tratan como variables de optimización cuyo límite máximo es cero.

Evidentemente, se ha prestado poca atención hasta ahora al manejo sistemático de las restricciones en las implementaciones en GPU de los algoritmos de PSO. La razón tiene que ver probablemente con el hecho de que la programación en GPU es una herramienta relativamente reciente y la mayoría de los esfuerzos de la comunidad científica, hasta el momento, se ha centrado en desarrollar estrategias de paralelismo y realizar estudios de aceleración, más que abordar modelos de interés práctico.

Otra crítica que se puede extraer de la revisión anterior es que la mayoría de los estudios de paralelismo utilizan el tamaño del enjambre para investigar el rendimiento, ya que las aceleraciones aumentan sensiblemente a medida que aumenta este parámetro. Sin embargo, como se ha mencionado anteriormente, la evidencia experimental sugiere que la convergencia del PSO se vuelve insensible a enjambres de más de 100 partículas y que 50 partículas es una cantidad conveniente para la mayoría de los problemas típicos de referencia (Marini y Walzack, 2015). Por lo tanto, el tamaño del enjambre es un parámetro bastante engañoso para investigar el rendimiento. En conclusión, puede decirse que faltan estudios sobre las implementaciones paralelas de PSO con tamaños de enjambre moderados en modelos complejos (con gran cantidad de restricciones y variables).

3.5 Implementación propia

Se implementó una versión paralelizada del PSO utilizando la interfaz del programa de aplicación (API) PyCUDA[4] para explotar el potencial de las GPU de NVIDIA en computadoras de escritorio. Como se mencionó en el Capítulo 1, el uso de Python como lenguaje de programación permite un entorno altamente productivo y proporciona un conjunto sólido de bibliotecas numéricas que facilitan la implementación. De manera similar, el uso de PyCUDA permite el uso de un acelerador sin perder, en teoría, los beneficios de un lenguaje de programación interpretado.

Primeramente, se intentó programar una versión del PSO que asignara una partícula a cada bloque. Esto no se logró debido a que, si bien CUDA permite manejar hilos dentro de un bloque, no permite sincronizar hilos interbloques, lo cual impidió que el enjambre se comunicara para comparar las partículas entre sí y encontrar el mejor global. Actualmente, en la versión de CUDA 10 (2018), NVIDIA incorporó un comando para poder sincronizar los bloques, pero dicha versión es incompatible con PyCUDA por el momento.

[4] www.developer.nvidia.com/pycuda

En términos de recursos, se adoptó el esquema de asignar todo el enjambre del PSO a un solo bloque CUDA, dónde cada variable de cada partícula es manejada por un hilo (Fig. 3.9). Con esta estructura de programación, se hace un uso muy eficiente de la memoria compartida. Sin embargo, como cada bloque solo puede ejecutar un cierto número de hilos (NHB) determinado por el hardware de la placa, esta implementación admite un máximo de partículas (N) en la población que depende de la cantidad de variables (D) del problema analizado. En particular, se debe cumplir con la relación $N * D \leq NHB$.

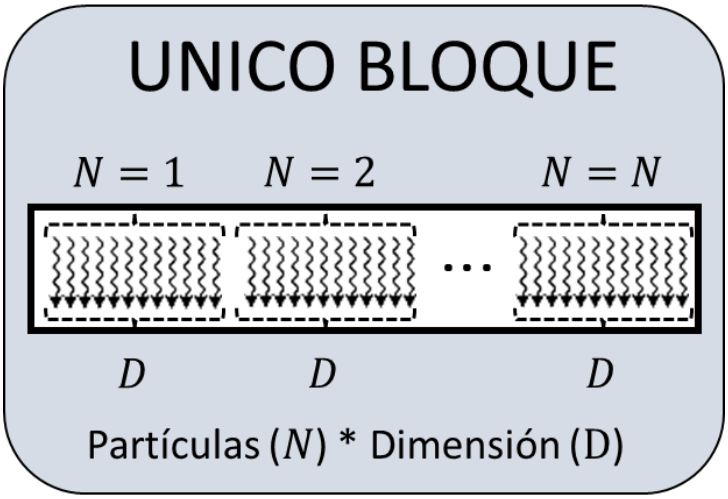

Fig. 3.9: Configuración del bloque en PyCUDA.

Como se mencionó anteriormente, la cantidad de partículas normalmente utilizadas para el PSO es de 50 o menos. Al ser comúnmente 1024 el número de hilos por bloque en una GPU moderna, nuestra implementación permitiría optimizar problemas de hasta 20 variables aproximadamente. La mayoría de los modelos benchmark de la literatura, con los que se testean habitualmente este tipo de algoritmos, poseen dimensiones inferiores a dicho número. Además, si se analizaran modelos con restricciones de igualdad y mayor dimensión, se podría reducir la misma eliminando algunas variables, al despejarlas de las ecuaciones de igualdad, hasta llegar a los 20 grados de libertad requeridos por la estructura implementada. De todas maneras, la estructura adoptada permitiría, en el futuro, escalar a una arquitectura multibloque y/o multiplaca.

En la implementación propuesta los números aleatorios ($r_{1_{ij}}\, y\, r_{2_{ij}}$), las posiciones y velocidades iniciales de las partículas ($\boldsymbol{x^0} y\, \boldsymbol{v^0}$) fueron generadas en la CPU. Por su parte, todos los pasos comprendidos en el paso 2 de la Fig. 2.6, correspondientes al lazo que itera hasta cumplir el criterio de convergencia, se paralelizaron en la GPU. De esta manera y tal como esquematiza la Fig. 3.10, se minimizó la sobrecarga causada por el proceso de intercambio de datos entre la GPU y la CPU. Siguiendo las clasificaciones propuestas por Tan y Ding (2016), la implementación presentada en este trabajo puede ser descrita como un modelo completamente paralelizado en la GPU.

La mayoría de los pasos del pseudocódigo de la Fig. 2.6 admiten una paralelización bastante directa ya que implican multiplicaciones, adiciones y comparaciones de elementos de matrices. La actualización del mejor global, sin embargo, constituye una operación más compleja. Por un lado, requiere que todos los individuos del enjambre se comparen entre sí, y no solo entre la posición actual y la anterior de una dada partícula, como sucede cuando se identifica el mejor local. Asimismo, para verificar que una partícula cumpla con los criterios de factibilidad, es necesario implementar varias comparaciones adicionales consecutivas.

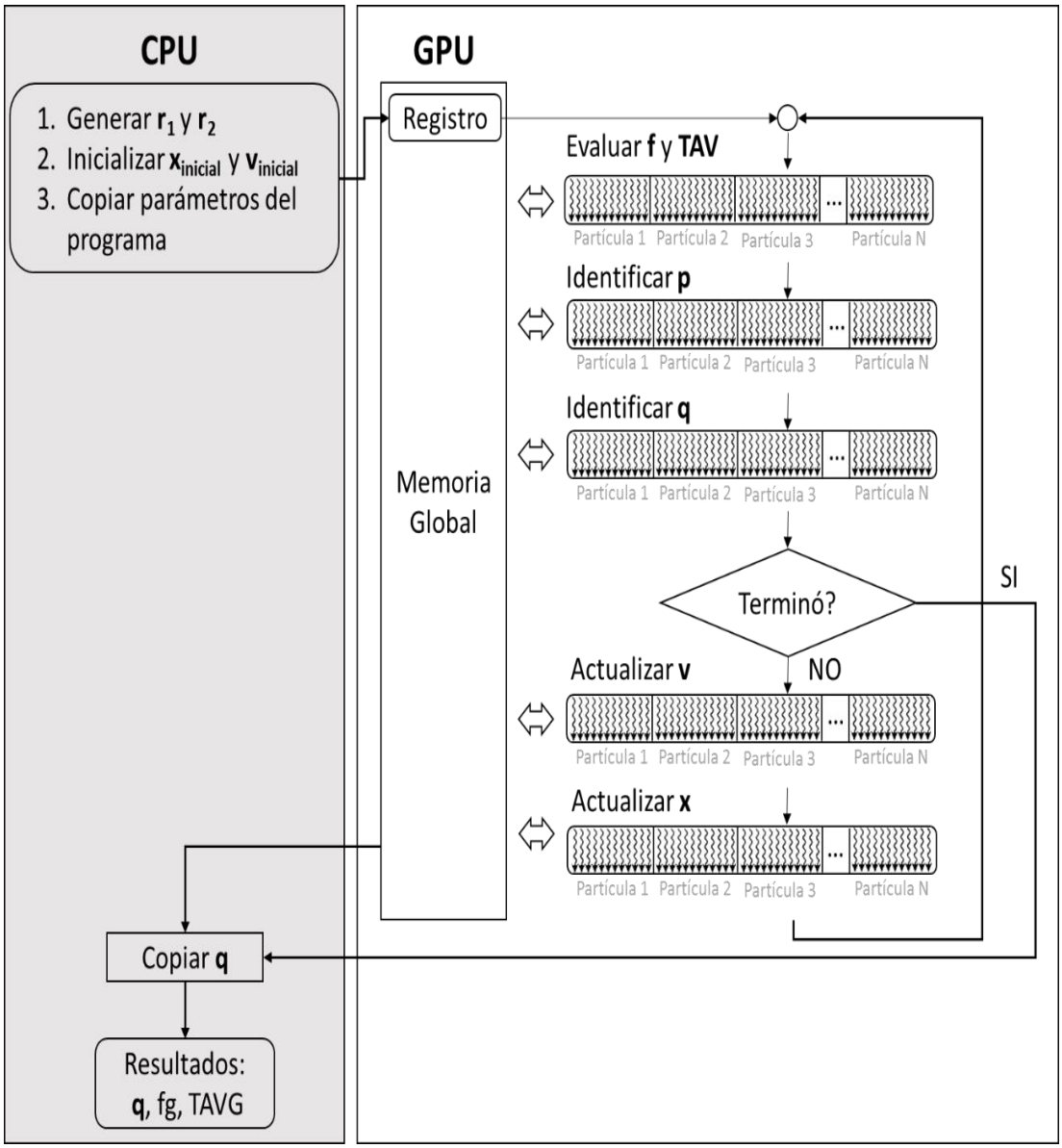

Fig. 3.10: Esquema de implementación de PSO en PyCUDA

Para desarrollar una versión paralela del inciso 2.g de la Fig. 2.6, se programó una reducción paralela basada en árboles para identificar el elemento más pequeño de un vector y el índice correspondiente (Roberge y Tarbouchi, 2012). La misma consiste en que un hilo compare dos elementos: el primer elemento del vector con el elemento que se encuentra en la mitad del vector más uno, conservando el más pequeño. Luego, otro hilo elige el menor valor entre el segundo elemento del vector con el de la mitad más dos. Este proceso se repite sucesivamente hasta que se reduce el vector a la mitad de sus elementos originales. Luego, con el vector resultante se vuelve a aplicar el mismo procedimiento hasta que se identifica el único valor más pequeño y la posición que éste

ocupa en el vector originalmente analizado. Esta técnica se puede visualizar mejor en el ejemplo de la Fig. 3.11, en donde un vector de ocho elementos, al compararlos de a pares, se reduce a cuatro, luego a dos hasta que finalmente se encuentra el mínimo.

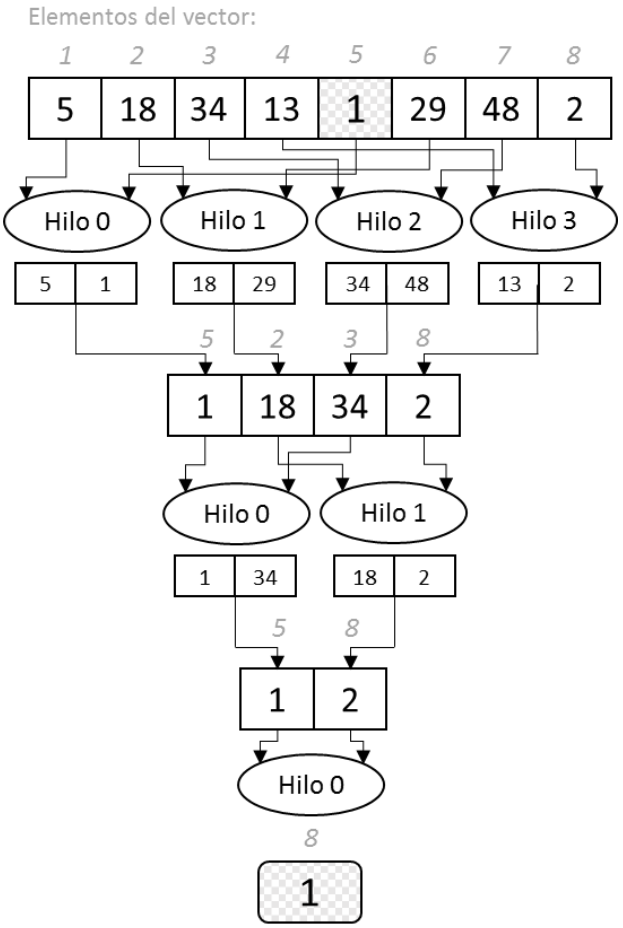

Fig. 3.11: Ejemplo de reducción en serie

En base a esta técnica, se implementó el procedimiento detallado en la Fig. 3.12 para identificar el mejor candidato según los criterios de factibilidad.

a. Evaluar el vector: $[\boldsymbol{TAV^k} - \mu^k]$

b. Aplicar la reducción en paralelo al vector $[\boldsymbol{TAV^k} - \mu^k]$ e identificar el elemento más pequeño.

c. Si el resultado de la reducción es un número positivo o cero:

   i. Asignar la partícula identificada como mejor global.

d. Si el resultado de la reducción es un número negativo:

   i. Asignar infinito al elemento del vector $\boldsymbol{f^k}$ de cada partícula que cumpla la condición $[\boldsymbol{TAV^k} - \mu^k]$ mayor que cero.

   ii. Aplicar la reducción en paralelo al nuevo vector $\boldsymbol{f^k}$ e identificar el elemento más pequeño.

   iii. Asignar la partícula identificada como mejor global.

Fig. 3.12: Reducción en paralelo e identificación del mejor global según los criterios de factibilidad

De esta manera, si todas las partículas resultan infactibles ($\boldsymbol{TAV^k} - \mu^k > 0$), se puede encontrar aquella con menor TAV de todo el enjambre con el paso c. Sin embargo, si alguna/s partículas de la población son factibles ($\boldsymbol{TAV^k} - \mu^k < 0$), con el paso d se identifica aquel individuo que, además de ser factible, tiene la menor función objetivo ($\boldsymbol{f^k}$). Todas las operaciones anteriores admiten paralelización y se implementaron en el kernel PyCUDA.

En lo que respecta al uso de memoria, como se dijo anteriormente, la memoria global es más lenta que la memoria compartida, se optó por transferir a ésta última todos los parámetros y variables de entrada al kernel. De esta forma, se disminuyeron aún más los tiempos de cómputo de la versión paralelizada del algoritmo y, a pesar de ser una memoria pequeña, no limita el buen funcionamiento del PSO.

3.6 Evaluación del algoritmo

Con el fin de cuantificar el grado de aceleración del algoritmo en paralelo respecto de la versión secuencial, es necesario definir la métrica de evaluación. Si bien existe una gran

cantidad de opciones, la más popular es el Speed-Up, y es la que se eligió en este estudio para determinar la eficiencia del algoritmo paralelizado.

El Speed-Up cuantifica el grado de aprovechamiento que el algoritmo paralelo logra de los recursos físicos del sistema, es decir, determina la ganancia de velocidad que se ha obtenido con la ejecución en paralelo. Según la Ec. 3.1, el Speed-Up se define como el cociente entre el tiempo de ejecución de un programa secuencial en la CPU, T_S, y el tiempo de ejecución de la versión paralela de dicho programa en los procesadores disponibles en la GPU, T_P. Dado que puede haber distintas versiones secuenciales, se elige el T_S de la versión secuencial del PSO ejecutada en la CPU con Python. En nuestro caso, T_P corresponde a la versión del PSO implementada en la GPU con PyCUDA.

$$\text{Speed-Up} = T_S/T_P \tag{3.1}$$

Por ejemplo, un Speed-Up igual a 2 indicaría que se ha reducido a la mitad el tiempo al ejecutar el programa con varios procesadores.

El máximo valor que podría llegar a alcanzar el Speed-Up de un algoritmo paralelo será igual a la cantidad de procesadores, p, con los que cuente. Esto se debe a que, si todos ellos tienen la misma potencia de cálculo, a lo sumo lograran que el tiempo de ejecución del programa en paralelo sea p veces inferior a si se ejecutara en un solo procesador. De todas maneras, el tiempo nunca disminuirá en un orden igual a p, ya que es necesario considerar la sobrecarga extra que genera resolver un problema en varios procesadores, debido a sincronizaciones y dependencias entre ellos.

CAPÍTULO 4

TESTEO DEL ALGORITMO

En este capítulo se evalúan cuantitativamente las metodologías descritas en las secciones anteriores. El PSO, en sus versiones serie y paralelizada, se aplica a diversos modelos benchmark de diferente complejidad, cuya solución es conocida y su empleo es habitual en el testeo de algoritmos de optimización. Esto permitió estimar la eficiencia del algoritmo, por medio de la evaluación de métricas relacionadas con la factibilidad y la optimalidad global de las soluciones halladas. Además, en todos los casos se reporta el Speed-Up obtenido para evaluar el desempeño de la versión paralelizada del algoritmo.

Asimismo, en este estudio, se incluyen distintas instancias del problema de optimización de mezclas de crudos de distintas características, con el objetivo de encontrar la combinación más conveniente para cumplir con las especificaciones de calidad del producto. Si bien estos modelos son de pequeña escala, su resolución constituye un gran desafío al ser problemas que poseen bilinealidades, lo que se traduce en la existencia de múltiples óptimos locales con regiones factibles pequeñas y no-convexas.

4.1 Parametrización del PSO y sistemas

Para resolver todos los problemas descritos en esta sección, los experimentos se realizaron usando la siguiente parametrización del PSO: factor de inercia (w) = 0.75, constante de aceleración cognitiva (c_1) = 1.5, constante de aceleración social (c_2) = 1.5, cantidad de partículas (N) = 50, iteraciones máximas ($k_{máx}$) = 10000.

En el caso de la versión paralelizada del algoritmo, se debe establecer también la estructura del bloque y de la grilla proporcionada por la GPU, como se detalló en el capítulo anterior. Se decidió trabajar con un solo bloque de dos dimensiones (número

de partículas y número de variables del modelo) y con una grilla que abarcara todo el bloque. Entonces, se definió el bloque como (N, D, 1) y la grilla como (1, 1, 1).

Los experimentos se llevaron a cabo en dos sistemas diferentes para investigar el rendimiento de la versión paralelizada:

Sistema 1	**Sistema 2**
➢ CPU: Intel Core i7-7700K con 4 núcleos, 4.20GHz y 16 GB de memoria RAM compartida.	➢ CPU: Intel Core i7-2600 con 4 núcleos, 3.40GHz y 16 GB de memoria RAM compartida.
➢ GPU: GeForce GTX TITAN X con 3840 núcleos y 12 GB de RAM dedicada.	➢ GPU: GeForce GTX 1060 con 1150 núcleos y 3 GB de RAM dedicada.

