Facultad de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN INGENIERíA DEL SOFTWARE

Paralelización del algoritmo de optimización de enjambre de partículas usando Spark

Estudiante: Roberto José Núñez Rodríguez
Dirección: Patricia González Gómez
Xoán Carlos Pardo Martínez

A Coruña, xuño de 2020.

A mi familia, a Sara y a todas las personas que me han acompañado en este camino.

Agradecimientos

En primer lugar, me gustaría agradecer especialmente a mis directores Patricia y Xoán, por la oportunidad de desarrollar este trabajo así como toda la ayuda y apoyo que me ofrecieron durante estos meses.

A mis padres y, en general, a toda mi familia, por ayudarme, no solo durante el grado, sino durante toda la vida. A Sara, por acompañarme, entenderme y aguantarme. Tu apoyo durante estos cuatro años ha sido impagable.

A todos ellos, muchas gracias.

Resumen

Muchos problemas de optimización hoy en día, en multitud de áreas como la bioinformática, la logística o la ingeniería, presentan una gran dificultad que requiere el uso de herramientas y métodos eficientes, desde el punto de vista computacional, para alcanzar buenos resultados en tiempos razonables. Para resolverlos se suelen usar métodos de optimización global, y, entre ellos, las metaheurísticas son los preferidos, por su eficiencia resolviendo problemas complejos. Sin embargo, en muchos casos reales las metaheurísticas todavía necesitan mucho tiempo de cálculo para obtener resultados aceptables. La computación de altas prestaciones (HPC) puede ayudar con metaheurísticas paralelas a recortar este tiempo. No obstante, el acceso a recursos HPC no siempre es fácil y la programación paralela con modelos tradicionales requiere experiencia por parte del programador. Hoy en día, el acceso a potentes recursos computacionales es más sencillo a través de las nuevas infraestructuras cloud, que nos ofrecen plataformas como servicio (PaaS). Además, existen nuevos modelos de programación para entornos distribuidos que facilitan la tarea al programador, abaratando y acortando el tiempo de desarrollo del producto. El objetivo de este trabajo es el diseño, implementación y evaluación de estrategias de paralelización para la metaheurística PSO (*Particle Swarm Optimization*) usando el framework Spark.

Abstract

Many optimization problems nowadays, in many areas such as bioinformatics, logistics or engineering, are so challenging that require the use of efficient tools and methods, from the computational point of view, to achieve good results in reasonable times. To solve these problems, global optimization methods are usually used, and among them, metaheuristics are preferred, due to their efficiency in solving complex problems. However, in many real problems, metaheuristics still require a large computing time to obtain acceptable results. High performance computing (HPC) can help, by means of parallel metaheuristics, to cut this computation time. Nevertheless, access to HPC resources is not always easy and parallel programming with traditional models requires programmer expertise. Nowadays, access to powerful computing resources is easier through cloud infrastructures, which offer platforms as a service (PaaS). Further, there are new programming models for distributed environments that facilitate programming by reducing and shortening the time of product development.

The objective of this work is the design, implementation and evaluation of parallel strategies for PSO metaheuristic using the Spark framework.

Palabras clave:

- Optimización global
- Metaheurísticas
- Optimización de enjambre de partículas
- Computación distribuida
- Cloud
- Spark
- Computación de altas prestaciones
- Recursos TI

Keywords:

- Global optimization
- Metaheuristics
- Particle Swarn Optimization
- Distributed computing
- Cloud
- Spark
- High performance computing
- IT resources

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este capítulo introductorio ofrece una breve contextualización del proyecto, indicando los objetivos del mismo, la metodología empleada y el plan de trabajo, y concluyendo con la descripción de la estructura del documento.

1.1 Objetivos del proyecto

Existen muchas áreas en diferentes disciplinas, como la bioinformática o la ingeniería, que presentan problemas de optimización de gran complejidad, y que necesitan para su resolución herramientas eficientes, tanto métodos como infraestructuras, para ser resueltos en tiempos razonables. Entre los métodos matemáticos que se suelen usar para resolver este tipo de problemas están las metaheurísticas. Sin embargo, en la mayoría de los casos las metaheurísticas todavía necesitan mucho tiempo de cálculo en computadores convencionales para obtener buenos resultados. La computación de altas prestaciones (HPC), a través de la paralelización de las metaheurísticas y del uso de infraestructuras paralelas, puede contribuir a mejorar estos tiempos de respuesta.

Hay dos problemas a tener en cuenta al considerar el uso de las herramientas HPC. Por una parte, el acceso a los recursos, como clústeres privados, no están al alcance de todos los interesados. Por otra parte, paralelizar las metaheurísticas usando paradigmas tradicionales no resulta fácil para el programador si no tiene experiencia previa.

Afortunadamente, cada día es más sencillo poder acceder a potentes recursos computacionales a través de las nuevas infraestructuras cloud, que nos ofrecen plataformas como servicio (PaaS). Además, existen nuevos modelos de programación y herramientas que facilitan la tarea de paralelización al programador.

El objetivo de este trabajo es el estudio de estrategias de paralelización de una metaheurística muy popular, el algoritmo de optimización de enjambre de partículas o PSO (*Particle Swarm Optimization*), usando un framework para procesamiento distribuido que está ganando

popularidad para computación cloud y grandes volúmenes de datos, como es Spark. El principal objetivo del proyecto es evaluar la viabilidad de usar este tipo de frameworks, así como su rendimiento, para problemas complejos de estimación de parámetros usando entornos tipo cloud.

1.2 Metodología y plan de trabajo

Durante el desarrollo del trabajo propuesto se decidió aplicar la metodología ágil Scrum [1, 2] que proporciona un desarrollo incremental que nos permite la inspección continua del software. Scrum es un marco de trabajo para el desarrollo y mantenimiento de productos software. Se caracteriza por ser ligero y simple de entender, aunque difícil de dominar. Permite abordar problemas complejos y al mismo tiempo entregar productos con el máximo valor posible.

Entre las características de Scrum destacan: los equipos auto-organizados, de entre 3 y 9 personas para que sean lo suficientemente pequeños para ser ágiles y lo suficientemente grandes para abarcar una cantidad de trabajo significativa; los eventos, como los Sprints, que suponen una oportunidad para la inspección y adaptación de algún aspecto de la planificación; los artefactos, que representan trabajo o valor en diversas formas, como la pila del producto (*Product Backlog*) que permite capturar los requisitos; y las reglas asociadas que relacionan los eventos, los roles del equipo y los artefactos.

Con el fin de crear regularidad y minimizar la necesidad de reuniones no definidas en Scrum existen diversos eventos como: el *Sprint Planning*, donde se planifica el trabajo a realizar durante el Sprint; el *Daily Scrum*, donde el equipo planifica las siguientes 24 horas; el *Sprint Review*, donde se lleva a cabo la revisión del Sprint; y el *Sprint Retrospective*, donde se persigue la auto-inspección del equipo Scrum y la creación de un plan de mejoras para el siguiente Sprint.

Al tratarse de un desarrollo realizado por un equipo de 3 personas, con un único desarrollador, se adaptó la metodología Scrum al desarrollo del proyecto.

El desarrollo se dividió en una serie de Sprints, de duración variable dependiendo del número de tareas a abordar pero estableciendo como límite de máxima duración un mes. Al inicio de cada Sprint se llevaba a cabo una *Sprint Planning* para decidir las funcionalidades a abordar a partir de la pila del producto. Al finalizar cada Sprint se procedía a realizar la *Sprint Review*, con la participación de los directores, mostrando las funcionalidades completadas durante el mismo y definiendo los elementos de la pila del producto posibles para el siguiente Sprint.

Algunos eventos de Scrum como pueden ser las reuniones de retrospectiva no se realizaron debido a la naturaleza del trabajo. Sin embargo, cada día se empleó una adaptación de

las reuniones diarias, haciendo balance del día anterior y planificando el trabajo a realizar durante ese día, realizando así una introspección individual intentando dar respuesta a las siguientes preguntas: ¿qué hice ayer para ayudar a lograr el objetivo del Sprint?, ¿qué haré hoy para ayudar a lograr el objetivo del Sprint?, ¿detecto algún impedimento que evite lograr el objetivo del Sprint?

Durante todo el desarrollo se mantuvo la pila del producto, añadiendo funcionalidades futuras a medida que se decidían. De esta manera se llevó a cabo un desarrollo ágil e incremental, adaptándose a la evolución del proyecto y minimizando el impacto de los imprevistos que pudieran surgir.

Al empezar el proyecto se realizó una *Sprint Planning* donde se planificó el trabajo a realizar durante el primer Sprint a partir de la pila del producto inicial, con una duración aproximada de cuatro semanas. A medida que se terminaban los Sprints se realizaban los *Sprints Reviews* con el objetivo de revisar el trabajo y planificar el siguiente Sprint en la *Sprint Planning*. Finalmente, el proyecto estuvo compuesto por los siguientes Sprints:

- **Sprint 0:** Búsqueda de información y antecedentes en el estado del arte, así como la implementación del algoritmo PSO usando el lenguaje de programación Scala.
- **Sprint 1:** Configuración y ejecución en el clúster Pluton proporcionado por la UDC para comprobar la fiabilidad y rendimiento del algoritmo.
- **Sprint 2:** Propuesta de estrategias de paralelización a través de Spark e implementación de las mismas.
- **Sprint 3:** Análisis exhaustivo del rendimiento de las propuestas anteriores con diferentes benchmarks e infraestructuras.
- **Sprint 4:** Elaboración de una memoria final con la descripción de las soluciones, los resultados experimentales obtenidos y las conclusiones alcanzadas.

En la figura 1.1 se puede observar la forma de trabajo comentada anteriormente, organizada a través de Sprints y sus respectivos eventos. Por otro lado, en la tabla 1.1 se pueden ver las estimaciones para el trabajo de cada Sprint y el tiempo real dedicado a cada uno de ellos.

Todos los equipos, máquinas y recursos necesarios estaban disponibles antes del comienzo del proyecto, por lo que no se calcula los costes generados por estos elementos. En cuanto a los recursos humanos, en el análisis se tienen en cuenta a tres personas, con diferentes grado de implicación en el proyecto: el alumno y dos directores. Como se puede ver en la tabla 1.2 las horas dedicadas al proyecto son 447 horas. Utilizando como costo medio 35€/hora, salario medio en España para un desarrollador freeland según [3], y 60€/hora para los directores, el coste total del desarrollo asciende a 16845€.