Como PSO contiene componentes que dependen de números aleatorios que causan que la búsqueda de este algoritmo sea estocástica, para cada problema se informan los resultados promediados de 30 ejecuciones independientes. De esta manera, se proporciona cierta validez estadística del rendimiento para las diferentes versiones del algoritmo.

4.2 Problemas benchmark

Con el fin de validar el algoritmo de optimización implementado, se tomaron distintos modelos de distinta complejidad, que se utilizan habitualmente en la literatura para evaluar el rendimiento de los algoritmos metaheurísticos. Dichos problemas son de diferentes tamaños en términos de número de variables, ecuaciones y no linealidades. Algunos poseen múltiples soluciones o cuentan con restricciones de mayor o menor complejidad, entre otras características que provocan que la resolución de cada uno de ellos sea un desafío particular.

En la Tabla 4.1 se detallan las principales características del conjunto de problemas utilizado. Las formulaciones completas de cada modelo se presentan en el Anexo. En la

columna "**Mod**", se proporciona la referencia del problema correspondiente en el Anexo. En las columnas "$\boldsymbol{D}$", "$\boldsymbol{n}$" y "$\boldsymbol{m}$" se informan los números de variables, de restricciones de igualdad y de desigualdad de cada problema, respectivamente, junto con el valor de la función objetivo ($\boldsymbol{f}^*$) y la solución global conocida ($\boldsymbol{x}^*$).

Tabla 4.1. Descripción problemas benchmark

Mod	D	n	m	f^*	x^*
U1	2	0	2D	0.3978	(0, 0)
U2	2	0	2D	-186	(0, 0)
U3	2	0	2D	-1.8013	(2.20, 1.57)
U4	4	0	2D	0	(1, 1, 1, 1)
U5	10	0	2D	0	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
U6	10	0	2D	0	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
U7	10	0	2D	0	(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
U8	10	0	2D	0	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
U9	10	0	2D	0	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
U10	2	0	2D	0	(-10, 1)
U11	10	0	2D	0	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
U12	10	0	2D	0	(-0.5,-0.5,-0.5,-0.5,-0.5,-0.5,-0.5,-0.5,-0.5,-0.5)
C1	13	0	9+2D	-15	(1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 1)
C2	10	1	2D	-1.0005	(0.31, 0.31, 0.31, 0.31, 0.31, 0.31, 0.31, 0.31, 0.31, 0.31)
C3	5	0	6+2D	-30665.5	(78, 33, 29.99, 45, 36.77)
C4	4	3	2+2D	5126.49	(679.94, 1026.06, 0.1188, -0.3962)
C5	2	0	2+2D	-6961.81	(14.09, 0.84)
C6	10	0	8+2D	24.30	(2.17,2.36,8.77,5.09, 0.99, 1.43, 1.32, 9.82, 8.28, 8.37)
C7	2	0	2+2D	-0.0958	(1.22,4.24)
C8	7	0	4+2D	680.63	(2.33, 1.91, -0.47, 4.36, -0.62, 1.03, 1.59)
C9	8	0	6+2D	7049,24	(579.30, 1359.9, 5109.9, 182, 295.60, 217.98, 286.41, 395.6)
C10	2	1	2D	0.7499	(-0.70, 0.50)
C11	5	3	2D	0.0539	(1.71, 1.59, 1.82, -0.76, -0.76)
C12	10	3	2D	-47.76	(0.04, 0.14, 0.78, 0, 0.48, 0, 0.02, 0.01, 0.03, 0.09)
C13	3	2	2D	961.71	(3.51, 0.21)
C14	9	0	13+2D	-0,866	(-0.6, -0.15, 0.32, -0.94, -0.65, -0.75, 0.32, -0.34, 0.59)
C15	2	0	2+2D	-5,508	(2.32, 3,17)

Los modelos U1 a U12, tomados de Chen y Chi (2010), son funciones objetivo que cuentan solo con restricciones del tipo “caja” conformada por los límites superiores e inferiores que tiene cada variable ($m = 2D$). Por otro lado, los problemas C1 a C15, seleccionados de Liang y col. (2006), poseen además distinto número de restricciones de igualdad y desigualdad.

4.2.1 Resultados

En la Tabla 4.2, se detallan los resultados obtenidos al ejecutar las versiones en serie y en paralelo del PSO en los dos sistemas utilizados y según la parametrización enunciada en la sección 4.1.

La columna "T_{CPU}" representa el tiempo promedio de 30 ejecuciones independientes del PSO en serie. La columna "Speed-Up" reporta la ganancia de velocidad promedio también obtenida de 30 ejecuciones independientes, utilizando el algoritmo paralelizado en la GPU en el sistema correspondiente.

La columna "Tasa de factibilidad" informa el porcentaje de las ejecuciones que lograron soluciones factibles dentro de una tolerancia ($TAV < 10^{-4}$). La columna "Tasa de optimalidad" indica el porcentaje de ejecuciones factibles que lograron la solución óptima global con una tolerancia inferior a 10^{-1} en el Error Relativo que se calcula según la Ec. 4.1:

$$Error\ Relativo = \begin{cases} \frac{|f - f^*|}{f^*} & Si\ f^* \neq 0 \\ |f - f^*|\ Si\ \ f^* = 0 \end{cases} \tag{4.1}$$

Donde f^* es el valor de la función objetivo conocida del problema y f es el valor de la función obtenido por el algoritmo). Cabe aclarar que, en todos los casos, la tasa de optimalidad se calcula sobre aquellas soluciones que resultan factibles, es decir, primero

se comprueba si resultado alcanzado por el PSO cumple con el criterio de factibilidad y, sobre las soluciones que lo verifican, se determina la tasa de optimalidad.

Con respecto a la tasa de factibilidad, el 92.59% de los problemas alcanzan soluciones factibles en el 100% de las corridas. El modelo C4 sólo en dos corridas no llega a la tolerancia de factibilidad impuesta, mientras que C13 no lo hace en cuatro de las ejecuciones. Ambos casos incluyen restricciones de igualdad lo que determina una región factible muy reducida.

La tasa de optimalidad fue superior al 70% en el 83% de los problemas con límites sobre las variables (serie U) y en el 73% de los problemas con restricciones (serie C). En los casos restantes, la tasa de optimalidad fue superior al 23%.

Los resultados de optimizar los distintos problemas benchmark sugieren que el PSO implementado logra un buen rendimiento general, tanto en problemas sin restricciones como en aquellos que si las poseen. En la gran mayoría de los casos, el algoritmo converge a soluciones factibles, a pesar de que en algunas ejecuciones sólo alcance soluciones subóptimas. Es importante destacar que en todos los problemas reportados fue posible encontrar el óptimo global dentro de la tolerancia definida en algún porcentaje de los experimentos.

Desde el punto de vista de la aceleración lograda por uso de la GPU, puede observarse de la Tabla 4.2, que se obtuvieron importantes aceleraciones promediando 116x para el Sistema 1 y 299x para el Sistema 2. Si bien se esperaría obtener el Speed-Up más alto con el Sistema 1, ya que se compone de una GPU con mayor capacidad de procesamiento, esto no sucedió debido a que también es necesario tener en cuenta los recursos de la CPU involucrada, ya que son distintas para cada sistema. Dado que el Speed-Up se definió como el cociente entre el tiempo de ejecución de la versión serie y paralelo del PSO, y considerando que el Sistema 2 posee una CPU de menor generación que provoca que el tiempo serie sea mucho mayor que en el primer sistema, es razonable que esta métrica resulte mejor en el segundo sistema.

Tabla 4.2. Resultados problemas benchmark

Mod	Tasa de Factibilidad	Tasa de Optimalidad	Sistema 1		Sistema 2	
			T_{CPU} (seg)	Speed-Up	T_{CPU} (seg)	Speed-Up
U1	100	100	15,88	77,74	34,35	217,36
U2	100	100	29,53	118,26	68,01	342,03
U3	100	100	18	66,72	39,91	197,58
U4	100	100	29,78	108,73	64,66	286,45
U5	100	100	59,17	110,38	120,84	268,09
U6	100	100	58,48	135,6	119,6	320,42
U7	100	80	82,08	115,51	169,15	299,98
U8	100	36,67	83,58	132,64	172,87	333,81
U9	100	30	68,24	63,88	151,06	190,08
U10	100	83,33	15,17	80,14	30,42	208,86
U11	100	100	57,98	112,19	115,11	269,46
U12	100	73,33	62,86	100,87	120,6	238,26
C1	100	76,67	66,62	144,65	135,74	333,51
C2	100	36,67	62,91	115,65	132,94	294,71
C3	100	100	54,27	168,96	113,11	453,57
C4	93,33	93,33	34,76	72,87	74,16	224,6
C5	100	100	19,19	89,57	40,04	243,12
C6	100	96,67	109,33	182,01	223,74	447,27
C7	100	100	18,96	71,6	41	206,39
C8	100	100	71,83	180,17	153,55	475,03
C9	100	23,33	53,58	129,73	106,28	321,65
C10	100	100	15,03	76,88	31,83	214,4
C11	100	23,33	32,09	136,14	66,35	321,84
C12	100	43,33	97,75	154,08	196,22	373,04
C13	86,67	86,67	28,25	116,88	59,45	313,73
C14	100	86,67	78,65	165,82	161,59	395,97
C15	100	100	22,46	111,52	46,48	304,13

Cabe mencionar que, como el objetivo principal de este estudio fue disminuir el tiempo de cómputo del algoritmo en serie a través de la versión paralelizada del mismo, el PSO no se ajustó específicamente para cada uno de los diferentes problemas estudiados y solamente se adoptó una parametrización estándar en todos los casos, tal como se describió anteriormente. Por esta razón, la tasa de optimalidad podría no llegar a parecer tan impresionante como en otros estudios donde el énfasis está en la convergencia al óptimo global.

4.3 Problemas de mezclado de petróleo

Un problema muy común en la industria del petróleo es el mezclado de crudos provenientes de distintos yacimientos, y por ende de diferentes características. El objetivo de la mezcla es cumplir con ciertas especificaciones de calidad. Si bien existen muchas maneras de caracterizar un crudo, una de las más empleadas es aquella que utiliza la proporción de azufre como parámetro de calidad. Físicamente, el mezclado se produce en piletas, donde convergen las distintas corrientes de crudos. A su vez, las salidas de las piletas se combinan en los tanques de productos con las especificaciones requeridas.

La optimización del mezclado suele mejorar considerablemente el rendimiento económico de la actividad por ahorros en los costos y mayores márgenes de ganancia. Sin embargo, los modelos de optimización del proceso de mezclado poseen no linealidades y no convexidades que conducen a la existencia de múltiples soluciones óptimas locales y regiones factibles pequeñas, tornando desafiante su resolución computacional eficiente.

Existen varios modelos que describen diversos tipos de problemas de mezclado de distinta dificultad. En esta sección se aborda un sistema particularmente bien conocido, el de la familia de modelos Haverly (Adhya y col., 1999), cuya estructura se representa en la Fig. 4.1.

Estos modelos consisten en tres fuentes del petróleo crudo de diversos costos y contenidos de azufre. Dos de estas fuentes (f_{11} y f_{21}) se mezclan en una pileta. Las corrientes x_{11} y x_{12} se puede mezclar con la tercera fuente (f_{12}) para alcanzar los requisitos de los dos productos finales (X e Y). Cada uno de los productos tiene un contenido de azufre máximo, una cota máxima de producción establecida por la demanda (s_1 y s_2) y un cierto precio.

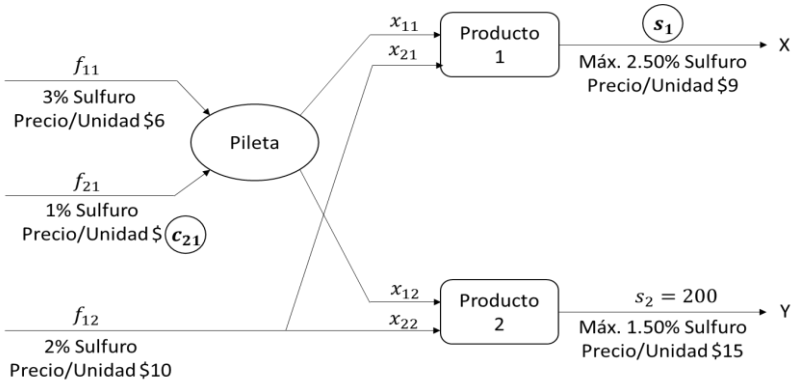

Fig. 4.1: Flowsheet de los modelos de Haverly

La formulación correspondiente es (Ec. 4.2):

$$Min\ 6\ f_{11} + c_{21}\ f_{21} + 10\ f_{12} - 9\ (x_{11} + x_{21}) - 15\ (x_{12} + x_{22})$$

Sujeto a:

$$f_{11} + f_{21} - x_{11} - x_{12} = 0$$

$$f_{12} - x_{21} - x_{22} = 0$$

$$q(x_{11} + x_{12}) - 3f_{11} - f_{21} = 0$$

$$qx_{11} + 2x_{21} - 2.5(x_{11} + x_{21}) \leq 0$$

$$qx_{12} + 2x_{22} - 1.5(x_{12} + x_{22}) \leq 0$$

$$x_{11} + x_{21} \leq s_1$$

$$x_{12} + x_{22} \leq 200$$

$$x_{lo,j} - x_j \leq 0 \ \textit{(j=1…D)}$$

$$x_j - x_{up,j} \leq 0 \ \textit{(j=1…D)} \qquad (4.2)$$

Las distintas versiones difieren básicamente en algunos de los parámetros del modelo. La Tabla 4.3 detalla los parámetros específicos de cada uno de los problemas analizados en este trabajo (Haverly1, Haverly2 y Haverly3) y los valores de la función objetivo y de las variables correspondientes a las mejores soluciones conocidas.

Tabla 4.3. Parámetros y soluciones de los problemas de mezclado

Parámetros y variables	**Haverly 1**	**Haverly 2**	**Haverly 3**
c_{21}	16	16	13
s_1	100	600	100
x^{lo}	(0, 0, 0, 0, 0, 0, 0, 1)	Idem Haverly 1	Idem Haverly 1
x^{up}	(300, 300, 300, 300, 300, 300, 300, 4)	(600, 600, 600, 600, 600, 600, 600, 4)	Idem Haverly 1
FO^*	-400	-600	-750
f_{11}^*	0	300	50
f_{12}^*	100	0	0
f_{21}^*	100	300	150
x_{11}^*	0	300	0
x_{12}^*	100	0	200
x_{21}^*	0	300	0
x_{22}^*	100	0	0
q^*	1	3	1.5
Tasa de Factibilidad (*)	100	100	100
Tasa de Optimalidad (*)	56	18	63

*Fuente: Adhya y col. (1999)

Con fines comparativos, para presentar algunos resultados alcanzados por otras técnicas de optimización, también se indica en la Tabla 4.3, las tasas de factibilidad y optimalidad informadas en Adhya y col. (1999) obtenidas de 100 búsquedas locales a partir de puntos

iniciales aleatorios con el código de programación no lineal comercial MINOS (Murtagh y Saunders, 1995). Como puede observarse, las tasas de optimalidad global no son muy altas. Esto sugiere que, aun con una estrategia de múltiples puntos de partida y un solver como MINOS, encontrar el óptimo para este tipo de problemas no resulta sencillo.

4.3.1 Algoritmo con reducción secuencial de caja

Puesto que los términos bilineales en la formulación del problema producen espacios de búsqueda no convexos y pequeños, el óptimo global es generalmente esquivo. En los primeros experimentos realizados con el PSO original se encontraron tasas de factibilidad y optimalidad relativamente bajas, las cuales se consideraron insatisfactorias. Con el fin de mejorar estos indicadores, la implementación de PSO original se extendió para permitir varias búsquedas consecutivas en regiones progresivamente más pequeñas.

En la Fig. 4.2, se esquematiza el procedimiento propuesto. El optimizador comienza con una población de individuos aleatorios delimitados por los límites superiores e inferiores originales de las variables del modelo y procede durante un cierto número de iteraciones ($k_{máx}$). Luego, se inicia una segunda búsqueda durante otras $k_{máx}$ iteraciones a partir de una población generada nuevamente de manera aleatoria, pero esta vez alrededor de la mejor solución de la búsqueda anterior (q). Para ello, se define una nueva caja más pequeña alrededor de la solución anterior, tal que: $(1-\beta)q \leq x \leq (1+\beta)q$, donde β es un parámetro definido por el usuario para controlar el tamaño de la nueva caja. Este procedimiento se repite $K_{máx}$ veces.

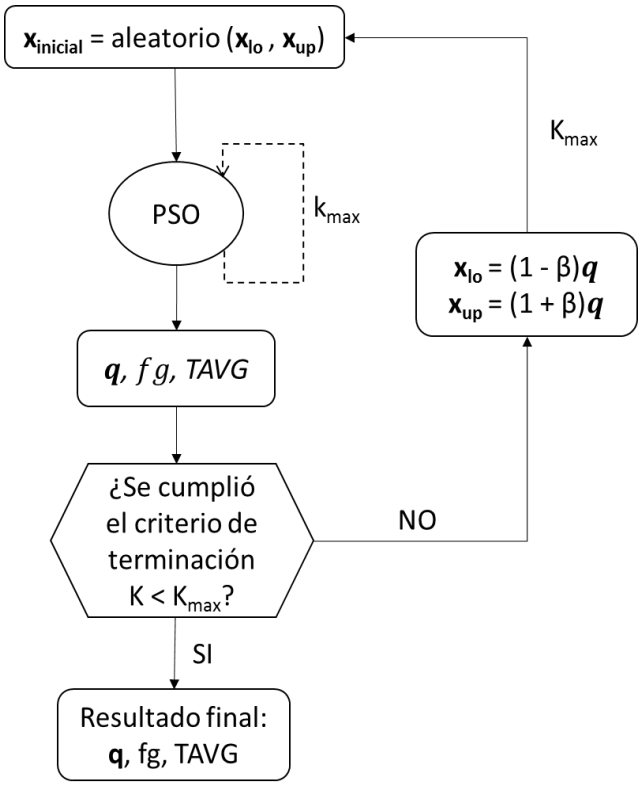

Fig. 4.2: Esquema de representación de la implementación del PSO con reducción secuencial de caja

4.3.1.1 Resultados algoritmo con reducción secuencial de caja

A los efectos de esta investigación, se adoptaron los siguientes parámetros para el algoritmo con reducción secuencial de caja: β = 0.5, $k_{máx}$ = 1000, $K_{máx}$ = 10. Los parámetros del PSO se mantuvieron sin cambios. La Tabla 4.4 muestra los resultados obtenidos:

Tabla 4.4. Resultados de los problemas de mezclado con PSO con reducción secuencial de caja

Modelo	Tasa de Factibilidad	Tasa de Optimalidad	Sistema 1		Sistema 2	
			T_{CPU} (seg)	Speed-Up	T_{CPU} (seg)	Speed-Up
Haverly 1	100	23.33	69.8	193.74	69.72	214.26
Haverly 2	80	0	72.63	208.25	72.74	228.31
Haverly 3	100	56.67	70.55	197.12	70.62	218.70

Asimismo, se puede observar que, si bien el algoritmo siempre convergió a soluciones factibles en el Haverly 1 y 3, solo lo hizo en el 80% de los casos en el Haverly 2. Con respecto a la optimalidad, las soluciones óptimas globales fueron encontradas para el problema Haverly 1 un 23,33% de los experimentos y para el Haverly 3 en el 56,67% de las ejecuciones.