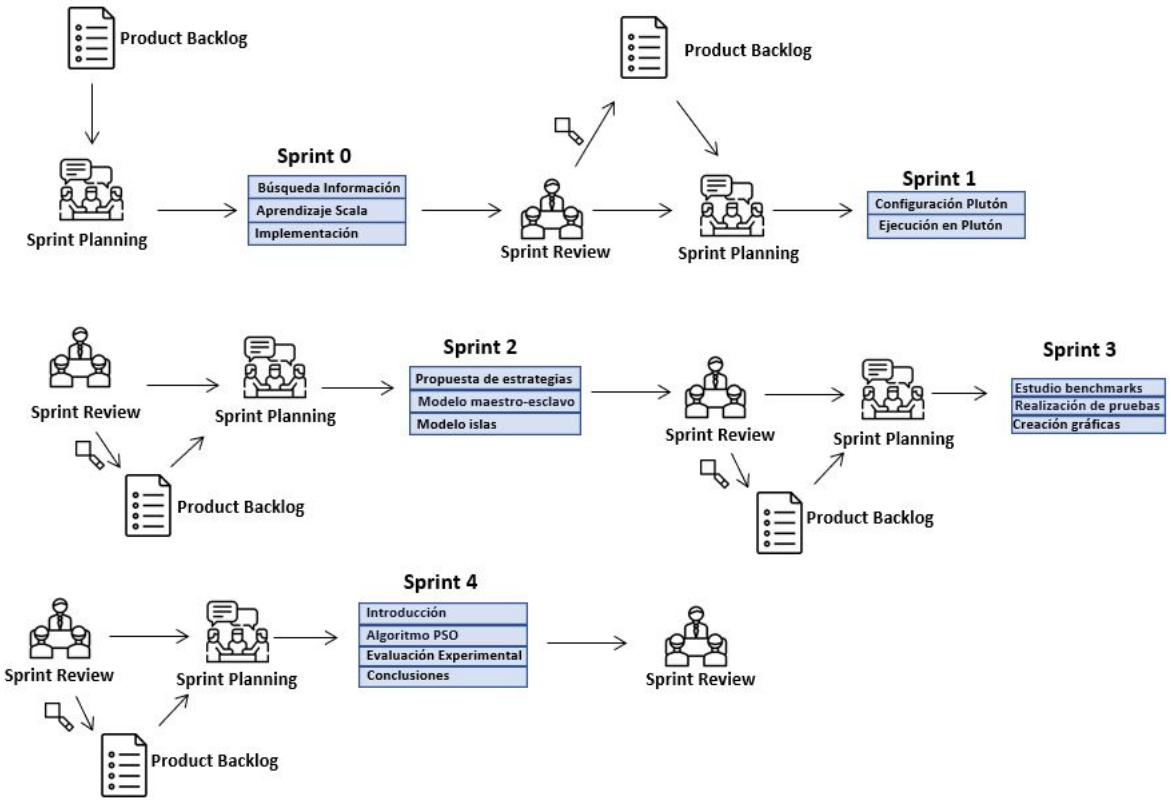


Figura 1.1: Esquema del plan de trabajo usando Scrum

Sprint	Comienzo	Fin	Estimado	Trabajo
Sprint 0	03/02/2020	02/03/2020	90 horas	96 horas
Sprint 1	04/03/2020	18/03/2020	45 horas	40 horas
Sprint 2	21/03/2020	20/04/2020	120 horas	116 horas
Sprint 3	23/04/2020	20/05/2020	100 horas	105 horas
Sprint 4	23/05/2020	24/06/2020	80 horas	90 horas
			Total	**447 horas**

Tabla 1.1: Planificación del proyecto

Rol	Horas	Precio Hora	Total
Alumno	447	35	15645 €
Directores UDC	20	60	1200 €
Total	447	-	16845 €

Tabla 1.2: Costes del proyecto

1.3 Estructura de la memoria

Esta memoria está organizada en cinco capítulos, contando con el presente, que se describen brevemente a continuación:

- El **capítulo 1** ayuda a contextualizar el proyecto, describiendo los objetivos del mismo, así como la metodología empleada, la planificación del trabajo y el coste del mismo.
- El **capítulo 2** realiza una introducción a conceptos previos necesarios para entender el desarrollo del trabajo. En este capítulo se incluye una breve explicación a lo que es la optimización y los métodos empleados en esta, así como una breve descripción del algoritmo abordado en este trabajo, el PSO. También se hace una introducción a la computación de altas prestaciones y al uso de entornos cloud como proveedor de recursos HPC, así como al framework utilizado para la implementación paralela en este trabajo.
- El **capítulo 3** describe en detalle la implementación secuencial del algoritmo, usada como base para el desarrollo paralelo y también para la posterior comparación. También aborda las estrategias de paralelización del algoritmo y las implementaciones llevadas a cabo.
- El **capítulo 4** se dedica a los resultados experimentales. Comienza con una descripción del entorno de pruebas utilizado para posteriormente discutir los resultados obtenidos de las diferentes estrategias de paralelización propuestas. En este capítulo se incluyen también pruebas en varias infraestructuras y una comparación entre ellas.
- El **capítulo 5** presenta las conclusiones obtenidas del trabajo, y también proporciona una breve descripción de posibles líneas de trabajo futuro.

Capítulo 2

Conceptos Previos

En este capítulo se realiza una introducción a aquellas cuestiones directamente relacionadas con el trabajo que se llevó a cabo, y que son necesarias para entender posteriormente los desarrollos realizados.

En primer lugar se describen los métodos de optimización, prestando especial atención a las metaheurísticas, y particularizando en el algoritmo de enjambre de partículas (PSO) que se usa en este trabajo.

Después, se incluye una introducción a la computación de altas prestaciones (HPC), y su importancia a la hora de resolver casos reales que, por sus características, requieren una gran cantidad de tiempo para completarse. También se describe la computación y los entornos cloud, así como los paradigmas y herramientas que se usan en la actualidad, entrando en más detalle en el framework Spark, utilizado en el desarrollo de este proyecto.

2.1 Métodos de optimización y metaheurísticas

La optimización matemática consiste en la selección del mejor elemento o solución entre aquellos de un conjunto o espacio de búsqueda dado, descrito por una o más funciones matemáticas [4]. Una clase importante de problemas que requieren de métodos de optimización son los problemas de estimación de parámetros [5]. En la actualidad, existen una gran cantidad de ellos que dependen de la utilización de un número de variables o parámetros adecuado para proporcionar una respuesta óptima. Estos están presentes en una gran variedad de áreas como la ingeniería, la logística o la bioinformática. La resolución eficiente de estos problemas tiene un gran impacto en diversos ámbitos de la sociedad: en la economía, ya que puede proporcionar una mejor manera de gestionar los recursos pertenecientes a un ente y hacer que este goce de una mejor salud económica; en el ámbito sanitario, ya que permite optimizar las cantidades utilizadas en la fabricación de fármacos y como consecuencia obtener productos mejor elaborados; o en el ámbito social, campo transversal a los comentados anteriormente,

ya que, la mejora económica y sanitaria se puede traducir en un incremento del nivel de vida para las familias que conforman una sociedad.

Para resolver los problemas de estimación de parámetros se suele hacer uso de métodos de optimización global [4]. Estos se pueden clasificar en:

- **Métodos deterministas**. Exploran todo el espacio de búsqueda y consiguen encontrar el óptimo. Sin embargo, el gran esfuerzo computacional que requiere la convergencia en la solución global hace que estos métodos no sean útiles en la mayoría de casos.