Para Haverly 2 el óptimo global real nunca pudo ser identificado dentro de la tolerancia adoptada. El promedio de las 30 corridas indica que se llega a un óptimo local de -354, cuando el global es de -600 (error relativo de 0.41). Este modelo es uno de los más complejos de resolver, dentro de los propuestos por Haverly, tal como se observa también en la Tabla 4.3 para la estrategia de búsqueda local con múltiples puntos de partida. Esto se debe a que este caso cuenta con un espacio de búsqueda mucho más extenso que los otros dos, lo que también explicaría la dificultad en encontrar soluciones factibles en todas las ejecuciones del algoritmo.

En comparación con el rendimiento de la estrategia de búsqueda local de inicio múltiple que utiliza un solver determinístico comercial, nuestros resultados son inferiores, en particular para Haverly 2. Sin embargo, estas soluciones podrían mejorar con parametrizaciones ad hoc del algoritmo. Asimismo, desde una perspectiva ingenieril la identificación de soluciones subóptimas factibles puede ser de interés práctico para estos problemas difíciles. Dado que tal condición se logró en una gran proporción de las ejecuciones, el rendimiento general del algoritmo se considera satisfactorio en esta etapa de desarrollo.

En lo que respecta a las aceleraciones, éstas alcanzaron valores muy significativos, ya que excedieron 193x y 214x en los sistemas 1 y 2 respectivamente para los tres problemas. Entonces, aun considerando que la implementación del PSO con reducción secuencial de caja demanda mayor tiempo de cómputo que el algoritmo original, sigue resultando conveniente implementarla, porque aun así alcanza Speed-Ups importantes mientras mejora la eficiencia del algoritmo.

CAPÍTULO 5

APLICACIONES DEL ALGORITMO

En este capítulo se presentan problemas más desafiantes que los abordados en el capítulo anterior por poseer mayor número de variables y restricciones. En particular, se consideraron cuatro aplicaciones de la ingeniería química tomadas de literatura (Andrei, 2013): (i) diseño óptimo de un sistema de refrigeración industrial; (ii) diseño óptimo de una red de intercambiadores de calor en paralelo; (iii) separación de propano, isobutano y n-butano en dos columnas de destilación; y (iv) un problema de mezclado con 5 materias primas, 5 productos y 3 piletas. Todos estos modelos han sido empleados en numerosos estudios de optimización por parte de la comunidad de ingeniería de sistemas y procesos en los últimos años.

5.1 Reformulación de los problemas en el espacio reducido

Dado que los problemas estudiados en este capítulo son más complejos que los analizados anteriormente al tener una mayor dimensión, se espera que el algoritmo base no presente un buen desempeño en términos de las métricas adoptadas. Por esto se decidió trabajarlos en el espacio reducido, es decir, reformularlos de manera de eliminar variables y restricciones de igualdad y de ese modo reducir el número de variables de optimización.

Para ello, primero es conveniente precisar el concepto de los grados de libertad de un problema (GL). Los mismos se determinan como la cantidad de variables (D) menos la cantidad de ecuaciones de igualdad (n) que posee el sistema (Ec. 5.1).

$$GL = D - n \tag{5.1}$$

Dichos grados de libertad representan el número de variables cuyos valores pueden ser asignados de forma arbitraria. Una vez que éstos quedan establecidos, el sistema queda definido, es decir, la cantidad de ecuaciones del problema es igual al número de

variables. Entonces, resolviendo dicho sistema de ecuaciones de igualdad, es posible encontrar el valor del resto de las variables. De esta manera, se minimiza la complejidad del problema por eliminación de una cantidad de restricciones de igualdad, las cuales, habitualmente, son difíciles de verificar por los algoritmos de optimización en general y por los estocásticos en particular. Esta estrategia de reducción es ampliamente utilizada en los algoritmos de optimización y es una de las principales fortalezas de los solvers comerciales como CONOPT, que la implementan muy eficientemente en la etapa de preprocesamiento del modelo.

Una dificultad surge al momento de decidir qué variables se especificarán, y cuáles se obtendrán a partir de la resolución del sistema de ecuaciones. Según cada elección, quedarán determinados distintos sistemas de ecuaciones que, aun representando el mismo problema, puede provocar que la resolución sea más o menos sencilla. Por ejemplo, Everyoglu y Bilge (2016) proponen que uno de los sets de variables que causan la bilinealidad en el modelo sean variables de búsqueda y las manipule el algoritmo de optimización adoptado. Así, el resto de las variables, se puede calcular simplemente resolviendo el problema lineal resultante.

Existen incluso algunas herramientas para ayudar a los desarrolladores a establecer más o menos automáticamente el mejor sistema de ecuaciones para trabajar en el espacio reducido. Por ejemplo, Bunus y Fritzson (2002) crearon un marco de depuración basado en distintas técnicas y algoritmos que permiten encontrar la mejor manera de representar el mismo problema.

En particular, en esta tesis se analizó la cantidad de ecuaciones en las que aparecía cada variable y se definieron como variables manipuladas ($\boldsymbol{x_M}$) aquellas que intervenían en la mayor cantidad de ecuaciones. Las variables calculadas ($\boldsymbol{x_C}$) se obtuvieron resolviendo el sistema resultante por despeje directo de las ecuaciones de igualdad, en el caso que las variables se pudieran explicitar (Ec. 5.2). El nuevo sistema a resolver no sólo tiene una menor cantidad de variables a optimizar, como se pretendía, sino que también está sujeto a una menor cantidad de restricciones de igualdad.

$$x_{C1} = h_1^{'}(\boldsymbol{x_M})$$
$$x_{C2} = h_2^{'}(\boldsymbol{x_M}, x_{C1})$$
$$x_{C3} = h_3^{'}(\boldsymbol{x_M}, x_{C1}, x_{C2})$$
$$\vdots$$
$$x_{Ci} = h_i^{'}(\boldsymbol{x_M}, x_{C1}, x_{C2}, \dots, x_{Ci}, \dots, x_{Ci-1}) \qquad (5.2)$$

Cabe destacar que, en algunos casos, no se logró reducir completamente la dimensión del problema a los grados de libertad del mismo. Esto sucedió en aquellos modelos donde no fue posible explicitar cada variable a partir de alguna de las ecuaciones, una vez que ya se fijaron las variables manipuladas. En estas situaciones, las restricciones de igualdad que no pudieron ser eliminadas fueron tratadas con la técnica TAV descripta anteriormente. En la Fig. 5.1 se muestra esquemáticamente el proceso de optimización en espacio reducido.

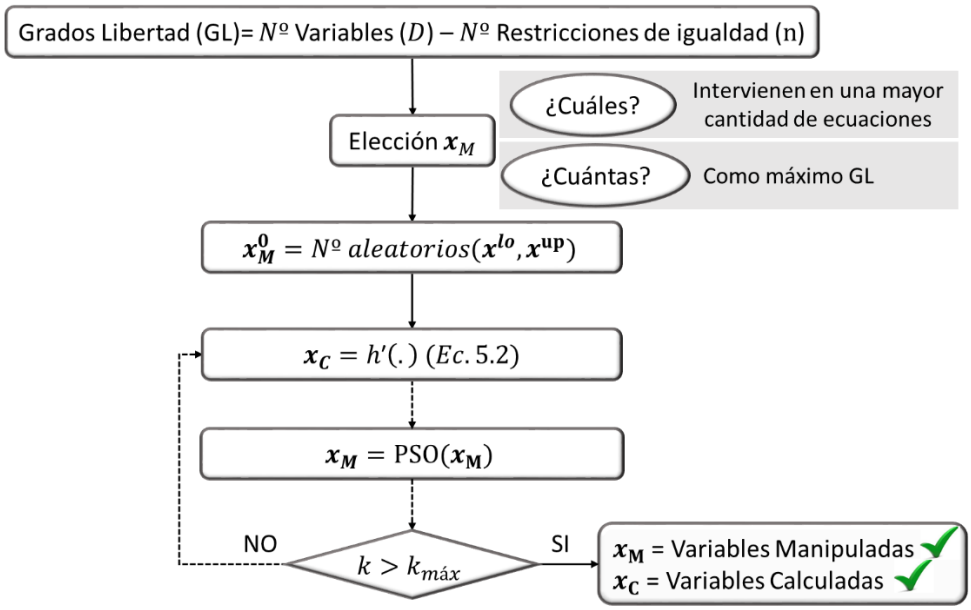

Fig. 5.1 Proceso de optimización en espacio reducido

5.2 Aplicaciones

En esta sección se presenta cada problema estudiado, se detalla su expresión matemática y se reporta la solución conocida. Asimismo, para algunos de los modelos, se especificará también la reformulación en el espacio reducido adoptada. Todos los

casos de estudio fueron extraídos de Andrei (2013), donde se presenta una compilación de aplicaciones de optimización no lineal de interés a diversas ramas de la ingeniería formuladas en la plataforma GAMS.

5.2.1 Diseño óptimo de un sistema de refrigeración industrial

Los sistemas de refrigeración se utilizan para disminuir la temperatura de un fluido por intercambio calorífico con un refrigerante. Este proceso se realiza en una amplia variedad de industrias tales como refinerías y plantas químicas de todo tipo. El correcto diseño y funcionamiento de los distintos sistemas de enfriamiento es fundamental debido a que muchos productos y procesos requieren estabilidad y consistencia en la temperatura, factor que puede ser indispensable para asegurar la calidad del producto. Asimismo, se pretende que este tipo de proceso consuma poca energía con el objetivo de ahorrar recursos económicos y proteger el medio ambiente.

Debido a su importancia, estos procesos han sido intensamente estudiados en la literatura de optimización. En este trabajo se estudia el sistema de refrigeración industrial descrito en Paul (1987) y en Pant y col. (2009 a, b). A continuación, se presenta la formulación (Ec. 5.3) reportada en Andrei (2013) (Aplicación 7.3).

Minimizar:

$$\begin{aligned} f(\mathbf{x}) = {} & 63098.88x_2x_4x_{12} + 5441.5x_2^2x_{12} + 115055.5x_2^{1.664}x_6 \\ & +6172.27x_2^2x_6 + 63098.88x_1x_3x_{11} + 5441.5x_1^2x_{11} \\ & +115055.5x_5x_1^{1.664} + 6172.27x_1^2x_5 + 140.53x_1x_{11} \\ & +281.29x_3x_{11} + 70.26x_1^2 + 281.29x_1x_3 + 281.20x_3^2 \\ & +14437x_1^2x_7x_8^{1.8812}x_9^{-1}x_{10}x_{12}^{0.3424}x_{14}^{-1} + 20470.2x_1^2x_7^{2.893}x_{11}^{0.316} \end{aligned}$$

Sujeto a:

$$\begin{aligned} g_1(\mathbf{x}) &= 1.524x_7^{-1} - 1 \leq 0 \\ g_2(\mathbf{x}) &= 1.524x_8^{-1} - 1 \leq 0 \\ g_3(\mathbf{x}) &= 0.07789x_1 - 2x_7^{-1}x_9 - 1 \leq 0 \\ g_4(\mathbf{x}) &= 7.05305x_1^2x_2^{-1}x_8^{-1}x_9^{-1}x_{10}x_{14}^{-1} - 1 \leq 0 \\ g_5(\mathbf{x}) &= 0.0833x_{13}^{-1}x_{14} - 1 \leq 0 \end{aligned}$$

$$g_6(\mathbf{x}) = 47.136x_2^{0.33}x_{10}^{-1}x_{12} + 62.08x_8^{0.2}x_{10}^{-1}x_{12}^{-1}x_{13}^{2.1195}$$
$$-1.333x_8x_{13}^{2.1195} - 1 \leq 0$$
$$g_7(\mathbf{x}) = 0.0477x_8^{1.8812}x_{10}x_{12}^{0.3424} - 1 \leq 0$$
$$g_8(\mathbf{x}) = 0.0488x_7^{1.893}x_9x_{11}^{0.316} - 1 \leq 0$$
$$g_9(\mathbf{x}) = 0.0099x_1x_3^{-1} - 1 \leq 0$$
$$g_{10}(\mathbf{x}) = 0.0193x_2x_4^{-1} - 1 \leq 0$$
$$g_{11}(\mathbf{x}) = 0.0298x_1x_5^{-1} - 1 \leq 0$$
$$g_{12}(\mathbf{x}) = 0.056x_2x_6^{-1} - 1 \leq 0$$
$$g_{13}(\mathbf{x}) = 2x_9^{-1} - 1 \leq 0$$
$$g_{14}(\mathbf{x}) = 2x_{10}^{-1} - 1 \leq 0$$
$$g_{15}(\mathbf{x}) = x_{11}^{-1}x_{12} - 1 \leq 0 \qquad (5.3)$$

$Límites: 0.001 \leq x_i \leq 5, i = 1, \ldots .14.$

$Mínimo\ global: f(\mathbf{x}^*) = 0.032213$

$\mathbf{x}^* = (x_1 = 0.001, x_2 = 0.001, x_3 = 0.001, x_4 = 0.001, x_5 = 0.001, x_6 = 0.001,$
$x_7 = 1.524, x_8 = 1.524, x_9 = 5, x_{10} = 2, x_{11} = 0.001, x_{12} = 0.001,$
$x_{13} = 0.0072934, x_{14} = 0.0875558)$

Este modelo posee 14 variables y 15 restricciones de desigualdad no lineales. Si bien su dimensión no es mucho mayor que algunos de los problemas benchmark analizados en el Capítulo 4, sus restricciones no lineales son de una mayor complejidad.

5.2.2 Diseño óptimo de una red de intercambiadores de calor en paralelo

La mayoría de los procesos industriales involucran la transferencia de calor, ya sea de una corriente de proceso a otra o de una corriente de servicio auxiliar a una corriente de proceso. Los intercambiadores de calor son dispositivos cuya función es transferir el calor de un fluido a otro de menor temperatura.

Actualmente, en cualquier diseño de proceso industrial, se busca la maximización de la recuperación de calor y la minimización de uso de servicios auxiliares. Para lograr el objetivo de recuperación máxima de energía o de consumo mínimo, se requiere de una

red de intercambio de calor apropiada. El diseño de una red de esta naturaleza no es fácil si se considera el hecho de que la mayoría de los procesos involucran un gran número de corrientes y de servicios auxiliares. En este caso en particular, se requiere que una corriente de agua fría sea calentada desde $T_{C,i} = 150°\text{F}$ hasta $T_{C,o} = 310°\text{F}$ utilizando dos intercambiadores de calor en paralelo.

Tal como muestra la Fig 5.2, de cada intercambiador de calor, se conoce: el flujo de entrada ($f_{H1,2}, f_{H2,2}$), la temperatura de entrada ($T_{H1,i}, T_{H2,i}$) y la de salida ($T_{H1,o}, T_{H2,o}$) del agua caliente.

Las variables del problema, para cada equipo son: $\Delta T_{1,1}$, $\Delta T_{1,2}$, $\Delta T_{2,1}$ y $\Delta T_{2,2}$ las cuales representan la diferencia de temperatura (entre el agua caliente y el agua fría); $T_{1,i}$, $T_{2,i}$ son las temperaturas de entrada de cada equipo y $T_{1,o}$, $T_{2,o}$ las de salida; $f_{1,1}, f_{1,2}, f_{1,3}, f_{1,4}, f_{2,1}, f_{2,2}, f_{2,3}, f_{2,4}$ determinan el flujo de agua del proceso en la entrada, en la salida y en las recirculaciones.

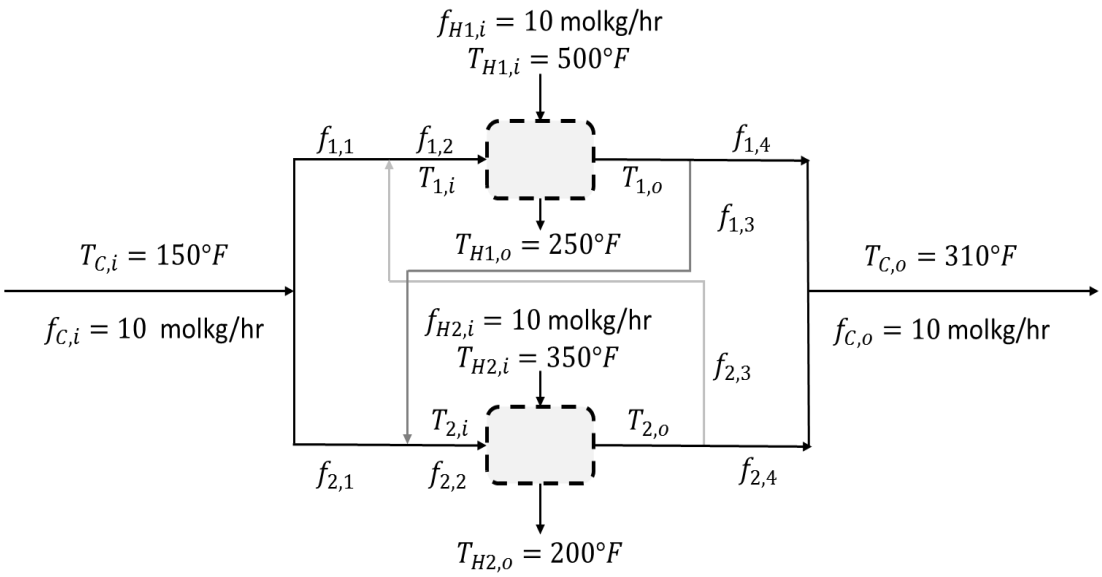

Fig. 5.2 Red de intercambiadores de calor en paralelo

El modelo matemático original para esta aplicación fue presentado en Floudas y col. (1999) y Visweswaran y Floudas (1996). La formulación que se detalla a continuación (Ec. 5.4) se tomó de Andrei (2013) (Aplicación 7.14):

Minimizar:

$$f(\mathbf{x}) = 1300\left(\frac{1000}{\frac{(\Delta T_{11}\Delta T_{12})}{30}+\frac{(\Delta T_{11}\Delta T_{12})}{6}}\right)^{0.6} + 1300\left(\frac{600}{\frac{(\Delta T_{21}\Delta T_{22})}{30}+\frac{(\Delta T_{21}\Delta T_{22})}{6}}\right)^{0.6}$$