- **Métodos estocásticos**. Estos métodos no garantizan la convergencia al óptimo global pero consiguen una solución cercana a este en un tiempo razonable. En este trabajo nos vamos a centrar en estos métodos, porque el objetivo será resolver problemas muy costosos, computacionalmente hablando, y que necesitarían mucho tiempo y recursos si se resolviesen usando métodos determinísticos.

- **Métodos híbridos**. Nacen de la combinación de dos o más metodologías, como puede ser la unión de un método global y de una búsqueda local o de un conjunto de métodos de optimización global combinados entre ellos.

Entre los métodos estocásticas se encuentran las metaheurísticas, que siguen procedimientos iterativos combinando diferentes tipos de exploración y explotación del espacio de búsqueda. En cada iteración se suelen mover de la solución actual a otra solución cercana. De esta forma, en cada iteración se progresa hacía una solución cercana al óptimo.

Debido a su capacidad para encontrar una solución buena a problemas complejos en un tiempo razonable, las metaheurísticas se han convertido en una solución muy extendida para resolver problemas de optimización [6]. Cada metaheurística tiene su propio comportamiento y características. Las metaheurísticas se pueden clasificar en diferentes categorías:

- **Búsquedas basadas en población frente a búsquedas basadas en solución única**. En algoritmos basados en población existe un conjunto de soluciones que es modificado y gestionado por el algoritmo (es el caso de los algoritmos genéticos - GAs - o el algoritmo de enjambre de partículas - PSO), mientras que en caso de búsquedas basadas en solución única el algoritmo cuenta con solo una solución que evoluciona a lo largo de la búsqueda (como en el algoritmo *Simulated Annealing*).

- **Inspiradas en la naturaleza frente a no inspiradas en la naturaleza**. Existen algoritmos que se basan en el comportamiento de determinados elementos de la naturaleza (por ejemplo, el PSO), mientras que otros no incorporan estos elementos (como el *Differential Evolution* - DE).

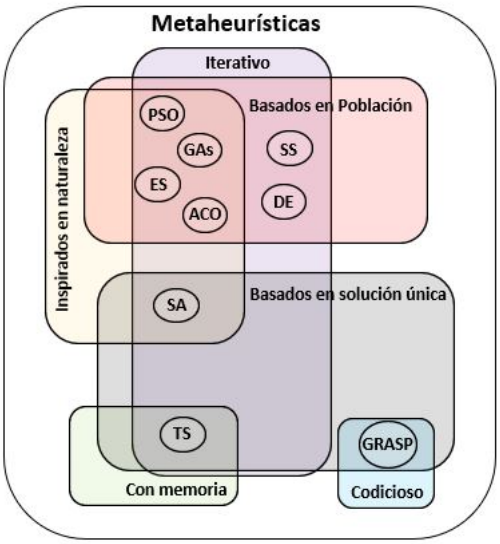


Figura 2.1: Clasificación de las metaheurísticas más populares

- **Iterativo frente a codicioso**. En los algoritmos codiciosos (*greedy*) se empieza con una solución vacía y en cada paso se asigna una variable de decisión hasta que se alcanza el resultado (por ejemplo, el algoritmo GRASP). En el caso de las metaheurísticas iterativas, las más populares, comienzan con una o más soluciones que evolucionan en cada iteración usando los mismos operadores de búsqueda (es el caso del algoritmo PSO).

- **Métodos con memoria frente a métodos sin memoria**. Algunos métodos almacenan información sobre el periodo de búsqueda (por ejemplo, los métodos que usan búsqueda tabú - TS).

En la figura 2.1 se muestra la clasificación de las metaheurísticas comentada anteriormente y a que categoría pertenecen algunos de los algoritmos de optimización más usados, entre los que se encuentra el algoritmo usado en este trabajo.

2.1.1 Algoritmo de enjambre de partículas

El algoritmo de enjambre de partículas o *Particle Swarm Optimization* (PSO) [7] es un algoritmo de optimización inspirado en el comportamiento de algunos grupos animales como pueden ser los enjambres de pájaros o bancos de peces. Inicialmente se comienza con una población creada de forma aleatoria, esta evoluciona iterativamente a lo largo de la ejecución. Al contrario que otros algoritmos, el PSO no cuenta con operadores de evolución como la mutación o el cruce. En este caso, a través de la evaluación y movimiento individual de cada partícula se encuentran óptimos que conducen al conjunto hacia una solución global.

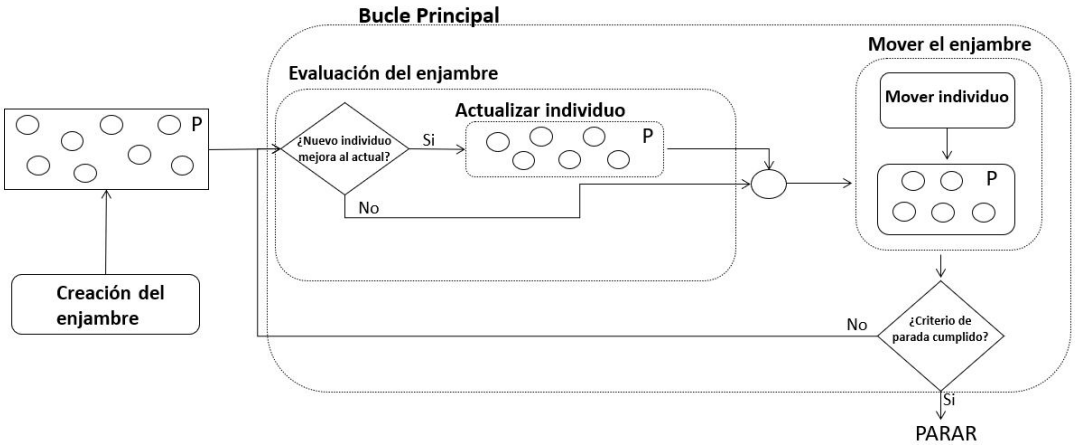


Figura 2.2: Representación esquemática del PSO

En la figura 2.2 se representa de manera simplificada el funcionamiento del PSO. Como se puede ver, el algoritmo se divide en una serie de fases:

- Creación del enjambre
- Evaluación del enjambre
- Movimiento del enjambre

Los individuos de la población, denominados en el contexto del algoritmo como partículas, recorren el espacio de búsqueda. Cada partícula posee su mejor valor histórico, que se actualiza de manera iterativa si se encuentra una solución mejor, sino el valor actual de la partícula se conserva.

2.2 Computación de Altas Prestaciones y Cloud

En la actualidad, existen diversas formas de aumentar la potencia computacional de los sistemas informáticos a la hora de resolver un problema dado: mejorando el hardware del dispositivo, modificando la lógica del algoritmo, proponiendo nuevos guarismos para el problema en cuestión, o empleando el procesamiento paralelo, entre otros.

La *computación de altas prestaciones (High Performance Computing - HPC)* utiliza el procesamiento paralelo para resolver problemas computacionalmente muy costosos, que requieren para su resolución de un gran número de recursos y mucho tiempo de ejecución [8] [9]. La estrategia de paralelizar se basa en dividir un trabajo en diferentes tareas para que puedan ser ejecutadas de forma concurrente.

El *cloud* [10] se puede definir como un paradigma que hace uso de diferentes tecnologías para facilitarle a los usuarios el acceso a los *recursos TI*. El concepto de recurso TI hace

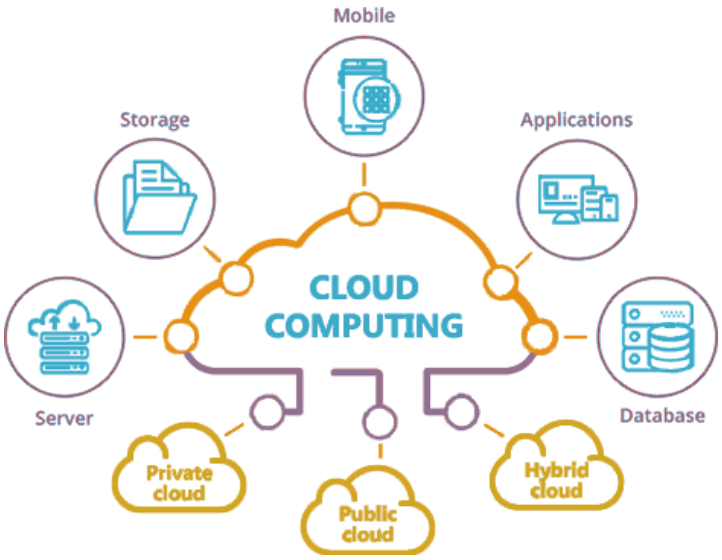


Figura 2.3: Modelos de aprovisionamiento y recursos TI ofrecidos en el cloud

referencia a redes, servidores, almacenamiento, aplicaciones y servicios. Un aspecto importante de esta plataforma es la virtualización de recursos, que permite la reutilización de estos aumentando su disponibilidad.

Existen diferentes tipos según su privacidad: público, privado y híbrido. Un cloud público pertenece a un proveedor de servicios y se ofrece a través de una suscripción, cualquier usuario que haya pagado por el servicio puede acceder a su uso. En caso del cloud privado, se crea dentro de la intranet de una organización y su acceso está limitado para los clientes, además, suele ser más eficiente que los públicos. Finalmente, los cloud híbridos están compuestos por los clouds comentados anteriormente.

Los servicios que ofrece el cloud son variados y se pueden agrupar en tres modelos de servicio [11]: *Software como servicio (SaaS)*, donde se ofrece una solución completa; *Plataforma como servicio (PaaS)*, donde una parte del software se encapsula y se ofrece al usuario; e *Infraestructura como servicio (IaaS)*, donde se ofrece un conjunto de recursos TI como almacenamiento o servidores sobre una red. En la figura 2.3 se muestran algunos de los recursos ofrecidos en el cloud [12].

La computación cloud describe un nuevo paradigma de computación donde un gran número de sistemas se conectan a redes públicas o privadas para proporcionar una infraestructura que dé soporte a aplicaciones, datos y almacenamiento, que se pueden escalar dinámicamente. La naturaleza paralela del HPC provoca un gran consumo de recursos que pueden ser proporcionados por las infraestructuras cloud gracias a su capacidad para realizar reservas de nuevos entornos rápidamente. Además, el cloud proporciona una escalabilidad y elasticidad de recursos a los usuarios que precisen HPC de la que no disponen habitualmente en los

centros de supercomputación tradicionales.

Para sacar partido a la computación cloud se pueden usar diferentes paradigmas y herramientas. El más popular es el modelo *Map-Reduce* [13], los usuarios especifican una función de mapeo (*map*) que recibe como parámetros un par clave-valor y que devuelve una lista de pares, a esta lista se le aplica una función (*reduce*) en paralelo que se llama una vez por cada clave de salida del *Map*. Este modelo se ha aplicado con éxito a un gran número de aplicaciones, como búsquedas de patrones, ordenamiento distribuido o procesamiento gráfico, entre otros. No obstante, se mostró ineficiente a la hora de tratar con algoritmos iterativos debido a que no proporciona una forma eficiente de reutilizar los resultados de iteraciones previas. Por este motivo, surgieron alternativas a este paradigma que no están basadas en este modelo, como Spark [14], que será el framework que se utilizará en este trabajo para realizar la paralelización del algoritmo PSO.

2.2.1 Spark

Spark es un framework de procesamiento distribuido usado normalmente en entornos *Big Data*. Es compatible con diferentes lenguajes de programación como Java, Python o el lenguaje elegido para el trabajo, Scala. Además, ofrece librerías que dan soporte en diferentes ámbitos como: procesamiento de SQL, aprendizaje máquina o análisis en tiempo real.