Sujeto a:

$$\begin{aligned}
h_1(\mathbf{x}) &= f_{1,1} + f_{2,1} - 10 = 0\\
h_2(\mathbf{x}) &= f_{1,1} + f_{2,3} - f_{1,2} = 0\\
h_3(\mathbf{x}) &= f_{2,1} + f_{1,3} - f_{2,2} = 0\\
h_4(\mathbf{x}) &= f_{1,4} + f_{1,3} - f_{1,2} = 0\\
h_5(\mathbf{x}) &= f_{2,4} + f_{2,2} - f_{2,2} = 0\\
h_6(\mathbf{x}) &= T_{C,in}f_{1,1} + T_{2,o}f_{2,3} - T_{1,i}f_{1,2} = 0\\
h_7(\mathbf{x}) &= T_{C,in}f_{2,1} + T_{1,o}f_{1,3} - T_{2,i}f_{2,2} = 0\\
h_8(\mathbf{x}) &= f_{1,2}\left(T_{1,o} - T_{1,i}\right) - 1000 = 0\\
h_9(\mathbf{x}) &= f_{2,2}\left(T_{2,o} - T_{2,i}\right) - 600 = 0\\
h_{10}(\mathbf{x}) &= \Delta T_{11} + T_{1,o} - 500 = 0\\
h_{11}(\mathbf{x}) &= \Delta T_{12} + T_{1,i} - 250 = 0\\
h_{12}(\mathbf{x}) &= \Delta T_{21} + T_{2,o} - 350 = 0\\
h_{13}(\mathbf{x}) &= \Delta T_{22} + T_{2,i} - 200 = 0
\end{aligned} \qquad (5.4)$$

$$\begin{aligned}
&Límites: 10 \le \Delta T_{11} \le 350; 10 \le \Delta T_{12} \le 350; 10 \le \Delta T_{21} \le 200; 10 \le \Delta T_{22} \le 200;\\
&\quad 150 \le T_{1,i}, T_{2,i} \le T_{C,o}\,; 150 \le T_{1,o}, T_{2,o} \le T_{C,o};\\
&\quad 0 \le f_{1,1}, f_{1,2}, f_{1,3}, f_{1,4}, f_{2,1}, f_{2,2}, f_{2,3}, f_{2,4} \le 10;\\
&Mínimo\ global: f(\mathbf{x}^*) = 4845.462\\
&\mathbf{x} = (\Delta T_{11} = 190, \Delta T_{12} = 40, \Delta T_{21} = 140, \Delta T_{22} = 50, f_{1,1} = 0, f_{1,2} = 10, f_{1,3} = 0,\\
&\quad f_{1,4} = 10, f_{2,1} = 10, f_{2,2} = 10, f_{2,3} = 10, f_{2,4} = 0, T_{1,i} = 210, T_{2,i} = 150,\\
&\quad T_{1,o} = 310, T_{2,o} = 210)
\end{aligned}$$

Este modelo optimiza 16 variables y posee 13 restricciones de igualdad. Para aumentar la eficiencia en la optimización de este problema, se redujo su dimensión. Si bien el número de grados de libertad es 3, solo es posible explicitar fácilmente 11 variables, por

lo tanto, se adoptó una reformulación con 11 variables calculadas ($\boldsymbol{x_C}$) y 5 manipuladas ($\boldsymbol{x_M}$). La elección de éstas últimas fue realizada como se explicó al comienzo del capítulo, estableciendo entonces:

Variables manipuladas ($\boldsymbol{x_M}$):

$$f_{1,2}, f_{2,2}, f_{1,1}, T_{1,o}, T_{2,o}$$

Variables calculadas ($\boldsymbol{x_C}$):

$$h_1(\mathbf{x}) \rightarrow f_{2,1} = 10 - f_{1,1}$$
$$h_2(\mathbf{x}) \rightarrow f_{2,3} = f_{1,2} - f_{1,1}$$
$$h_3(\mathbf{x}) \rightarrow f_{1,3} = f_{2,2} - f_{2,1}$$
$$h_4(\mathbf{x}) \rightarrow f_{1,4} = f_{1,2} - f_{1,3}$$
$$h_5(\mathbf{x}) \rightarrow f_{2,4} = f_{2,2} - f_{2,3}$$
$$h_8(\mathbf{x}) \rightarrow T_{1,i} = T_{1,o} - 1000/f_{1,2} \quad (\text{con } f_{1,2} \neq 0)$$
$$h_9(\mathbf{x}) \rightarrow T_{2,i} = T_{2,o} - 600/f_{2,2} \quad (\text{con } f_{2,2} \neq 0)$$
$$h_{10}(\mathbf{x}) \rightarrow \Delta T_{11} = 500 - T_{1,o}$$
$$h_{11}(\mathbf{x}) \rightarrow \Delta T_{12} = 250 - T_{1,i}$$
$$h_{12}(\mathbf{x}) \rightarrow \Delta T_{21} = 350 - T_{2,o}$$
$$h_{13}(\mathbf{x}) \rightarrow \Delta T_{22} = 200 - T_{2,i}$$

Entonces, el modelo a optimizar queda definido como sigue (Ec. 5.5):

Minimizar:

$$f(\mathbf{x}) = 1300\left(\frac{1000}{\frac{(\Delta T_{11}\Delta T_{12})}{30} + \frac{(\Delta T_{11}\Delta T_{12})}{6}}\right)^{0.6} + 1300\left(\frac{600}{\frac{(\Delta T_{21}\Delta T_{22})}{30} + \frac{(\Delta T_{21}\Delta T_{22})}{6}}\right)^{0.6}$$

Sujeto a:

$$h_6(\mathbf{x}) = T_{C,in}f_{1,1} + T_{2,o}f_{2,3} - T_{1,i}f_{1,2} = 0$$
$$h_7(\mathbf{x}) = T_{C,in}f_{2,1} + T_{1,o}f_{1,3} - T_{2,i}f_{2,2} = 0 \qquad (5.5)$$

$$Límites: 10 \leq \Delta T_{11} \leq 350; 10 \leq \Delta T_{12} \leq 350; 10 \leq \Delta T_{21} \leq 200; 10 \leq \Delta T_{22} \leq 200;$$
$$150 \leq T_{1,i}, T_{2,i} \leq T_{C,o}\,; 150 \leq T_{1,o}, T_{2,o} \leq T_{C,o};$$

$$0 \leq f_{1,1}, f_{1,2}, f_{1,3}, f_{1,4}, f_{2,1}, f_{2,2}, f_{2,3}, f_{2,4} \leq 10;$$

$Mínimo\ global: f(\boldsymbol{x}^*) = 4845.462$

$\boldsymbol{x}_{\boldsymbol{M}}^* = (f_{1,1} = 0, f_{1,2} = 10, f_{2,2} = 10, T_{1,o} = 310, T_{2,o} = 210)$

$\boldsymbol{x}_{\boldsymbol{C}}^* = (\Delta T_{11} = 190, \Delta T_{12} = 40, \Delta T_{21} = 140, \Delta T_{22} = 50, f_{1,3} = 0, f_{1,4} = 10,$
$f_{2,1} = 10, f_{2,3} = 10, f_{2,4} = 0, T_{1,i} = 210, T_{2,i} = 150)$

5.2.3 Separación non-sharp de propano, isobutano y n-butano en dos columnas de destilación

La destilación se produce cuando una mezcla líquida de dos o más sustancias de diferente volatilidad es separada en sus componentes por la aplicación de calor. Las columnas de destilación se encargan de realizar dicho proceso a través de una serie de etapas, en donde se desencadenan evaporaciones y condensaciones escalonadas acopladas entre sí.

Este proceso es fundamental en la elaboración de numerosos productos industriales, aunque es en el sector petroquímico donde adquiere una mayor importancia. El costo de operación de las columnas de destilación es, muchas veces, la parte más onerosa de los procesos industriales en los que intervienen. Por ello, disponer de técnicas prácticas para modelar estos equipos de una manera más o menos realista y desarrollar sistemas de control eficaces es muy importante para conseguir un funcionamiento seguro y estable de los sistemas de destilación industriales.

Debido a la importancia mencionada, los sistemas de columnas han sido estudiadas con fines de optimización en innumerables trabajos, tales como los presentados por Aggarwal y Floudas (1990), Adjiman y col. (1998) y Floudas y col. (1999). El problema abordado consiste en encontrar el diseño óptimo para la separación de una mezcla de propano, isobutano y n-butano con el objetivo de lograr dos productos, P_1 y P_2, que cumplan con ciertas especificaciones (Fig. 5.3). El primero debe tener un flujo total de 110 molkg/hr, compuesto por 30 molkg/hr de propano, 50 molkg/hr de isobutano y 30 molkg/hr de n-butano. El producto 2 debe poseer un flujo total de 190 molkg/hr,

compuesto por 70 molkg/hr de propano, 50 molkg/hr de isobutano y 70 molkg/hr de n-butano.

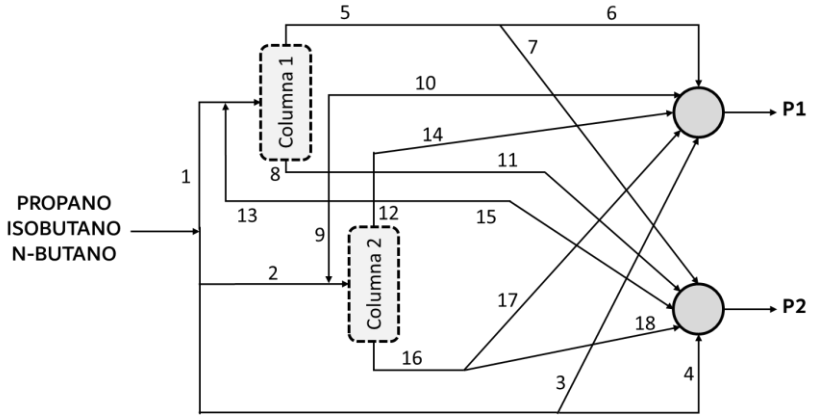

Fig. 5.3 Secuencia de destilación para la separación de la mezcla de propano, isobutano y n-butano.

A continuación, se presenta la formulación matemática de este problema (Ec. 5.6), la cual fue extraída de Andrei (2013) (Aplicación 7.9).

Minimizar:

$$f(\boldsymbol{x}) = 0.9979 + 0.00432F_1 + 0.00432F_{13} + 0.01517F_2 + 0.01517F_9$$

Sujeto a:

$$h_1(\boldsymbol{x}) = F_1 + F_2 + F_3 + F_4 - 300 = 0$$
$$h_2(\boldsymbol{x}) = F_5 - F_6 - F_7 = 0$$
$$h_3(\boldsymbol{x}) = F_8 - F_9 - F_{10} - F_{11} = 0$$
$$h_4(\boldsymbol{x}) = F_{12} - F_{13} - F_{14} - F_{15} = 0$$
$$h_5(\boldsymbol{x}) = F_{16} - F_{17} - F_{18} = 0$$
$$h_6(\boldsymbol{x}) = F_{13}x_{A12} - F_5 + 0.333F_1 = 0$$
$$h_7(\boldsymbol{x}) = F_{13}x_{B12} - F_8x_{B8} + 0.333F_1 = 0$$
$$h_8(\boldsymbol{x}) = -F_8x_{C8} + 0.333F_1 = 0$$
$$h_9(\boldsymbol{x}) = -F_{12}x_{A12} - 0.333F_2 = 0$$
$$h_{10}(\boldsymbol{x}) = F_9x_{B8} - F_{12}x_{B12} + 0.333F_2 = 0$$
$$h_{11}(\boldsymbol{x}) = F_9x_{C8} - F_{16} + 0.333F_2 = 0$$
$$h_{12}(\boldsymbol{x}) = F_{14}x_{A12} + 0.333F_3 + F_6 - 30 = 0$$
$$h_{13}(\boldsymbol{x}) = F_{10}x_{B8} + F_{14}x_{B12} + 0.333F_3 - 50 = 0$$

$$h_{14}(\boldsymbol{x}) = F_{10}x_{C8} + 0.333F_3 + F_{17} - 30 = 0$$
$$h_{15}(\boldsymbol{x}) = x_{A8} = 0$$
$$h_{16}(\boldsymbol{x}) = x_{B8} + x_{C8} - 1 = 0$$
$$h_{17}(\boldsymbol{x}) = x_{A12} + x_{B12} - 1 = 0$$
$$h_{18}(\boldsymbol{x}) = x_{C12} = 0 \qquad (5.6)$$

$Límites: 0 \leq F_i \leq 300, i = 1, \ldots, 18$

$0 \leq x_{ji} \leq 1, j \in \{A, B, C\}, i \in \{8,12\}$

$Mínimo\ global: f(\boldsymbol{x}^*) = 1.864159$

$\boldsymbol{x}^* = (F_1 = 60.060, F_2 = 0, F_3 = 90.09, F_4 = 149.849, F_5 = 20, F_6 = 0, F_7 = 20,$
$F_8 = 40, F_9 = 40, F_{10} = 0, F_{11} = 0, F_{12} = 20, F_{13} = 0, F_{14} = 20, F_{15} = 0,$
$F_{16} = 20, F_{17} = 0, F_{18} = 20, x_{A8} = 0, x_{B8} = 0.5, x_{C8} = 0.5, x_{A12} = 0,$
$x_{B12} = 1, x_{C12} = 0)$

El modelo que representa la separación de propano, isobutano y n-butano en dos columnas de destilación consta de 24 variables y 18 restricciones de igualdad. En este caso, los grados de libertad son 6. Con el criterio de selección antes comentado, se optó por elegir 9 variables manipuladas ($\boldsymbol{x_M}$), resultando así 15 variables calculadas ($\boldsymbol{x_C}$) a partir de despejarlas de algunas de las restricciones de igualdad. En este caso tampoco fue posible reducir a 6 las variables manipuladas, como el cálculo de los grados de libertad indicaba, debido a la imposibilidad de despejar fácilmente las 18 variables a partir de las ecuaciones de igualdad.

Variables manipuladas ($\boldsymbol{x_M}$):

$$F_1, F_2, F_3, x_{B8}, x_{A12}, x_{A8}, x_{C12}, F_9, F_{10}$$

Variables calculadas ($\boldsymbol{x_C}$):

$$h_1(\boldsymbol{x}) \rightarrow F_4 = 300 - F_1 - F_2 - F_3$$
$$h_2(\boldsymbol{x}) \rightarrow F_7 = F_5 - F_6$$
$$h_3(\boldsymbol{x}) \rightarrow F_{11} = F_8 - F_9 - F_{10}$$
$$h_4(\boldsymbol{x}) \rightarrow F_{15} = F_{12} - F_{13} - F_{14}$$
$$h_5(\boldsymbol{x}) \rightarrow F_{18} = F_{16} - F_{17}$$

$$h_6(\boldsymbol{x}) \rightarrow F_5 = F_{13}x_{A12} + 0.333F_1$$
$$h_7(\boldsymbol{x}) \rightarrow F_{13} = (F_8x_{B8} - 0.333F_1)/x_{B12} \quad (\text{con } x_{B12} \neq 0)$$
$$h_8(\boldsymbol{x}) \rightarrow F_8 = 0.333F_1/x_{C8} \quad (\text{con } x_{C8} \neq 0)$$
$$h_{10}(\boldsymbol{x}) \rightarrow F_{12} = (F_9x_{B8} + 0.333F_2)/x_{B12} \quad (\text{con } x_{B12} \neq 0)$$
$$h_{11}(\boldsymbol{x}) \rightarrow F_{16} = F_9x_{C8} + 0.333F_2$$
$$h_{12}(\boldsymbol{x}) \rightarrow F_6 = 30 - F_{14}x_{A12} - 0.333F_3$$
$$h_{13}(\boldsymbol{x}) \rightarrow F_{14} = (50 - F_{10}x_{B8} - 0.333F_3)/x_{B12}$$
$$(\text{con } x_{B12} \neq 0)$$
$$h_{14}(\boldsymbol{x}) \rightarrow F_{17} = 30 - F_{10}x_{C8} - 0.333F_3$$
$$h_{16}(\boldsymbol{x}) \rightarrow x_{C8} = 1 - x_{B8}$$
$$h_{17}(\boldsymbol{x}) \rightarrow x_{B12} = 1 - x_{A12}$$

Finalmente, la función a optimizar queda sujeta solo a tres restricciones de igualdad (Ec. 5.7):

Minimizar:

$$f(\boldsymbol{x}) = 0.9979 + 0.00432F_1 + 0.00432F_{13} + 0.01517F_2 + 0.01517F_9$$

Sujeto a:

$$h_9(\boldsymbol{x}) = -F_{12}x_{A12} - 0.333F_2 = 0$$
$$h_{15}(\boldsymbol{x}) = x_{A8} = 0$$
$$h_{18}(\boldsymbol{x}) = x_{C12} = 0 \qquad (5.7)$$

$Límites: 0 \leq F_1, F_2, F_9, F_{12}, F_{13} \leq 300,\ 0 \leq x_{A8}, x_{A12}, x_{C12} \leq 1$

$Mínimo\ global: f(\vec{x}^*) = 1.864159$

$\boldsymbol{x_M^*} = (F_1 = 60.060, F_2 = 0, F_3 = 90.09, F_9 = 40, F_{10} = 0, x_{A8} = 0, x_{B8} = 0.5,$
$x_{A12} = 0, x_{C12} = 0)$

$\boldsymbol{x_C^*} = (F_4 = 149.849, F_5 = 20, F_6 = 0, F_7 = 20, F_8 = 40, F_{11} = 0, F_{12} = 20,$
$F_{13} = 0, F_{14} = 20, F_{15} = 0, F_{16} = 20, F_{17} = 0, F_{18} = 20, x_{C8} = 0.5,$
$x_{B12} = 1)$

5.2.4 Problema de mezclado de petróleo: Ben-Tal5

Como se mencionó en el capítulo anterior, la mezcla de petróleo crudo se produce cuando corrientes con distintas propiedades se mezclan en un tanque de almacenamiento a fin de obtener productos finales que tienen diversas especificaciones de calidad de los componentes.

En esta sección se aborda el modelo Ben-Tal 5 analizado en Adhya y col. (1999). El mismo es de un mayor tamaño que los de la familia Haverly estudiados en el Capítulo 4, con más términos bilineales y un gran número de óptimos locales, constituyendo así un problema mucho más desafiante para un solver no lineal.

En la Fig. 5.4 se observa el diagrama de flujo del Ben-Tal 5, en donde c_{ij} representa el costo unitario de la corriente "i" en la pileta "j"; λ_{ijw} es la especificación de calidad del componente "w" en la corriente "i" de la pileta "j"; d_k es el costo unitario del producto "k"; S_k es la demanda del producto "k" y z_{kw} es la especificación de calidad del componente "w" del producto "k".

Este modelo consta de cinco fuentes de petróleo crudo (f_{1j}, f_{5j}) con dos especificaciones distintas de calidad de las corrientes en la pileta ($w1$ y $w2$). Las mismas se mezclan en tres piletas (1, 2 y 3), para luego alcanzar las especificaciones impuestas para cinco productos finales. Se requiere además que $f_{11} + f_{21} + f_{31} \leq 50, f_{12} + f_{22} + f_{32} \leq 50$ y que $f_{13} + f_{23} + f_{33} \leq 50$.