La estructura de los programas en Spark está formada por un proceso director (*driver*) que se corresponde con el *main()* de la aplicación. Una vez comienza su ejecución divide el código de usuario en tareas que se envían a los procesos trabajadores (*workers*) que las procesan y las devuelven al proceso director. Estos procesos pueden correr localmente en una única máquina virtual o consumiendo recursos de un clúster. No existe comunicación entre los procesos workers, por lo que para actualizar sus datos es necesario siempre usar el proceso driver. Los procesos workers contienen los ejecutores (*executors*) que representan un proceso capaz de realizar tareas y almacenar información.

En esta estructura también participa el *contexto* de Spark, que representa el punto de entrada a la ejecución del programa, y actúa como un cliente del entorno de ejecución. Una vez se crea se pueden utilizar todas las herramientas proporcionadas por Spark. En la figura 2.4 puede verse una representación simplificada de la arquitectura comentada.

A diferencia de otros frameworks, como *Hadoop [15]*, basados en el modelo *MapReduce [13]*, que usan el disco como almacenamiento principal, Spark realiza la mayor parte de sus cálculos en memoria, primando de esta manera la velocidad en ciertas aplicaciones como aquellas que usan algoritmos iterativos. Spark trabaja con lo que se denomina conjunto de datos distribuidos resilientes, o *resilient distributed datasets (RDDs)*, que son conjuntos de datos que mejoran la eficiencia de los cálculos, sin coste de entrada/salida, y que aumentan la tolerancia a fallos.

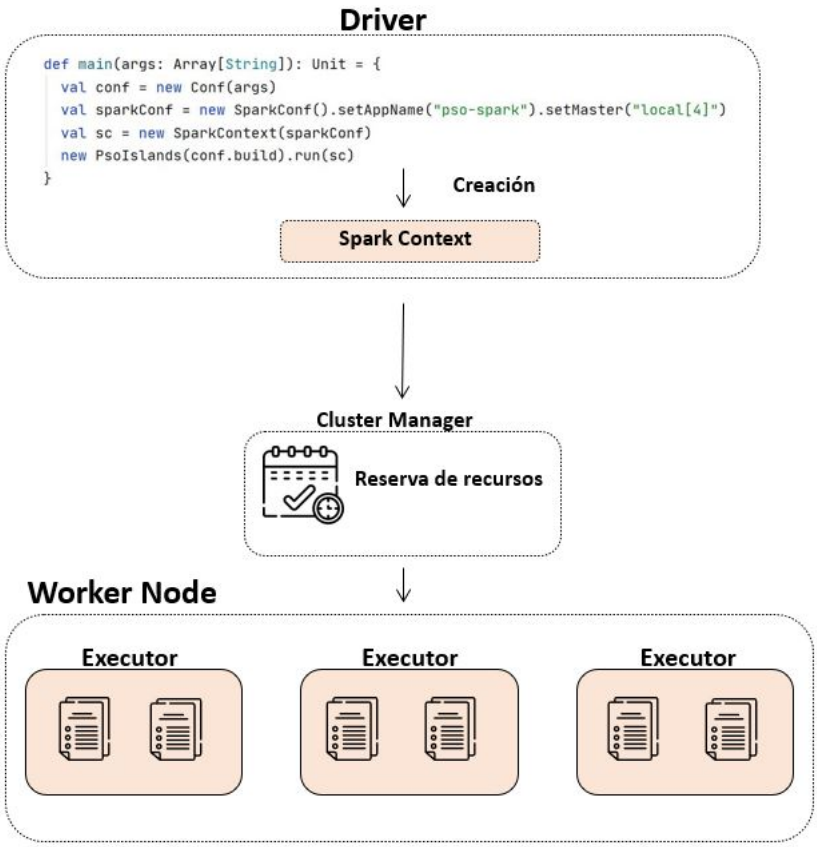


Figura 2.4: Arquitectura Spark

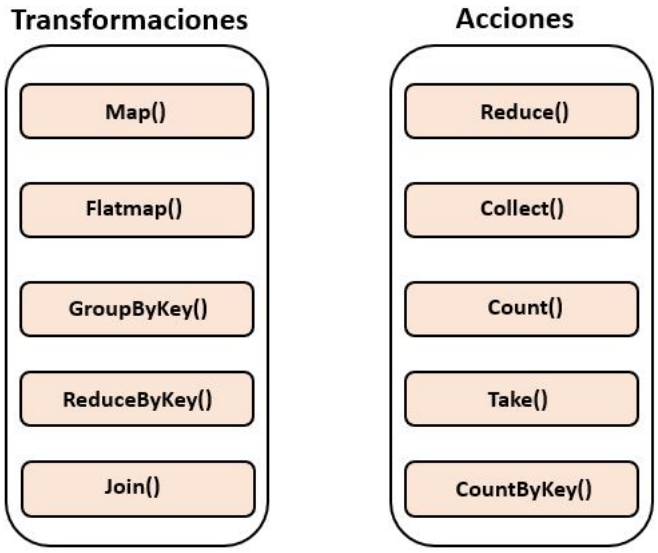


Figura 2.5: Diversas acciones y transformaciones de Spark

Spark permite dos tipos de operaciones sobre RDDs: las transformaciones y las acciones. Por un lado, las transformaciones devuelven un nuevo RDD y son calculadas de manera perezosa (*lazy*), esto quiere decir que solo se ejecutan cuando se invoca una acción. Por su lado, las acciones son operaciones que devuelven un resultado al *driver* y se ejecutan en el momento de su invocación. En la figura 2.5 pueden verse varias transformaciones y acciones que proporciona Spark.

Entre las ventajas que proporciona Spark, en relación especialmente con el desarrollo del proyecto, destacan las siguientes:

- Facilidad de programación: Spark ofrece la posibilidad de ejecutar un programa a través de un shell interactivo. De esta forma, permite el desarrollo de aplicaciones de una manera más rápida y sencilla.

- Tolerancia a fallos: Spark posee una gran capacidad de recuperación ante posibles fallos del sistema gracias al uso de los RDDs, ya que están basados en grafos acíclicos dirigidos que, ante la caída de un nodo, son capaces de recomponerse. Además, la información no se almacena en un solo nodo, sino que está distribuida, aumentando con ello la tolerancia a fallos.

- Reuso de datos ya procesados: al usar la memoria como almacenamiento principal facilita el mantenimiento de datos hasta que los trabajos finalicen, lo que es fundamental para mejorar el rendimiento de algoritmos iterativos como los contemplados en este trabajo.

- Optimización a bajo nivel: el modo de ejecución perezoso permite aplicar múltiples optimizaciones antes de comenzar la ejecución de todas las tareas.

Capítulo 3

Implementación

En este capítulo se describen las implementaciones, tanto secuencial como paralelas, realizadas en este trabajo. Al final del capítulo se indica la referencia al repositorio donde se pueden encontrar todos los desarrollos, junto con una breve descripción de su estructura.

3.1 Implementación secuencial

En el capítulo anterior se presentó brevemente el algoritmo de enjambre de partículas y sus fundamentos. Este algoritmo es una de las metaheurísticas más populares que se pueden encontrar en la bibliografía. Sin embargo, como es habitual en este campo, existen múltiples variantes de un mismo algoritmo básico. En esta sección se detalla la implementación que se ha realizado para este trabajo.

Como ya se ha descrito en la sección 2.1.1, el PSO es un algoritmo iterativo, que se compone de una serie de fases o etapas:

- Creación del enjambre. Cada individuo de la población está definido por una serie parámetros que varían a medida que la partícula se mueve: una serie de vectores que representan la posición exacta, la velocidad y la mejor posición en la que estuvo la partícula en algún momento, y por otro lado, valores absolutos como el valor actual del individuo y su mejor valor histórico. En el momento de la creación se inicializan la posición y velocidad de manera aleatoria para cada partícula perteneciente al enjambre, el resto de parámetros no se inicializan. Además, el valor aleatorio para cada posición del vector está comprendido entre un límite inferior y superior.

- Evaluación del enjambre. Fase en la que se evalúa cada partícula del enjambre de manera individual. Consiste en determinar el valor de la función objetivo en la posición actual. Además, si el valor en la posición actual es mejor que el valor histórico del individuo se procede a actualizarlo. Finalmente, una vez se evaluó a cada una de las partículas,

se comparan sus mejores valores y se elige al mejor individuo. De manera análoga, se compara el mejor valor del individuo actual con el mejor individuo histórico y este último se actualiza en caso de ser mejorado.

- Mover el enjambre. Etapa de gran importancia ya que proporciona al algoritmo la capacidad de optimizar. Mover el enjambre implica actualizar la velocidad y posición de cada partícula. La velocidad se actualiza a través del cálculo de la siguiente ecuación:

$$v_i(t+1) = wv_i(t) + c_1 r_1[\hat{x}_i(t) - x_i(t)] + c_2 r_2[g(t) - x_i(t)] \tag{3.1}$$

  Cada elemento de la ecuación se define como:

  - $v_i(t+1)$, corresponde a la nueva velocidad de la partícula.
  - v_i, velocidad actual de la partícula.
  - w, coeficiente de inercia de la partícula, que consigue aumentar o reducir su velocidad.
  - c_1, coeficiente cognitivo.
  - r_1, vector formado por valores aleatorios entre 0 y 1.
  - $\hat{x}_i(t)$, mejor posición histórica de la partícula i en el momento t.
  - $x_i(t)$, posición de la partícula i en el momento t.
  - c_2, coeficiente social.
  - r_2, vector formado por valores aleatorios entre 0 y 1.
  - $g(t)$, la mejor posición del enjambre en el momento t.

  La ecuación está formada por tres componentes fundamentales:

  - $wv_i(t)$, definido como el componente de inercia, se encarga de que la partícula mantenga la dirección en la que se mueve.
  - $c_1 r_1[\hat{x}_i(t) - x_i(t)]$, es el componente cognitivo, que se encarga de que la partícula se mueva en la dirección correcta.
  - $c_2 r_2[g(t) - x_i(t)]$, componente social, que trata de que el enjambre se mueva hacia la mejor posición encontrada hasta el momento.

  Una de las características frecuentes del PSO es que suele alcanzar velocidades muy altas y por este motivo se suelen establecer límites que acoten su valor. No obstante, en este caso, dado que tratamos con algunos problemas multimodales, no se implementaron los límites puesto que aumentan las posibilidades de estancamiento en un óptimo local.

Algorithm 1 Pseudocódigo del bucle principal

```
1:  for 1:NumeroParticulas do
2:      Inicializar velocidad y posicion
3:  end for
4:  while No se cumpla la condicion de parada do
5:      for 1:NumeroParticulas do
6:          Evaluar particula
7:          if Valor actual es mejor que valor historico then
8:              Actualizar particula
9:          end if
10:         Mover partícula
11:     end for
12:     Devolver mejor particula
13: end while
```

Por su parte, la posición de cada partícula se actualiza a través de la siguiente ecuación:

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{3.2}$$

Para finalizar, a partir de la primera fase de creación del enjambre, la segunda y tercera etapa se repiten iterativamente hasta que se cumpla el criterio de parada. Este criterio puede establecerse a través de un tiempo máximo, de un número de iteraciones limite o cuando se alcanza una solución objetivo. Los dos primeros criterios suponen una parada por el esfuerzo y el último representa una parada por calidad de la solución alcanzada. En este trabajo se han implementado todos estos criterios, de forma que el usuario pueda seleccionar el que más le interese.

El algoritmo 1 muestra el pseudocódigo, para el lazo iterativo, de la implementación realizada. Esta implementación se llevó a cabo usando el lenguaje de programación Scala [16], lenguaje multi-paradigma de alto nivel que combina la orientación a objetos con la programación funcional. Además, está diseñado para presentar patrones de programación de forma elegante y sencilla. En la figura 3.1 se puede ver el bucle principal de la implementación secuencial escrito en Scala.