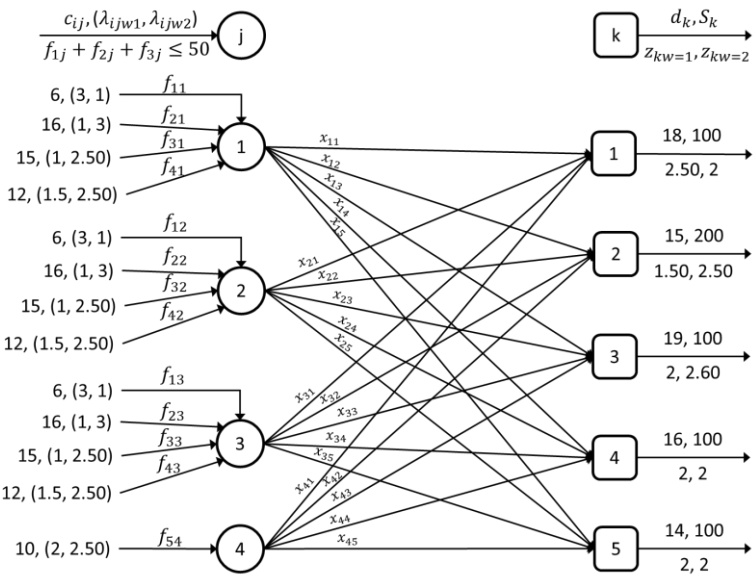

Fig. 5.4 Mezclado de petróleo Ben-Tal5

A partir de la Fig. 5.5 se construyó el planteo matemático (Ec. 5.8) que se describe a continuación (Andrei, 2013) (Aplicación 7.8):

Minimizar:

$$\begin{aligned} f(\boldsymbol{x}) = {} & 6f_{11} + 16f_{21} + 15f_{31} + 12f_{41} + 6f_{12} + 16f_{22} + 15f_{32} \\ & + 12f_{42} + 6f_{13} + 16f_{23} + 15f_{33} + 12f_{43} + 10f_{54} \\ & - 18(x_{11} + x_{21} + x_{31} + x_{41}) - 15(x_{12} + x_{22} + x_{32} + x_{42}) \\ & - 19(x_{13} + x_{23} + x_{33} + x_{43}) - 16(x_{14} + x_{24} + x_{34} + x_{44}) \\ & - 14(x_{15} + x_{25} + x_{35} + x_{45}) \end{aligned}$$

Balance de masa en las piletas:

$$h_1(\boldsymbol{x}) = f_{11} + f_{21} + f_{31} + f_{41} - x_{11} - x_{12} - x_{13} - x_{14} - x_{15} = 0$$
$$h_2(\boldsymbol{x}) = f_{12} + f_{22} + f_{32} + f_{42} - x_{21} - x_{22} - x_{23} - x_{24} - x_{25} = 0$$
$$h_3(\boldsymbol{x}) = f_{13} + f_{23} + f_{33} + f_{43} - x_{31} - x_{32} - x_{33} - x_{34} - x_{35} = 0$$
$$h_4(\boldsymbol{x}) = f_{54} - x_{41} - x_{42} - x_{43} - x_{44} - x_{45} = 0$$

Balance del componente 1 en las piletas:

$$h_5(\boldsymbol{x}) = q_{11}(x_{11} + x_{12} + x_{13} + x_{14} + x_{15}) - 0.03f_{11} - 0.01f_{21} - 0.01f_{31}$$

$$-0.015f_{41} = 0$$

$$h_6(\boldsymbol{x}) = q_{12}(x_{21} + x_{22} + x_{23} + x_{24} + x_{25}) - 0.03f_{12} - 0.01f_{22} - 0.01f_{32}$$
$$-0.015f_{42} = 0$$

$$h_7(\boldsymbol{x}) = q_{13}(x_{31} + x_{32} + x_{33} + x_{34} + x_{35}) - 0.03f_{13} - 0.01f_{23} - 0.01f_{33}$$
$$-0.015f_{43} = 0$$

Balance del componente 2 en las piletas:

$$h_8(\boldsymbol{x}) = q_{21}(x_{11} + x_{12} + x_{13} + x_{14} + x_{15}) - 0.01f_{11} - 0.03f_{21} - 0.025f_{31}$$
$$-0.025f_{41} = 0$$

$$h_9(\boldsymbol{x}) = q_{22}(x_{21} + x_{22} + x_{23} + x_{24} + x_{25}) - 0.01f_{12} - 0.03f_{22} - 0.025f_{32}$$
$$-0.025f_{42} = 0$$

$$h_{10}(\boldsymbol{x}) = q_{23}(x_{31} + x_{32} + x_{33} + x_{34} + x_{35}) - 0.01f_{13} - 0.03f_{23} - 0.025f_{33}$$
$$-0.025f_{34} = 0$$

Requerimientos máximos del componente 1 en los productos:

$$g_1(\boldsymbol{x}) = q_{11}x_{11} + q_{12}x_{21} + q_{13}x_{31} + 0.02x_{41} - 0.015(x_{11} + x_{21} + x_{31} + x_{41}) \leq 0$$
$$g_2(\boldsymbol{x}) = q_{11}x_{12} + q_{12}x_{22} + q_{13}x_{32} + 0.02x_{42} - 0.015(x_{12} + x_{22} + x_{32} + x_{42}) \leq 0$$
$$g_3(\boldsymbol{x}) = q_{11}x_{13} + q_{12}x_{23} + q_{13}x_{33} + 0.02x_{43} - 0.015(x_{13} + x_{23} + x_{33} + x_{43}) \leq 0$$
$$g_4(\boldsymbol{x}) = q_{11}x_{14} + q_{12}x_{24} + q_{13}x_{34} + 0.02x_{44} - 0.015(x_{14} + x_{24} + x_{34} + x_{44}) \leq 0$$
$$g_5(\boldsymbol{x}) = q_{11}x_{15} + q_{12}x_{25} + q_{13}x_{35} + 0.02x_{45} - 0.015(x_{15} + x_{25} + x_{35} + x_{45}) \leq 0$$

Requerimientos máximos del componente 2 en los productos:

$$g_6(\boldsymbol{x}) = q_{21}x_{11} + q_{22}x_{21} + q_{23}x_{31} + 0.025x_{41} - 0.02(x_{11} + x_{21} + x_{31} + x_{41}) \leq 0$$
$$g_7(\boldsymbol{x}) = q_{21}x_{12} + q_{22}x_{22} + q_{23}x_{32} + 0.025x_{42} - 0.025(x_{12} + x_{22} + x_{32} + x_{42}) \leq 0$$
$$g_8(\boldsymbol{x}) = q_{21}x_{13} + q_{22}x_{23} + q_{23}x_{33} + 0.025x_{43} - 0.026(x_{13} + x_{23} + x_{33} + x_{43}) \leq 0$$
$$g_9(\boldsymbol{x}) = q_{21}x_{14} + q_{22}x_{24} + q_{23}x_{34} + 0.025x_{44} - 0.02(x_{14} + x_{24} + x_{34} + x_{44}) \leq 0$$
$$g_{10}(\boldsymbol{x}) = q_{21}x_{15} + q_{22}x_{25} + q_{23}x_{35} + 0.025x_{45} - 0.02(x_{15} + x_{25} + x_{35} + x_{45}) \leq 0$$

Requerimientos máximos en los productos:

$$
\begin{aligned}
&g_{11}(\boldsymbol{x}) = x_{11} + x_{21} + x_{31} + x_{41} - s_1 \leq 0\\
&g_{12}(\boldsymbol{x}) = x_{12} + x_{22} + x_{32} + x_{42} - s_2 \leq 0\\
&g_{13}(\boldsymbol{x}) = x_{13} + x_{23} + x_{33} + x_{43} - s_3 \leq 0\\
&g_{14}(\boldsymbol{x}) = x_{14} + x_{24} + x_{34} + x_{44} - s_4 \leq 0\\
&g_{15}(\boldsymbol{x}) = x_{15} + x_{25} + x_{35} + x_{45} - s_5 \leq 0\\
&g_{16}(\boldsymbol{x}) = f_{11} + f_{21} + f_{31} - 50 \leq 0\\
&g_{17}(\boldsymbol{x}) = f_{12} + f_{22} + f_{32} - 50 \leq 0\\
&g_{18}(\boldsymbol{x}) = f_{13} + f_{23} + f_{33} - 50 \leq 0
\end{aligned}
\tag{5.8}
$$

$Límites: 0 \leq x_{12}, x_{22}, x_{32}, x_{42} \leq 200,\ 0.01 \leq q_{11}, q_{12}, q_{13}, q_{21}, q_{22}, q_{23} \leq 0.03,$

$0 \leq x_{11}, x_{13}, x_{14}, x_{15}, x_{21}, x_{23}, x_{24}, x_{25}, x_{31}, x_{33}, x_{34}, x_{35}, x_{41}, x_{43}, x_{44}, x_{45} \leq 100,$

$0 \leq f_{ij} \leq 600.$

$Mínimo\ global: f(\boldsymbol{x}^*) =$-3500

$$
\begin{aligned}
\boldsymbol{x}^* = (&f_{11} = 50, f_{21} = 0,\ f_{31} = 0,\ f_{41} = 101.48, f_{12} = 50, f_{22} = 0, f_{32} = 0,\\
&f_{42} = 0\ , f_{13} = 0, f_{23} = 0\ , f_{33} = 0\ , f_{43} = 231.85\ , f_{54} = 166.66\ ,\\
&x_{11} = 0, x_{12} = 0, x_{13} = 0, x_{14} = 94.26, x_{15} = 57.22, x_{21} = 33.33,\\
&x_{22} = 0, x_{23} = 0, x_{24} = 2.22, x_{25} = 14.446, x_{31} = 0, x_{32} = 200,\\
&x_{33} = 0, x_{34} = 3.52, x_{35} = 28.334, x_{41} = 66.667, x_{42} = 0, x_{43} = 100,\\
&x_{44} = 0, x_{45} = 0, q_{11} = 0.02, q_{12} = 0.03, q_{13} = 0.015, q_{21} = 0.02,\\
&q_{22} = 0.01, q_{23} = 0.025)
\end{aligned}
$$

El problema consta de 39 variables sujeto a 10 restricciones de igualdad y 18 restricciones de desigualdad. A través del método de selección ya explicado, se eligieron 29 variables para manipular ($\boldsymbol{x_M}$), que coincide con los grados de libertad del modelo, resultando entonces en 10 variables calculadas ($\boldsymbol{x_C}$) de forma explícita a partir de las ecuaciones de igualdad:

Variables manipuladas ($\boldsymbol{x_M}$):

$$
\begin{aligned}
&x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{21}, x_{22}, x_{23}, x_{24}, x_{25}, x_{31}, x_{32}, x_{33}, x_{34}, x_{35},\\
&x_{41}, x_{42}, x_{43}, x_{44}, x_{45}, f_{21}, f_{31}, f_{41}, f_{22}, f_{32}, f_{42}, f_{23}, f_{33}, f_{43}
\end{aligned}
$$

Variables calculadas ($\boldsymbol{x_C}$):

Balance de masa en las piletas:

$$h_1(\boldsymbol{x}) \rightarrow f_{11} = x_{11} + x_{12} + x_{13} + x_{14} + x_{15} - f_{21} - f_{31} - f_{41}$$
$$h_2(\boldsymbol{x}) \rightarrow f_{12} = x_{21} + x_{22} + x_{23} + x_{24} + x_{25} - f_{22} - f_{32} - f_{42}$$
$$h_3(\boldsymbol{x}) \rightarrow f_{13} = x_{31} + x_{32} + x_{33} + x_{34} + x_{35} - f_{23} - f_{33} - f_{43}$$
$$h_4(\boldsymbol{x}) \rightarrow f_{54} = x_{41} + x_{42} + x_{43} + x_{44} + x_{45}$$

Balance del componente 1 en las piletas:

$$h_5(\boldsymbol{x}) \rightarrow q_{11} = \frac{0.03f_{11} + 0.01f_{21} + 0.01f_{31} + 0.015f_{41}}{x_{11} + x_{12} + x_{13} + x_{14} + x_{15}}$$
$$\text{Con } x_{11} + x_{12} + x_{13} + x_{14} + x_{15} \neq 0$$

$$h_6(\boldsymbol{x}) \rightarrow q_{12} = \frac{0.03f_{12} + 0.01f_{22} + 0.01f_{32} + 0.015f_{42}}{x_{21} + x_{22} + x_{23} + x_{24} + x_{25}}$$
$$\text{Con } x_{21} + x_{22} + x_{23} + x_{24} + x_{25} \neq 0$$

$$h_7(\boldsymbol{x}) \rightarrow q_{13} = \frac{0.03f_{13} + 0.01f_{23} + 0.01f_{33} + 0.015f_{43}}{x_{31} + x_{32} + x_{33} + x_{34} + x_{35}}$$
$$\text{Con } x_{31} + x_{32} + x_{33} + x_{34} + x_{35} \neq 0$$

Balance del componente 2 en las piletas:

$$h_8(\boldsymbol{x}) \rightarrow q_{21} = \frac{0.01f_{11} + 0.03f_{21} + 0.025f_{31} + 0.025f_{41}}{x_{11} + x_{12} + x_{13} + x_{14} + x_{15}}$$
$$h_9(\boldsymbol{x}) \rightarrow q_{22} = \frac{0.01f_{12} + 0.03f_{22} + 0.025f_{32} + 0.025f_{42}}{x_{21} + x_{22} + x_{23} + x_{24} + x_{25}}$$
$$h_{10}(\boldsymbol{x}) \rightarrow q_{23} = \frac{0.01f_{13} + 0.03f_{23} + 0.025f_{33} + 0.025f_{43}}{x_{31} + x_{32} + x_{33} + x_{34} + x_{35}}$$

Entonces, la nueva función a optimizar no posee restricciones de igualdad, aunque sigue sujeta a las 18 restricciones de desigualdad (Ec. 5.9):

Minimizar:

$$\begin{aligned} f(\boldsymbol{x}) = {} & 6f_{11} + 16f_{21} + 15f_{31} + 12f_{41} + 6f_{12} + 16f_{22} + 15f_{32} \\ & +12f_{42} + 6f_{13} + 16f_{23} + 15f_{33} + 12f_{43} + 10f_{54} \\ & -18(x_{11} + x_{21} + x_{31} + x_{41}) - 15(x_{12} + x_{22} + x_{32} + x_{42}) \\ & -19(x_{13} + x_{23} + x_{33} + x_{43}) - 16(x_{14} + x_{24} + x_{34} + x_{44}) \\ & -14(x_{15} + x_{25} + x_{35} + x_{45}) \end{aligned}$$

Sujeto a las restricciones de desigualdad ya mencionadas:

$$g_1(\boldsymbol{x}) \leq 0 \ \ldots \ g_{18}(\boldsymbol{x}) \leq 0 \tag{5.9}$$

$Límites$: $0 \leq x_{12}, x_{22}, x_{32}, x_{42} \leq 200$, $0.01 \leq q_{11}, q_{12}, q_{13}, q_{21}, q_{22}, q_{23} \leq 0.03$,

$0 \leq x_{11}, x_{13}, x_{14}, x_{15}, x_{21}, x_{23}, x_{24}, x_{25}, x_{31}, x_{33}, x_{34}, x_{35}, x_{41}, x_{43}, x_{44}, x_{45} \leq 100$,

$0 \leq f_{ij} \leq 600$,

$Mínimo\ global$: $f(\boldsymbol{x}^*)$ =-3500

$\boldsymbol{x}_{\boldsymbol{M}}^* = (f_{21} = 0,\ f_{31} = 0,\ f_{41} = 101.48, f_{22} = 0, f_{32} = 0, f_{42} = 0\ , f_{23} = 0\ ,$
$f_{33} = 0\ , f_{43} = 231.85\ , x_{11} = 0, x_{12} = 0, x_{13} = 0, x_{14} = 94.26, x_{15} = 57.22,$
$x_{21} = 33.33, x_{22} = 0, x_{23} = 0, x_{24} = 2.22, x_{25} = 14.446, x_{31} = 0, x_{32} = 200,$
$x_{33} = 0, x_{34} = 3.52, x_{35} = 28.334, x_{41} = 66.667, x_{42} = 0, x_{43} = 100, x_{44} = 0,$
$x_{45} = 0)$

$\boldsymbol{x}_{\boldsymbol{C}}^* = (f_{11} = 50, f_{12} = 50, f_{13} = 0, f_{54} = 166.66\ , q_{11} = 0.02, q_{12} = 0.03,$
$q_{13} = 0.015, q_{21} = 0.02, q_{22} = 0.01, q_{23} = 0.025)$

5.3 Resultados

En la Tabla 5.1 se resumen las características de los problemas descriptos en la sección anterior, así como también las particularidades de sus reformulaciones en los casos correspondientes.

Tabla 5.1. Resumen de características de modelos sección 5.2

Mod	Descripción	D	D^*	n	n^*	m	f^*
5.2.1	Diseño de un sistema de refrigeración	14	14	0	0	15	0.032
5.2.2	Diseño red de intercambiadores de calor	16	5	13	2	0	4845.4
5.2.3	Separación columnas de destilación	24	9	18	3	0	1.864
5.2.4	Mezclado de petróleo BenTal5	39	29	10	0	0	-3500

En la columna "**Mod**", que se refiere al Modelo, se proporciona la referencia de la sección correspondiente al problema. En las columnas "D", "n" y "m" se informa la cantidad de variables, restricciones de igualdad y desigualdad de cada problema respectivamente. En las columnas "D^*" y "n^*" se indica, respectivamente, la cantidad de variables de cada problema que se ha reformulado manualmente en el espacio reducido y las restricciones de igualdad que deben cumplirse con esta nueva formulación del modelo. Por su parte, f^* es el mejor valor conocido de la función objetivo.

La Tabla 5.2 muestra los resultados obtenidos al ejecutar el PSO en serie y en paralelo según la parametrización enunciada en la sección 4.1. En este caso, los problemas se corrieron en el siguiente sistema:

Sistema 3

- CPU: Intel Core i7-7700k con 4 núcleos, 3.60 GHz y 16GB de memoria RAM compartida.
- GPU: GeForce GTX 480 con 480 núcleos y 1.5 GB de RAM dedicada.

La Tabla de resultados se presenta de la misma manera que en el Capítulo 4. La columna "**Tasa de factibilidad**" indica el porcentaje de las ejecuciones que alcanzaron soluciones factibles bajo cierta tolerancia impuesta ($TAV < 10^{-3}$). Por otro lado, la columna "**Tasa de optimalidad**" especifica el porcentaje de ejecuciones factibles que alcanzaron la solución óptima global con una tolerancia inferior a 10^{-1} en el Error Relativo, definido en la Ec. 5.10:

$$Error\ Relativo = \begin{cases} \frac{|f - f^*|}{f^*} & Si\ f^* \neq 0 \\ |f - f^*|\ Si\ \ f^* = 0 \end{cases} \quad (5.10)$$

Donde f^* es valor de la función objetivo óptima global conocida del problema y f es el valor de la función obtenido por el algoritmo. Es necesario recordar que, tal como se mencionó en el Capítulo anterior, la tasa de optimalidad se calcula sobre aquellas soluciones que resultan factibles según la tolerancia impuesta, entonces, primero se verifica el criterio de factibilidad y a las soluciones que lo cumplan se les determina la tasa de optimalidad.

Asimismo, la columna "$\boldsymbol{T_{CPU}}$" se refiere al tiempo promedio de aplicar el PSO en serie 30 ejecuciones independientes, mientras que la columna "**Speed-Up**" informa la ganancia de velocidad que se obtuvo al emplear el PSO paralelizado en la GPU, también determinado como un promedio de 30 ejecuciones.

Tabla 5.2. Resultados modelos sección 5.2

Mod	Tasa de Factibilidad	Tasa de Optimalidad	Error Relativo	T_{CPU} (seg)	Speed-Up
5.2.1	100.00	0.00	93.62	682.92	315.83
5.2.2	81.48	33.33	0.22	96.54	64.80
5.2.3	100.00	0.00	2.80	138.47	95.76
5.2.4	100.00	0.00	0.37	132.42	55.44

Como puede observarse en la Tabla 5.2, tres de los cuatro problemas convergen a una solución factible en todas las ejecuciones del algoritmo (Tasa de factibilidad 100%). Por su parte, el modelo 5.2.2 converge a una solución factible en el 81.48% de las corridas. Esto sucede porque existen 3 evaluaciones de las 30, en donde la violación total de las restricciones (TAV) es aproximadamente igual a $5*10^{-3}$, siendo este valor superior a la tolerancia impuesta de 10^{-3}.

Por otra parte, la mayoría de los problemas solo alcanza óptimos locales, ya que su tasa de optimalidad global es nula en tres de los cuatro casos de estudio. En el modelo 5.2.1

el TAV obtenido luego de algunas iteraciones alcanza siempre el mismo valor, el cual resulta inferior a la tolerancia impuesta, considerándose una solución factible, por lo tanto, la búsqueda sólo se guía por la mejoría que se pueda presentar en la función objetivo. Esto podría indicar que el PSO se dirige siempre a la misma región factible con un óptimo local cerca, en vez del global, ocasionando que el algoritmo encuentre una solución subóptima en todos los casos. Este comportamiento también se observa en los modelos 5.2.3 y 5.2.4.

En el problema 5.2.1 ($f^* = 0.032$), cuyo error relativo fue de 93.62, se observó que un 43.33% de las corridas se aproxima a un valor de la función objetivo de 0.350, mientras que un 40% alcanza una solución de 1.800 y el 17.67% restante alcanza distintos valores más apartados aún del óptimo conocido. Sin embargo, se puede concluir que un alto porcentaje de las evaluaciones (83.33%) converge a soluciones subóptimas más cercanas al óptimo global, mientras que el resto alcanza diversas soluciones más alejadas del mismo.

El modelo 5.2.2 ($f^* = 4845.4$) presentó un error relativo de 0.22 ya que el 33.33% de las evaluaciones que alcanzaron soluciones factibles, encuentran el óptimo global. En cambio, en el 66.66% de las evaluaciones restantes se alcanza una solución que se encuentra entre 5000 y 6500.

En el caso de estudio 5.2.3 ($f^* = 1.864$), el error relativo fue de 2.80 debido a que un 10% de las corridas llegó a valores de la función objetivo entre 6.000 y 7.000, mientras que el 90% restante alcanza soluciones entre 7.000 y 8.000. Si bien las soluciones se encuentran dentro del orden del óptimo global, nunca se logra avanzar más hacia este punto.

Finalmente, el problema 5.2.4 ($f^* = -3500$) alcanzó un error relativo de 0.3 a causa de que el 73.33% de las evaluaciones logró soluciones que varían entre -2000 y -3000, valores próximos al optimo global. Por otra parte, el 26.67% restante se detuvo en un subóptimo más alejado, entre -1000 y -2000.

Para intentar mejorar las tasas de factibilidad y optimalidad del algoritmo en estos problemas, se aplicó también la estrategia de reducción secuencial de caja explicada en el capítulo anterior.

La Tabla 5.3 presenta los resultados de ejecutar PSO con reducción secuencial de caja, en serie y paralelo, en el mismo sistema descrito en esta sección y bajo la misma parametrización que la utilizada en el Capítulo 4.

Tabla 5.3. Resultados modelos sección 5.2 con PSO con reducción secuencial de caja

Mod	Tasa de Factibilidad	Tasa de Optimalidad	Error Relativo	T_{CPU} (seg)	Speed-Up
5.2.1	100.00	0.00	13.02	654.74	317.65
5.2.2	100.00	100.00	0.00	93.99	68.36
5.2.3	100.00	0.00	0.99	137.69	94.79
5.2.4	100.00	0.00	0.35	125.93	50.92

Al resolver los distintos problemas con el PSO con reducción secuencial de caja se logró disminuir considerablemente el error relativo. De esta manera, el modelo 5.2.2 encontró, en todas las evaluaciones, el óptimo global según el criterio de optimalidad impuesto (Tabla 5.4), cuando con el PSO estándar solo cerca del 82% de las soluciones eran factibles y de ellas, un 33.33% alcanzaba la solución real.

Si bien para el resto de los casos tampoco se pudo encontrar el óptimo de acuerdo a la tolerancia impuesta, la solución local encontrada al utilizar el PSO con esta estrategia es superior a la hallada con el PSO estándar.

El modelo 5.2.1 ($f^* = 0.032$) se estanca siempre en un único optimo local, 0.450, que resulta mucho más cercano al global que la mejor solución encontrada con el PSO estándar (Error relativo de 13.02 versus 93.62).

Por su parte, el problema 5.2.3 ($f^* = 1.864$) ahora alcanza, en el 80% de los casos, valores de la función objetivo entre 3.000 y 5.000, cuando antes convergía a 6.950 en el mejor de los casos (Error relativo de 0.99 versus 2.80). Mientras que el 20% restante converge a valores entre 5.000 y 6.400.

Asimismo, en 5.2.4 ($f^* = -3500$), las 30 evaluaciones realizadas con el PSO con reducción secuencial de caja alcanzan el mismo subóptimo igual a -2286.03. De esa forma, se mejora el 56.67% de las soluciones obtenidas con el PSO estándar. Sin embargo, el 44.3% de las corridas restantes convergían a valores más cercanos al óptimo con la versión original. En conclusión, para este modelo la técnica de reducción gradual de la región de búsqueda conduce a que todas las ejecuciones lleven al mismo resultado, un subóptimo no tan alejado del global, pero con el PSO estándar se lograba que algunas ejecuciones alcancen mejores resultados (Error relativo de 0.35 versus 0.37).

En cuanto a la aceleración lograda por la GPU se obtuvieron interesantes reducciones del tiempo de cómputo. Debido a que el Sistema utilizado en esta ocasión posee menor capacidad de cómputo que los Sistemas 1 y 2 descrito en el Capítulo 4 por tratarse de una GPU más antigua, no se alcanzaron Speed-Ups tan significativos como en el capítulo anterior.

En particular, el modelo 5.2.1 muestra un Speed-Up muy elevado. Esto se debe a que dicho problema tiene 15 restricciones de desigualdad, mientras que los otros como mucho tienen 3 restricciones de igualdad (luego de la simplificación), entonces el tiempo de ejecución en la CPU es mayor que en los otros casos, por lo que el Speed-Up termina destacándose por sobre el resto debido a que el tiempo serie es mayor para este modelo.

Para ilustrar el desempeño de estos modelos empleando una plataforma comercial, se los resolvió en GAMS utilizando los solvers locales CONOPT y MINOS inicializando las variables en tres puntos distintos: límite superior, límite inferior y punto intermedio. Los resultados se muestran a continuación (Tabla 5.4):

Tabla 5.4. Resultados modelos sección 5.2 con GAMS

Mod	Puntos Iniciales	CONOPT	MINOS	BARON
5.2.1 ($f^* = 0.032$)	Punto Intermedio	0.032	0.032	0.032 t = 1000 seg
	Límite Superior	INF	0.032	
	Límite Inferior	0.032	0.032	
5.2.2 ($f^* = 4845.4$)	Punto Intermedio	5937.43	4845.46	4845.46 t = 0.664 seg
	Límite Superior	4845.46	4845.46	
	Límite Inferior	INF	5937.43	
5.2.3 ($f^* = 1.864$)	Punto Intermedio	1.864	1.864	1.864 t = 0.628 seg
	Límite Superior	1.864	1.864	
	Límite Inferior	INF	INF	
5.2.4 ($f^* = -3500$)	Punto Intermedio	-3433.33	-2733.33	-3433.33 t = 1000 seg
	Límite Superior	-3433.33	-900	
	Límite Inferior	-900	-900	

Como puede observarse en la Tabla 5.4 la solución encontrada por estos solvers depende del punto de inicial proporcionado, convirgiendo en algunos casos a óptimos globales o locales y, en otros, a soluciones infactibles. En particular, para el problema 5.2.4 ninguno de los solvers convirgió al óptimo global desde ninguno de los puntos de partida. En la Tabla 5.4 también se presentan los resultados obtenidos empleando el solver global BARON, el cual no requiere puntos iniciales. BARON alcanzó la solución global en todos los casos menos en el modelo 5.2.4, el cual, junto con el modelo 5.2.1, demandaron un tiempo de cómputo igual al reslim de 1000 segundos que tiene el sistema por defecto.

A partir de los resultados obtenidos al optimizar estos cuatro problemas en GAMS con distintos solvers, se puede concluir que, debido a las grandes diferencias de las soluciones alcanzadas a partir de los distintos valores iniciales y al tiempo de ejecución que demandaron algunos de los modelos, éstos son realmente complejos de resolver.

CAPÍTULO 6

CONCLUSIONES Y TRABAJOS FUTUROS

6.1 Contribución de la tesis

En esta tesis se desarrolló una implementación computacional de un algoritmo de optimización no-lineal de propósito general, basado en un método de búsqueda estocástica. La motivación principal de este proyecto fue la de contar con una herramienta de optimización propia que permita el desarrollo de actividades de investigación y transferencia sin depender exclusivamente de software comercial (GAMS, Matlab, etc.), el cual suele resultar muy oneroso.

En particular, se adoptó la técnica PSO debido principalmente a que se ha reportado que su implementación ha sido satisfactoria en diversas aplicaciones de ingeniería. Asimismo, se suma la experiencia previa con la que se contaba en el grupo de investigación en su uso (Montain y col., 2015), a su flexibilidad para hibridizarlo con otras técnicas y a su relativa sencillez de programación.

Sin embargo, el PSO estándar cuenta con dos grandes debilidades: la inexistencia de un tratamiento universal de las restricciones y la numerosa cantidad de evaluaciones de las funciones necesarias, habitualmente, para encontrar una solución aceptable.

La primera desventaja se abordó en esta tesis mediante la incorporación de una metodología basada en la medida TAV para manejar las restricciones de igualdad y desigualdad. De esta manera, el PSO puede aplicarse a una mayor variedad de problemas realistas, diferenciándose de la mayoría de las implementaciones previas de metaheurísticas de enjambre, que abordan principalmente problemas cuyas funciones objetivo se encuentran restringidas a la caja.

La segunda debilidad ocasiona que los tiempos de cómputo de este algoritmo sean prolongados, especialmente para modelos de gran escala. Para acelerar las ejecuciones y sacar provecho del paralelismo implícito que posee el algoritmo se programó una versión que corre en GPU para aprovechar estos dispositivos relativamente económicos y ampliamente disponibles en las PC modernas.

Con el objeto de validar el desarrollo, se testeó el PSO en sus versiones serie y paralelo aplicándolo, en una primera instancia, a distintos modelos benchmark ampliamente utilizados en estudios de optimización, que poseen diferente complejidad y cuya solución es conocida. Estas evaluaciones permitieron determinar que el PSO propuesto logra un excelente rendimiento, en términos de Speed-Up y tasas de factibilidad.

En cuanto a la tasa de optimalidad, que tiene que ver con la identificación de soluciones óptimas globales, los resultados muestran un comportamiento diverso que depende del problema específico. En la mayoría de los casos, sin embargo, se identificaron buenas soluciones subóptimas incluso en las instancias más difíciles.

A continuación, se aplicó el optimizador a una familia de modelos de mayor dificultad, como lo son los problemas de pooling de la familia Haverly, considerablemente estudiados en la literatura de optimización no lineal. Las tasas de factibilidad y optimalidad encontradas en estos casos resultaron ser bajas, por lo que se implementó un PSO con reducción secuencial de caja.

Dicha versión del algoritmo realiza varias búsquedas consecutivas en regiones gradualmente más pequeñas, logrando que los resultados alcanzados mejoraran. En líneas generales se logran soluciones factibles, aunque respecto a la optimalidad, se suele converger a soluciones sub-óptimas. A pesar de que estos problemas son de pequeña escala, se los considera desafiantes debido a las bilinealidades y los múltiples óptimos locales que presentan, además de contar con pequeñas regiones factibles no convexas.

Finalmente, se aplicó el PSO a cuatro problemas de interés para la ingeniería química, que se diferencian de los anteriores por poseer una mayor cantidad de variables y restricciones de igualdad. En este caso, la optimización se llevó a cabo previa reducción de sus dimensiones a través de las restricciones de igualdad. Se prefirió esta estrategia, en lugar de emplear una rutina de resolución de sistemas de ecuaciones no lineales, dado que implementar una de estas técnicas imposibilitaría su paralelización directa en la GPU.

Los resultados de ejecutar el PSO para resolver estos problemas más desafiantes de la manera propuesta arrojaron soluciones que, en la gran mayoría de los casos, fueron factibles, aunque bastante apartados del óptimo global (en términos de errores relativos). Nuevamente, para mejorar la calidad de las soluciones, se ejecutó el PSO con reducción secuencial de caja, comprobando de nuevo el aumento en la eficiencia del algoritmo respecto de la tasa de factibilidad y una reducción en el error relativo respecto del global.

En resumen, podemos decir que las contribuciones principales de la tesis son: i) un algoritmo de optimización no lineal genérico basado en PSO ampliado con una estrategia de manejo de restricciones y ii) una versión del algoritmo del ítem anterior que puede ejecutarse sobre GPU, lo que acelera significativamente las optimizaciones según pudo ser comprobado experimentalmente. Por ejemplo, para los problemas benchmark y de la familia Haverly, el Speed-Up del Sistema 1 fue del orden de 100x, mientras que en el caso del Sistema 2 fue del orden de 200x. En cuanto a los modelos más complejos ejecutados en el Sistema 3, se alcanzaron aceleraciones que van desde los 55x hasta los 315x.

Más allá de la utilidad práctica de estas herramientas, la versión ii) del algoritmo entraña cierto aporte a la disciplina de HPC (High Performance Computing) dado que constituye, de acuerdo a nuestro conocimiento, uno de los primeros algoritmos metaheurísticos para GPU que implementa una estrategia explícita de manejo de restricciones (Damiani y col., 2019).

Cabe aclarar que la implementación GPU mono-bloque propuesta restringe el tamaño de problemas a resolver dado que el número de partículas y variables está limitado por el número de hilos por bloque que posea la GPU que se esté utilizando (1024 hilos en las GPUs empleadas en este trabajo). Si bien esto puede considerarse restrictivo, si el modelo posee relativamente pocos grados de libertad se pueden abordar problemas de tamaño considerable si se trabaja en el espacio reducido. Además, a pesar de que todo el enjambre se ejecutó en un solo bloque, haciendo un uso muy limitado de la placa, se lograron muy buenos Speed-Ups.

Desde el punto de vista de la calidad de las soluciones logradas por el algoritmo para el conjunto de problemas estudiados, se encontró que las tasas de factibilidad fueron en general muy buenas, aunque en la mayoría de los casos las soluciones halladas fueron sub-óptimas respecto del global. En este sentido, la herramienta desarrollada no está a la altura de solvers de optimización global comerciales como por ejemplo BARON, el cual, para problemas pequeños y medianos converge siempre a la solución global. Comparado con solvers locales (MINOS, CONOPT) el desempeño fue más cercano dado que el éxito de estos depende fuertemente del punto inicial del algoritmo, conduciendo al óptimo global en algunos casos y a óptimos locales o soluciones no-factibles en otros.

Desde un punto de vista teórico, dado que el teorema de optimización “no free lunch” enuncia que no existe una estrategia de optimización universal de propósito general efectiva para todos los problemas posibles, se considera que los resultados obtenidos son satisfactorios, ya que se utilizó una única parametrización estándar del PSO para todos los experimentos. Es posible que con parametrizaciones alternativas puedan mejorarse las métricas de desempeño de algunos de los modelos estudiados.

Por otra parte, evidencia experimental de la literatura, así como nuestra propia experiencia personal, sugiere que modelos no lineales de mediana escala, aún con un número modesto de no-convexidades, representan un desafío importante aún para los solvers comerciales más sofisticados, a los que se deben proporcionar muy buenos puntos iniciales, adecuados acotamientos y escalados de variables y ecuaciones para lograr soluciones factibles. En otras palabras, contar con las herramientas comerciales

más desarrolladas no evita, en muchos casos, importantes esfuerzos de programación y reformulaciones adicionales para obtener desempeños satisfactorios de los métodos numéricos.

Como conclusión global, obtenida de los experimentos realizados, tanto en la etapa de testeo como en su uso en aplicaciones más desafiantes, se puede afirmar que el PSO desarrollado constituye una herramienta que proporciona soluciones de aceptable calidad en modelos más o menos complejos. Desde un punto de vista práctico, para muchas de las aplicaciones de ingeniería de procesos, caracterizadas por un gran nivel de incertidumbre, soluciones factibles subóptimas suelen ser aceptables desde el punto de vista ingenieril y son, muchas veces, superiores a las que se implementan artesanalmente en la práctica por parte de los operadores de los sistemas industriales. Asimismo, las soluciones sub-óptimas suelen constituir buenos puntos de partida para los algoritmos de búsqueda local.

En términos generales, la experiencia es alentadora para emprender nuevos desarrollos que permitan perfeccionar la herramienta, así como abordar aplicaciones de interés práctico. Sin embargo, en lo que respecta a la plataforma elegida para paralelizar, PyCUDA no es una herramienta tan amigable para utilizar la GPU desde Python, ya que no sólo se tuvo que aprender a programar con este lenguaje, sino que también se debió hacer uso de C. Por estas razones, una recomendación sería trabajar directamente en C y CUDA, a fin de sólo manejar un único lenguaje y aprovechar las directivas que se encuentran más desarrolladas en CUDA que en PyCUDA.

Todo el desarrollo realizado en esta tesis, se puede encontrar en el siguiente link al repositorio de Bitbucket: https://bitbucket.org/luciadamiani/tesismipp/

6.2 Trabajo Futuro

En el transcurso de esta tesis se identificaron varios frentes de investigación con un interesante potencial de desarrollo que podrían mejorar la implementación propia y ampliar su espectro de utilización.

Por una parte, el algoritmo PSO admite varias sofisticaciones que permitirían aumentar las tasas de factibilidad y de optimalidad y mejorar el rendimiento general para algunos problemas. A continuación, se lista una serie de posibles modificaciones/ampliaciones al algoritmo que han sido propuestas por diversos investigadores:

- Definir un peso de inercia (w) dinámico (Marini y Walczak, 2015).

- Realizar las modificaciones propuestas por Ali y Kaelo (2008), tales como modificar los valores de c_1 y c_2 para cada partícula en cada iteración según si alcanza cierto grado de mejora del mejor local; o la actualización de la posición según el valor de un nuevo factor que tiene en cuenta el mejor local y números aleatorios; o establecer valores aleatorios para los *m* peores locales encontrados.

- Incorporar al PSO las distintas maneras de calcular el valor de relajación de restricciones (μ), tales como las propuestas por Zhang y Rangaiah (2012).

- Adoptar otros criterios de terminación para combinarlos con el implementado, tales como finalizar la ejecución cuando se alcance un tiempo máximo o cuando no existe una mejoría notoria en la solución.

- Hibridizar el PSO con un algoritmo de búsqueda local que mejore aún más la solución obtenida, aumentando la posibilidad de alcanzar el óptimo global.

Otra escala de desarrollo tiene que ver con la naturaleza mixto-entera y multi-objetivo de la gran mayoría de las aplicaciones realistas de la ingeniería. Sumadas a la no linealidad, estas características hacen que la exploración del espacio de búsqueda sea una actividad muy exigente desde el punto de vista computacional y desafían las estrategias de aceleración de los algoritmos metaheurísticos.

Las variables enteras, en particular binarias, permiten representar decisiones discretas. Si bien es posible emplear un solver NLP para modelar este tipo de decisiones aplicando

una reformulación continua de las variables binarias $(y \epsilon \{0,1\} \rightarrow 0 \leq y \leq 1; y(1-y) \leq 0)$, la misma introduce una gran cantidad de restricciones y no convexidades difíciles de resolver. Por esta razón, se han propuesto diversas técnicas de manejo de variables binarias en algoritmos metaheurísticos (Crawford y col., 2017), como por ejemplo la técnica "Angle Modulation" propuesta por Pampara y col. (2005), la cual se pretende incluir en versiones futuras del algoritmo.

Otro desafío es el manejo de múltiples objetivos. El mayor inconveniente de este tipo de optimización es que no existe una única solución que resulte óptima para todos los objetivos, ya que en general se encuentran en conflicto entre sí, es decir, que la mejora de uno de ellos puede dar lugar al empeoramiento de otro. Existen muchos métodos para resolver problemas multi-objetivo (Ngatchou y col., 2005) tales como los que se basan en la eficiencia de la curva de Pareto, aquellos que combinan todos los objetivos en uno solo y los que le asignan distintas prioridades a cada objetivo, entre otros. La flexibilidad que posee el PSO para hibridizarse permitiría incluir alguna de estas técnicas sin demasiadas complicaciones, aumentando aún más el rango de problemas que podrían resolverse con esta herramienta. Particularmente, atrajo la atención la propuesta de Zhan y col. (2013) quienes utilizan múltiples poblaciones para múltiples objetivos. En vez de optimizar todos los objetivos juntos con una sola población, cada enjambre se encarga de optimizar un solo objetivo a la vez, para luego, interactuar entre las poblaciones y cooperar a fin de mejorar todos los objetivos. Esta técnica resulta interesante también porque se podría aprovechar mejor los recursos de la GPU, ejecutando las distintas poblaciones en paralelo en cada bloque.

En lo que respecta a la evaluación del desempeño de la versión acelerada del algoritmo y el cálculo del Speed-Up, Tan y Ding (2016) proponen que se debería tomar como tiempo de cómputo del PSO en serie aquel correspondiente al algoritmo programado en paralelo en los núcleos de la CPU. Cabe aclarar que si se paralelizara perfectamente la CPU, el tiempo serie del PSO se podría reducir en un factor de 8 ($t_{CPU\ multinúcleo} = t_{CPU\ mononúcleo}/8$) dado que ese es el número de núcleos de las CPU utilizadas, lo que provocaría que el Speed-Up obtenido se reduzca en un orden de magnitud, con lo cual seguiría siendo mejor el tiempo logrado al ejecutar el PSO en la GPU que en la CPU

multinúcleo. Igualmente, de manera ideal se debería paralelizar en los núcleos de la CPU para efectuar una comparación más justa entre los tiempos de ejecución del optimizador en la CPU y en la GPU. Cabe aclarar que la paralelización de la CPU requeriría la programación en otro lenguaje llamado OpenCL.

En lo concerniente a la paralelización sobre la GPU propiamente dicha, una extensión natural del trabajo realizado sería la implementación del PSO multi-bloque. A fines de 2018 NVIDIA introdujo los "grupos cooperativos", que funcionan a partir de la versión 10 de CUDA, y proporcionan un modelo más flexible para la sincronización y la comunicación entre los distintos bloques que ejecutan los núcleos. Si en un futuro, esta nueva herramienta fuera compatible con PyCUDA, podría crearse un PSO multi-bloque que asigne una partícula a un bloque y emplee enjambres con un tamaño máximo igual a la cantidad de bloques que posea la GPU. Este desarrollo permitiría resolver problemas con una mayor cantidad de partículas y variables, ya que no se encontraría limitado por el número de hilos por bloque que posee el PSO paralelizado presentado en esta tesis.

Otra opción para no perder las ventajas de programar en Python sería, por ejemplo, utilizar Numba[1] como acelerador. Éste es un compilador de Python de alto rendimiento "just in time" que traduce las funciones de Python a código optimizado en tiempo de ejecución utilizando una biblioteca de compiladores estándar. El código fuente de los algoritmos numéricos compilados por Numba sigue siendo Python puro, pero pueden acercarse a las velocidades de C. Esta opción parece la más promisoria para mantener un ambiente de programación muy productivo que permite aprovechar los recursos multinúcleo de nuestro sistema.

[1] www.numba.pydata.org

Referencias

Adams, T. A., & Seider, W. D. (2008). Semicontinuous distillation for ethyl lactate production. AIChE journal, 54(10), 2539-2552.

Adhya, N., Tawarmalani, M., & Sahinidis, N. V. (1999). A Lagrangian approach to the pooling problem. Industrial & Engineering Chemistry Research, 38(5), 1956-1972.

Adjiman, C. S., Androulakis, I. P., & Floudas, C. A. (1998). A global optimization method αBB, for general twice-differentiable NLPs – II. Implementation and computational results. Computers and Chemical Engineering, 22, 1159–1179.

Aggarwal, A., & Floudas, C. A. (1990). Synthesis of general distillation sequences—nonsharp separations. Computers & Chemical Engineering, 14(6), 631-653.

Alba, E., Luque, G., & Nesmachnow, S. (2013). Parallel metaheuristics: recent advances and new trends. International Transactions in Operational Research, 20(1), 1-48.

Ali, M. M., & Kaelo, P. (2008). Improved particle swarm algorithms for global optimization. Applied mathematics and computation, 196(2), 578-593.

Andrei, N. (2013). Nonlinear optimization applications using the GAMS technology. New York: Springer.

Armas, S., Mena, L., Samarín, A., Blanco, V., Morales, A., & Almeida, F. (2011). Memoria TAD: Experiencias con Python y CUDA en Computación de Altas Prestaciones.

Bansal, J. C., Singh, P. K., Saraswat, M., Verma, A., Jadon, S. S., & Abraham, A. (2011). Inertia weight strategies in particle swarm optimization. In Nature and Biologically Inspired Computing (NaBIC), 2011 Third World Congress on (pp. 633-640). IEEE.

Boussaïd, I., Lepagnot, J., & Siarry, P. (2013). A survey on optimization metaheuristics. Information Sciences, 237, 82-117.

Bunus, P., & Fritzson, P. (2002). A debugging scheme for declarative equation-based modeling languages. In International Symposium on Practical Aspects of Declarative Languages (pp. 280-298). Springer, Berlin, Heidelberg.

Calazan, R. M., Nedjah, N., & de Macedo Mourelle, L. (2013). Three alternatives for parallel GPU-based implementations of high performance particle swarm optimization. In International Work-Conference on Artificial Neural Networks (pp. 241-252). Springer, Berlin, Heidelberg.

Chen, T. Y., & Chi, T. M. (2010). On the improvements of the particle swarm optimization algorithm. Advances in Engineering Software, 41(2), 229-239.

Chen, Y. W., Wang, L. C., Wang, A., & Chen, T. L. (2017). A particle swarm approach for optimizing a multi-stage closed loop supply chain for the solar cell industry. Robotics and Computer-Integrated Manufacturing, 43, 111-123.

Coello, C. A. C. (2002). Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. Computer methods in applied mechanics and engineering, 191(11), 1245-1287.

Conejo, A. J., Castillo, E., Minguez, R., & Garcia-Bertrand, R. (2006). Decomposition techniques in mathematical programming: engineering and science applications. Springer Science & Business Media.

Crawford, B., Soto, R., Astorga, G., García, J., Castro, C., & Paredes, F. (2017). Putting continuous metaheuristics to work in binary search spaces. Complexity, 2017.

Damiani, L., Diaz, A. I., Iparraguirre, J., & Blanco, A. M (2019). Accelerated particle swarm optimization with explicit consideration of model constraints. Cluster Computing, 1-16.

Engelbrecht, A. P., & Cleghorn, C. W. (2018). Particle swarm optimization: a guide to effective, misconception free, real world use. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (pp. 831-857). ACM.

Erbeyoğlu, G., & Bilge, Ü. (2016). PSO-based and SA-based metaheuristics for bilinear programming problems: an application to the pooling problem. Journal of Heuristics, 22(2), 147-179.

Floudas, C. A., Pardalos, P. M., Adjiman, C. S., Esposito, W. R., Gumus¸, Z. H., Harding, S. T., Klepeis, J. L., Meyer, C. A., & Schweiger, C. A. (1999). Handbook of test problems in local and global optimization. Dordrecht: Kluwer Academic.

Gunnerud, V., Foss, B. A., McKinnon, K. I. M., Nygreen, B. (2012). Oil production optimization solved by piecewise linearization in a Branch & Price framework. Computers & Operations Research, 39(11), 2469-2477.

Ho, Y. C., & Pepyne, D. L. (2002). Simple explanation of the no-free-lunch theorem and its implications. Journal of optimization theory and applications, 115(3), 549-570.

Hung, Y., & Wang, W. (2012). Accelerating parallel particle swarm optimization via GPU. Optimization Methods and Software, 27(1), 33-51.

Jain, B. J., Pohlheim, H., & Wegener, J. (2001). On termination criteria of evolutionary algorithms. In Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001, San Francisco, CA, Morgan Kaufmann Publishers.

Kayhan, A. H., Ceylan, H., Ayvaz, M. T., & Gurarslan, G. (2010). PSOLVER: A new hybrid particle swarm optimization algorithm for solving continuous optimization problems. Expert Systems with Applications, 37(10), 6798-6808.

Kennedy, J., Eberhart, R. (1995). Particle swarm optimization (PSO). In Proc. IEEE International Conference on Neural Networks, Perth, Australia (pp. 1942-1948).

Kennedy, J. (1999). Small worlds and mega-minds: effects of neighborhood topology on particle swarm performance. In Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on (Vol. 3, pp. 1931-1938). IEEE.

Kołodziejczyk, J., Sychel, D., & Bera, A. (2017). Improved CUDA PSO Based on Global Topology. In International Conference on Artificial Intelligence and Soft Computing (pp. 347-358). Springer, Cham.

Laguna-Sánchez, G. A., Olguín-Carbajal, M., Cruz-Cortés, N., Barrón-Fernández, R., & Álvarez-Cedillo, J. A. (2009). Comparative study of parallel variants for a particle swarm optimization algorithm implemented on a multithreading GPU. Journal of applied research and technology, 7(3), 292-307.

Lang, J., & Zhao, J. (2016). Modeling and optimization for oil well production scheduling. Chinese Journal of Chemical Engineering, 24(10), 1423-1430.

Liang, J. J., Runarsson, T. P., Mezura-Montes, E., Clerc, M., Suganthan, P. N., Coello, C. C., & Deb, K. (2006). Problem definitions and evaluation criteria for the CEC 2006 special session on constrained real-parameter optimization. Journal of Applied Mechanics, 41(8).

Marini, F., & Walczak, B. (2015). Particle swarm optimization (PSO). A tutorial. Chemometrics and Intelligent Laboratory Systems, 149, 153-165.

Montain, M. E., Blanco, A. M., & Bandoni, J. A. (2015). Optimal drug infusion profiles using a particle swarm optimization algorithm. Computers & Chemical Engineering, 82, 13-24.

Murtagh, B. A., & Saunders, M. A. (1995). Minos 5.4 user's guide (revised). Technical Report SOL 83-20R, Department of Operations Research, Stanford University, Stanford, CA 94305, USA, 1993. Revised.

Mussi, L., Daolio, F., & Cagnoni, S. (2011). Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture. Information Sciences, 181(20), 4642-4657.

Ngatchou, P., Zarei, A., & El-Sharkawi, A. (2005). Pareto multi objective optimization. In Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems (pp. 84-91). IEEE.

Nickolls, J., & Dally, W. J. (2010). The GPU computing era. IEEE micro, 30(2).

Pampara, G., Franken, N., & Engelbrecht, A. P. (2005). Combining particle swarm optimisation with angle modulation to solve binary problems. In Evolutionary Computation, 2005. The 2005 IEEE Congress on (Vol. 1, pp. 89-96). IEEE.

Pant, M., Thangaraj, R., & Singh, V. P. (2009a). Optimization of mechanical design problems using improved differential evolution algorithm. International Journal of Recent Trends in Engineering, 1(5), 21–25.

Pant, M., Thangaraj, R., & Singh, V. P. (2009b). Particle swarm optimization with crossover operator and its engineering applications. IAENG International Journal of Computer Science, 36(2), 112–121.

Papadrakakis, M., Stavroulakis, G., & Karatarakis, A. (2011). A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures. Computer Methods in Applied Mechanics and Engineering, 200(13-16), 1490-1508.

Patterson, D. (2010). The trouble with multi-core. IEEE Spectrum, 47(7), 28-32.

Paul H., Tay. (1987) Optimal design of an industrial refrigeration system. In Proceedings of International Conference on Optimization Techniques and Applications (pp. 427–435), Singapore: National University of Singapore, 8–10 April 1987.

Pfetsch, M. E., Fügenschuh, A., Geißler, B., Geißler, N., Gollmer, R., Hiller, B., ... & Morsi, A. (2015). Validation of nominations in gas network optimization: models, methods, and solutions. Optimization Methods and Software, 30(1), 15-53.

Rangaiah, G. P. (2010). Stochastic global optimization: techniques and applications in chemical engineering. World Scientific.

Roberge, V., & Tarbouchi, M. (2012). Parallel particle swarm optimization on graphical processing unit for pose estimation. WSEAS Trans. Comput, 11(6), 170-179.

Shokrian, M., & High, K. A. (2014). Application of a multi objective multi-leader particle swarm optimization algorithm on NLP and MINLP problems. Computers & Chemical Engineering, 60, 57-75.

Souza, D., Teixeira, O., Monteiro, D., & de Oliveira, R. L. (2012). A CUDA-Based Cooperative Evolutionary Multi-Swarm Optimization Appliedto Engineering Problems. In Proc. of the XXXII Congress of the Brazilian Computing Society.

Tabkhi, F., Pibouleau, L., Hernandez-Rodriguez, G., Azzaro-Pantel, C., & Domenech, S. (2010). Improving the performance of natural gas pipeline networks fuel consumption minimization problems. AIChE journal, 56(4), 946-964.

Tan, Y., & Ding, K. (2016). A survey on GPU-based implementation of swarm intelligence algorithms. IEEE transactions on cybernetics, 46(9), 2028-2041.

Tawarmalani, M., & Sahinidis, N. V. (2002). Convexification and global optimization in continuous and mixed-integer nonlinear programming: theory, algorithms, software, and applications (Vol. 65). Springer Science & Business Media.

Visweswaran, V., & Floudas, C. A. (1996). Computational results for an efficient implementation of the GOP algorithm and its variants. In Global Optimization in Engineering Design (pp. 111-153). Springer, Boston, MA.

Yiqing, L., Xigang, Y., & Yongjian, L. (2007). An improved PSO algorithm for solving non-convex NLP/MINLP problems with equality constraints. Computers & chemical engineering, 31(3), 153-162.

Zhang, H., & Rangaiah, G. P. (2012). An efficient constraint handling method with integrated differential evolution for numerical and engineering optimization. Computers & Chemical Engineering, 37, 74-88.

Zhan, Z. H., Li, J., Cao, J., Zhang, J., Chung, H. S. H., & Shi, Y. H. (2013). Multiple populations for multiple objectives: A coevolutionary technique for solving multiobjective optimization problems. *IEEE Transactions on Cybernetics*, *43*(2), 445-463.

Zhou, Y., & Tan, Y. (2009). GPU-based parallel particle swarm optimization. In Evolutionary Computation, 2009. CEC'09. IEEE Congress on (pp. 1493-1500). IEEE.

Anexo A

Modelo U1: Función Branin RCOS

$$Minimizar f(x) = a(x_2 - bx_1^2 + cx_1 - d)^2 + e(1-f)\big(cos(x_1)\big) + e$$

$$Dondea = 1, b = 5.1/(4\pi^2), c = 5/\pi, d = 6, f = 1/(8\pi)$$

$$Límites: -5 \le x_1 \le 10, 0 \le x_2 \le 15$$

$Mínimoglobal: f(x\ ) =$0.397887

$x\ =$(0, 0)

Modelo U2: Función Shubert

$$Minimizar f(x) = \left(\sum_{j=1}^{5} jcos\big((j+1)x_1 + j\big)\right)\left(\sum_{j=1}^{5} jcos\ \big((j+1)x_2 + j\big)\right)$$

$$Límites: -10 \le x_j \le 10 (j = 1,2)$$

$$Mínimoglobal: f(x\ ) = -186$$

$$x\ = (0,0)$$

Modelo U3: Función Michalewicz

$$Minimizar f(x) = -\sum_{j=1}^{2} sin(x_j)\left(sin\big(jx_j^2/\pi\big)\right)^{2m}; m = 10$$

$$Límites: 0 \le x_j \le \pi (j = 1,2)$$

$$Mínimoglobal: f(x\ ) = -1.8013$$

$$x\ = (2.20, 1.57)$$

Modelo U4: Función Colville

$$Minimizar f(x) = 100(x_1^2 - x_2)^2 + (x_1 - 1)^2 + (x_3 - 1)^2 + 90(x_3^2 - x_4)^2 + 10.1[(x_2 - 1)^2 + (x_4 - 1)^2]$$
$$+19.8(x_2 - 1)(x_4 - 1)$$

$$Límites: -10 \le x_j \le 10 (j = 1, \dots, 4)$$

$$Mínimoglobal: f(x\ ) = 0$$

$$x\ = (1,1,1,1)$$

Modelo U5: Función Spherical

$$Minimizar f(x) = \sum_{j=1}^{D} x_j^2; D = 10$$

$$Límites: -100 \le x_j \le 100 (j = 1, \dots, 10)$$

$$Mínimoglobal: f(x\ ) = 0$$

$$x\ = (0,0,0,0,0,0,0,0,0,0)$$

Modelo U6: Función Quadric

$$Minimizar f(x) = \sum_{i=1}^{D}\left(\sum_{j=1}^{i} x_j\right)^2 ; D = 10$$

$$Límites: -100 \leq x_j \leq 100 (j = 1, \dots, 10)$$

$$Mínimoglobal: f(x\ ) = 0$$

$$x\ = (0,0,0,0,0,0,0,0,0,0)$$

Modelo U7: Función Rosenbrock

$$Minimizar f(x) = \sum_{j=1}^{D-1}\left(100\left(x_j^2 - x_{j+1}\right)^2 + \left(x_j - 1\right)^2\right); D = 10$$

$$Límites: -5 \leq x_j \leq 10 (j = 1, \dots, 10)$$

$$Mínimoglobal: f(x\ ) = 0$$

$$x\ = (1,1,1,1,1,1,1,1,1,1)$$

Modelo U8: Función Griewank

$$Minimizar f(x) = 1 + \frac{1}{4000}\sum_{j=1}^{D} x_j^2 - \prod_{j=1}^{D} cos\left(\frac{x_j}{\sqrt{j}}\right); D = 10$$

$$Límites: -600 \leq x_j \leq 600 (j = 1, \dots, 10)$$

$Mínimoglobal: f(x\ ) = 0$

$$x\ = (0,0,0,0,0,0,0,0,0,0)$$

Modelo U9: Función Rastrigin

$$Minimizar f(x) = 10D + \sum_{j=1}^{D}\left[x_j^2 - 10cos\left(2\pi x_j\right)\right]$$

$$Límites: -5.12 \leq x_j \leq 5.12 (j = 1, \dots, 10)$$

$$Mínimoglobal: f(x\ ) = 0$$

$$x\ = (0,0,0,0,0,0,0,0,0,0)$$

Modelo U10: Función Bukin

$$Minimizar f(x) = 100\sqrt{|x_2 - 0.01x_1^2|} + 0.01|x_1 + 10|$$

$$Límites: -15 \leq x_1 \leq 5, -3 \leq x_2 \leq 3$$

$$Mínimoglobal: f(x\ ) = 0$$

$$x \quad = (-10,1)$$

Modelo U11: Función Schwefel

$$Minimizar f(x) = 100\sqrt{|x_2 - 0.01x_1^2|} + 0.01|x_1 + 10|$$

$$Límites: -10 \leq x_j \leq 10 (j = 1, \dots, 10)$$

$$Mínimo global: f(x \quad) = 0$$

$$x \quad = (0,0,0,0,0,0,0,0,0,0)$$

Modelo U12: Función Step

$$Minimizar f(x) = 100\sqrt{|x_2 - 0.01x_1^2|} + 0.01|x_1 + 10|$$

$$Límites: -10 \leq x_j \leq 10 (j = 1, \dots, 10)$$

$$Mínimo global: f(x \quad) = 0$$

$$x \quad = (-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5)$$

Modelo C1:

$$Minimizar f(x) = 5\sum_{i=1}^{4} x_i - 5\sum_{i=1}^{4} x_i^2 - \sum_{i=5}^{13} x_i$$

Sujeto a:

$$g_1(x) = 2x_1 + 2x_2 + x_{10} + x_{11} - 10 \leq 0$$
$$g_2(x) = 2x_1 + 2x_3 + x_{10} + x_{12} - 10 \leq 0$$
$$g_3(x) = 2x_2 + 2x_3 + x_{11} + x_{12} - 10 \leq 0$$
$$g_4(x) = -8x_1 + x_{10} \leq 0$$
$$g_5(x) = -8x_2 + x_{11} \leq 0$$
$$g_6(x) = -8x_3 + x_{12} \leq 0$$
$$g_7(x) = -2x_4 - x_5 + x_{10} \leq 0$$
$$g_8(x) = -2x_6 - x_7 + x_{11} \leq 0$$
$$g_9(x) = -2x_8 - x_9 + x_{12} \leq 0$$

$$Límites: 0 \leq x_j \leq 1 (j = 1, \dots, 9), 0 \leq x_j \leq 100$$

$$(j = 10,11,12) \wedge 0 \leq x_{13} \leq 1$$

$$Mínimo global: f(x \quad) = -15.$$

$$x \quad = (1,1,1,1,1,1,1,1,1,3,3,3,1)$$

Modelo C2:

$$Minimizar f(x) = -\left(\sqrt{D}\right)^D \prod_{j=1}^{D} x_j$$

Sujeto a:

$$h_1(x) = \sum_{j=1}^{D} x_j^2 - 1 = 0$$

$$Límites: 0 \leq x_j \leq 1 (j = 1, \dots, 10).$$

$$Mínimoglobal: f(x\ ) = -1.00050010001000.$$

$$x\ = (0.31624357647283069, 0.31624357647283069, 0.31624357647283069, 0.31624357647283069, 0.31624357647283069, 0.31624357647283069, 0.31624357647283069, 0.31624357647283069, 0.31624357647283069, 0.31624357647283069)$$

Modelo C3:

$$Minimizar f(x) = 5.3578547x_3^2 + 0.000001x_1^3 + 2x_2 + (0.000002/3)x_2^3$$

Sujeto a:

$$g_1(x) = 85.334407 + 0.0056858x_2x_5 + 0.0006262x_1x_4 - 0.0022053x_3x_5 - 92 \leq 0$$

$$g_2(x) = -85.334407 - 0.0056858x_2x_5 - 0.0006262x_1x_4 + 0.0022053x_3x_5 \leq 0$$

$$g_3(x) = 80.51249 + 0.0071317x_2x_5 + 0.0029955x_1x_2 - 0.0021813x_3^2 - 110 \leq 0$$

$$g_4(x) = -80.51249 - 0.0071317x_2x_5 - 0.0029955x_1x_2 - 0.0021813x_3^2 + 90 \leq 0$$

$$g_5(x) = 9.300961 + 0.0047026x_3x_5 + 0.0012547x_1x_3 + 0.0019085x_3x_4 - 25 \leq 0$$

$$g_6(x) = -9.300961 - 0.0047026x_3x_5 - 0.0012547x_1x_3 - 0.0019085x_3x_4 + 20 \leq 0$$

$$Límites: 78 \leq x_1 \leq 102, 33 \leq x_2 \leq 45 \wedge 27 \leq x_j \leq 45 (j = 3,4,5).$$

$$Mínimoglobal: f(x\ ) = -30665.53867178332.$$

$$x\ = (78, 33, 29.9952560256815985, 45, 36.7758129057882073)$$

Modelo C4:

$$Minimizar f(x) = 3x_1 + 0.000001x_1^3 + 2x_2 + (0.000002/3)x_2^3$$

Sujeto a:

$$g_1(x) = -x_4 + x_3 - 0.55 \leq 0$$

$$g_2(x) = -x_3 + x_4 - 0.55 \leq 0$$

$$h_1(x) = 1000sin(-x_3 - 0.25) + 1000sin(-x_4 - 0.25) + 894.8 - x_1 = 0$$

$$h_2(x) = 1000sin(x_3 - 0.25) + 1000sin(x_3 - x_4 - 0.25) + 894.8 - x_2 = 0$$

$$h_3(x) = 1000sin(x_4 - 0.25) + 1000sin(x_4 - x_3 - 0.25) + 1294.8 = 0$$

$$Límites: 0 \leq x_1 \leq 1200, 0 \leq x_2 \leq 1200, -0.55 \leq x_3 \leq 0.55 \wedge -0.55 \leq x_4 \leq 0.55$$

$$Mínimoglobal: f(x\ ) = 5126.4967140071.$$

$$x\ = (679.9451482970287, 1026.0669760000469, 0.11887636909441043, -0.3962334852151782)$$

Modelo C5:

$$Minimizar f(x) = (x_1 - 10)^3 + (x_2 - 20)^3$$

Sujeto a:

$$g_1(x) = -(x_1 - 5)^2 - (x_2 - 5)^2 + 100 \leq 0$$

$$g_2(x) = (x_1 - 6)^2 + (x_2 - 5)^2 - 82.81 \leq 0$$

$$Límites: 13 \leq x_1 \leq 100 \wedge 0 \leq x_2 \leq 100.$$

$$Mínimoglobal: f(x\ ) = -6961.81387558015.$$

$$x\ = (14.09500000000000064, 0.8429607892154795668)$$

Modelo C6:

$$Minimizar f(x) = x_1^2 + x_2^2 + x_1 x_2 - 14x_1 - 16x_2 + (x_3 - 10)^2 + 4(x_4 - 5)^2 + (x_5 - 3)^2$$
$$+2(x_6 - 1)^2 + 5x_7^2 + 7(x_8 - 11)^2 + 2(x_9 - 10)^2 + (x_{10} - 7)^2 + 45$$

Sujeto a:

$$g_1(x) = -105 + 4x_1 + 5x_2 - 3x_7 + 9x_8 \leq 0$$

$$g_2(x) = 10x_1 - 8x_2 - 17x_7 + 2x_8 \leq 0$$

$$g_3(x) = -8x_1 + 2x_2 + 5x_9 - 2x_{10} - 12 \leq 0$$

$$g_4(x) = 3(x_1 - 2)^2 + 4(x_2 - 3)^2 + 2x_3^2 - 7x_4 - 120 \leq 0$$

$$g_5(x) = 5x_1^2 + 8x_2 + (x_3 - 6)^2 - 2x_4 - 40 \leq 0$$

$$g_6(x) = x_1^2 + 2(x_2 - 2)^2 - 2x_1 x_2 + 14x_5 - 6x_6 \leq 0$$

$$g_7(x) = 0.5(x_1 - 8)^2 + 2(x_2 - 4)^2 + 3x_5^2 - x_6 - 30 \leq 0$$

$$g_8(x) = -3x_1 + 6x_2 + 12(x_9 - 8)^2 - 7x_{10} \leq 0$$

$$Límites: -10 \leq x_j \leq 10 (j = 1, \ldots, 10).$$

$$Mínimoglobal: f(x\ ) = 24.30620906818.$$

$$x\ = 2.17199634142692, 2.3636830416034, 8.77392573913157, 5.09598443745173,$$
$$0.9906547565604931.43057392853463, 1.32164415364306, 9.82872576524495,$$
$$8.2800915887356, 8.3759266477347$$

Modelo C7:

$$Minimizar f(x) = \frac{-sin^3(2\pi x_1) sin(2\pi x_2)}{x_1^3 (x_1 + x_2)}$$

Sujeto a:

$$g_1(x) = x_1^2 - x_2 + 1 \leq 0$$

$$g_2(x) = 1 - x_1 + (x_2 - 4)^2 \leq 0$$

$$Límites: 0 \leq x_j \leq 1 (j = 1, \ldots, 10).$$

$$Mínimoglobal: f(x\ ) = -0.0958250414180359.$$

$$x\ = (1.22797135260752599, 4.24537336612274885)$$

Modelo C8:

$$Minimizar: f(x) = (x_1 - 10)^2 + 5(x_2 - 12)^2 + x_3^4 + 3(x_4 - 11)^2 + 10x_5^6 + 7x_6^2 + x_7^4 - 4x_6 x_7 - 10x_6 - 8x_7$$

Sujeto a:

$$g_1(x) = -127 + 2x_1^2 + 3x_2^4 + x_3 + 4x_4^2 + 5x_5 \leq 0$$

$$g_2(x) = -282+7x_1 + 3x_2 + 10x_3^2 + x_4 - x_5 \leq 0$$
$$g_3(x) = -196 + 23x_1 + x_2^2 + 6x_6^2 - 8x_7 \leq 0$$
$$g_4(x) = 4x_1^1 + x_2^2 - 3x_1x_2 + 2x_3^2 + 5x_6 - 11x_7 \leq 0$$
$$Límites: -10 \leq x_j \leq 10(j = 1, \dots, 7).$$
$$Mínimoglobal: f(x\ ) = 680.630057374402.$$
$$x\ = \begin{matrix} 2.33049935147405174, 1.9137236847114592, -0.477541399510615805, 4.36572624923625874, \\ -0.624486959100388983, 1.03813099410962173, 1.5942266780671519 \end{matrix}$$

Modelo C9

$$Minimizar f(x) = x_1 + x_2 + x_3$$

Sujeto a:

$$g_1(x) = -1 + 0.0025(x_4 + x_6) \leq 0$$
$$g_2(x) = -1 + 0.0025(x_5 + x_7 - x_4) \leq 0$$
$$g_3(x) = -1 + 0.01(x_8 - x_5) \leq 0$$
$$g_4(x) = -x_1x_6 + 833.33252x_4 + 100x_1 - 83333.333 \leq 0$$
$$g_5(x) = -x_2x_7 + 1250x_5 + x_2x_4 - 1250x_4 \leq 0$$
$$g_6(x) = -x_3x_8 + 1250000 + x_3x_5 - 2500x_5 \leq 0$$
$$Límites: 100 \leq x_1 \leq 1000, 1000 \leq x_j \leq 10000(j = 2,3),$$
$$10 \leq x_j \leq 1000(j = 4, \dots, 8).$$
$$Mínimoglobal: f(x\ ) = 7049.24802052867.$$
$$x\ = \begin{matrix} 579.306685017979589, 1359.97067807935605, 5109.97065743133317, 182.01769963061534, \\ 295.601173702746792, 217.982300369384632, 286.41652592786852, 395.601173702746735 \end{matrix}$$

Modelo C10:

$$Minimizar f(x) = x_1^2 + (x_2 - 1)^2$$

Sujeto a:

$$h_1(x) = x_2 - x_1^2 = 0$$
$$Límites: -1 \leq x_1 \leq 1, -1 \leq x_2 \leq 1$$
$$Mínimoglobal: f(x\ ) = 0.7499.$$
$$x\ = (-0.707036070037170616, 0.500000004333606807)$$

Modelo C11:

$$Minimizar f(x) = e^{x_1x_2x_3x_4x_5}$$

Sujeto a:

$$h_1(x) = x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 10 = 0$$
$$h_2(x) = x_2x_3 - 5x_4x_5 = 0$$
$$h_3(x) = x_1^3 + x_2^3 + 1 = 0$$

$$Límites: -2.3 \leq x_j \leq 2.3 (j = 1,2),$$

$$-3.2 \leq x_j \leq 3.2 (j = 3,4,5).$$

$$Mínimoglobal: f(x\ ) = 0.053941514041898.$$

$$x\ = \begin{matrix} 1.71714224003, 1.59572124049468, 1.8272502406271, -0.763659881912867, \\ -0.76365986736498 \end{matrix}$$

Modelo C12:

$$Minimizar f(x) = \sum_{j=1}^{10} x_j \left(c_j + ln \frac{x_j}{\sum_{j=1}^{10} x_j} \right)$$

Sujeto a:

$$h_1(x) = x_1 + 2x_2 + 2x_3 + x_6 + x_{10} - 2 = 0$$

$$h_2(x) = x_4 + 2x_5 + x_6 + x_7 - 1 = 0$$

$$h_3(x) = x_3 + x_7 + x_8 + 2x_9 + x_{10} - 1 = 0$$

$$Límites: 0 \leq x_j \leq 10 (j = 1, \dots, 10).$$

$$Donde c_1 = -6.089, c_2 = -17.164, c_3 = -34.054, c_4 = -5.914, c_5 = -24.721, c_6 = -14.986, c_7 = -24.1,$$

$$c_8 = -10.708, c_9 = -26.662, c_{10} = -22.179$$

$$Mínimoglobal: f(x\ ) = -47.7648884594915.$$

$$x\ = \begin{matrix} 0.0406684113216282, 0.147721240492452, 0.783205732104114, 0.00141433931889084, \\ 0.485293636780388, 0.000693183051556082, 0.0274052040687766, 0.0179509660214818, \\ 0.0373268186859717, 0.0968844604336845. \end{matrix}$$

Modelo C13:

$$Minimizar f(x) = 1000 - x_1^2 - 2x_2^2 - x_3^2 - x_1 x_2 - x_1 x_3$$

Sujeto a:

$$h_1(x) = x_1^2 + x_2^2 + x_3^2 - 25 = 0$$

$$h_2(x) = 8x_1 + 14x_2 + 7x_3 - 56 = 0$$

$$Límites: 0 \leq x_j \leq 10 (j = 1,2,3).$$

$$Mínimoglobal: f(x\ ) = 961.715022289961.$$

$$x\ = (3.51212812611795133, 0.216987510429556135).$$

Modelo C14:

$$\begin{matrix} x \\ (\ \ |1x_4 - x_2 x_3 + x_3 x_9 - x_5 x_9 + x_5 x_8 - x_6 x_7) \\ Minimizar f(x) = -0.5 \end{matrix}$$

Sujeto a:

$$g_1(x) = x_3^2 + x_4^2 - 1 \leq 0$$

$$g_2(x) = x_9^2 - 1 \leq 0$$

$$g_3(x) = x_5^2 + x_6^2 - 1 \leq 0$$

$$g_4(x) = x_1^2 + (x_2 - x_9)^2 - 1 \leq 0$$

$$g_5(x) = (x_1 - x_5)^2 + (x_2 - x_6)^2 - 1 \leq 0$$

$$g_6(x) = (x_1 - x_7)^2 + (x_2 - x_8)^2 - 1 \leq 0$$

$$g_7(x) = (x_3 - x_5)^2 + (x_4 - x_6)^2 - 1 \leq 0$$

$$g_8(x) = (x_3 - x_7)^2 + (x_4 - x_8)^2 - 1 \leq 0$$

$$g_9(x) = x_7^2 (x_8 - x_9)^2 - 1 \leq 0$$

$$g_{10}(x) = x_2 x_3 - x_1 x_4 \leq 0$$

$$g_{11}(x) = -x_3 x_9 \leq 0$$

$$g_{12}(x) = x_5 x_9 \leq 0$$

$$g_{13}(x) = x_6 x_7 - x_5 x_8 \leq 0$$

$$Límites: -10 \leq x_j \leq 10 (j = 1, \ldots, 8), 0 \leq x_9 \leq 20.$$

$$Mínimo global: f(x^*) = -0.866025403784439.$$

$$x^* = (-0.65777619242794316, -0.15341877348243854, 0.323413871675240938,$$

$$-0.946257611651304398, -0.657776194376798906, -0.753213434632691414,$$

$$0.323413874123576972, -0.346462947962331735, 0.59979466285217542)$$

Modelo C15:

$$Minimizar f(x) = -x_1 - x_2$$

Sujeto a:

$$g_1(x) = -2x_1^4 + 8x_1^3 - 8x_1^2 + x_2 - 2 \leq 0$$

$$g_2(x) = -4x_1^4 + 32x_1^3 - 88x_1^2 + 96x_1 + x_2 - 36 \leq 0$$

$$Límites: 0 \leq x_1 \leq 3, 0 \leq x_2 \leq 4.$$

$$Mínimo global: f(x^*) = -5.50801327159536.$$

$$x^* = (2.32952019747762, 3.17849307411774).$$