La elección de Scala como lenguaje de programación está motivada por las ventajas que nos proporciona para este tipo de problemas. En primer lugar, posee un sistema de inferencia de tipos muy sofisticado y permite usar fácilmente librerías de Java. Por otro lado, la *API* que ofrece Spark [14], el framework que usaremos para la paralelización, tiene una relación 1 a 1 con las operaciones sobre las colecciones en Scala. De esta forma es fácil gestionar grandes cantidades de datos a través del manejo funcional de listas en Scala. Finalmente, la sencillez o elegancia en la programación que proporciona Scala es otro punto a favor para la elección de este lenguaje. Gracias a su parte funcional que permite escribir código con una *sintaxis ligera*

```
do{
  val initTime = System.nanoTime

  enjambre = evaluarEnjambre(enjambre, func, optimizacion)

  val new_inercia = (inercia_max - inercia_min) * (n_iteraciones - i) / (n_iteraciones + inercia_min)

  enjambre = moverEnjambre(enjambre, new_inercia, peso_cognitivo, peso_social)


  val mejor_valor = enjambre.minBy(p => p.mejorValor.get).mejorValor.get

  historico_enjambre = historico_enjambre :+ data(mejor_valor,time)

  i+=1

  termination = if (parada >= BigDecimal(mejor_valor).setScale(120,BigDecimal.RoundingMode.HALF_UP).toDouble) true else false
} while (if (criterio == "esf") i < n_iteraciones else if (criterio == "cal") !termination else (i < n_iteraciones && !termination))
```

Figura 3.1: Bucle principal para la implementación secuencial

proporciona una manera de escribir funciones de manera más simple y rápida.

3.2 Implementación Paralela

En esta sección se describe la implementación paralela empezando por especificar las distintas estrategias de paralelización y cuales se adaptan mejor a las infraestructuras elegidas. Posteriormente, se explica de manera detallada cada una de las estrategias propuestas.

3.2.1 Estrategias de paralelización

Las metaheurísticas basadas en población suelen paralelizarse utilizando alguno de los siguientes modelos:

- Modelo maestro-esclavo. Se trata de un modelo en el que un proceso adopta el papel de ”maestro” y reparte o divide el trabajo entre los demás procesos ”esclavos” que ejecutan la función de evaluación y aplican el operador de variación. Este modelo no modifica la lógica del algoritmo secuencial.

- Modelo basado en islas. Consiste en separar la población inicial en diversas ”islas”, que se asignarán a recursos computacionales diferentes, y ejecutar el algoritmo de manera aislada en cada una de ellas. No obstante, estas islas intercambian individuos y se comunican entre sí cada cierto tiempo para mejorar la convergencia del método global.

- Modelo celular. En este modelo se introduce el concepto de ”vecindad”, puesto que cada individuo de la población solo puede interactuar con sus vecinos más cercanos. El modelo de vecindad implantado en el algoritmo ayuda a explorar el espacio de búsqueda a través de una lenta difusión de las soluciones.

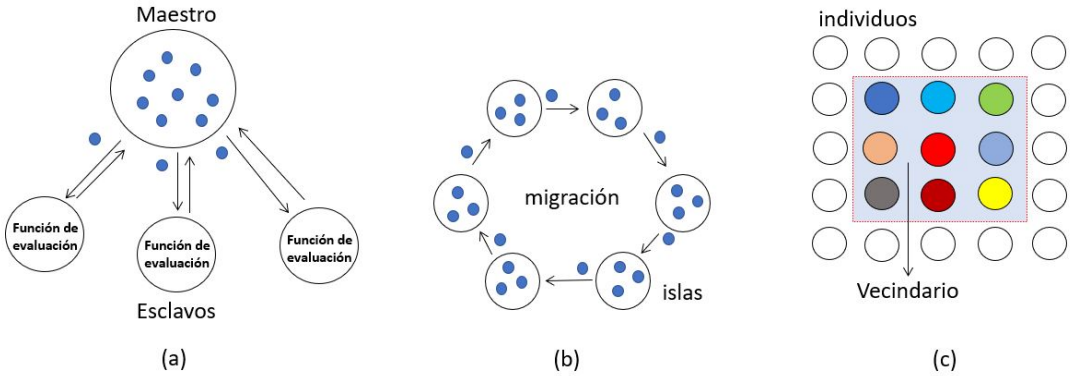


Figura 3.2: (a) Modelo maestro-esclavo; (b) Modelo basado en islas; (c) Modelo celular

Teniendo en cuenta las características, comentadas ya en la sección 2.2.1, de Spark y la imposibilidad de comunicación entre los *workers* directamente, el modelo celular, a priori, no resultaría fácil de implementar en este proyecto. Por ello, en este trabajo nos hemos centrado en la paralelización usando Spark de los otros dos modelos: el maestro-esclavo y el basado en islas.

3.2.2 Implementación maestro-esclavo

Cuando se utiliza Spark para implementar el modelo maestro-esclavo es el proceso *driver* el que asume el papel de maestro y reparte una serie de tareas entre los distintos *workers*, que toman el papel de esclavos, para que estas sean ejecutadas. En este caso, el proceso *driver* reparte el enjambre inicial entre los *workers* a través de la función *parallelize*. Es el programador quien decide que tareas son ejecutadas en paralelo. En la implementación realizada, se opta por hacer la función que evalúa las partículas distribuida, ejecutando en cada *worker* la función de evaluación sobre la partición que se le asignó para mejorar así el tiempo de respuesta del algoritmo. Además, es importante destacar, por su comparación posterior con otros modelos de paralelización, que en este modelo la lógica del algoritmo no sufre modificaciones, y por lo tanto las propiedades sistémicas del mismo no cambian.

La figura 3.3 representa el funcionamiento principal de esta estrategia. El único cambio frente a la implementación secuencial es el reparto de la función de evaluación de las distintas partículas entre los *workers*. Para el reparto de la función indicada en la figura se utiliza la transformación que proporciona Spark denominada *map()*, gracias a esta, cada *worker* recibe la función para evaluar la partícula. Por otra parte, para obtener los resultados de los *workers* en el *driver* se utiliza la acción *collect()*. Es necesario destacar que ningún *map()* es ejecutado hasta que se produzca el anterior *collect()* al ejecutar el código en el interior del bucle.

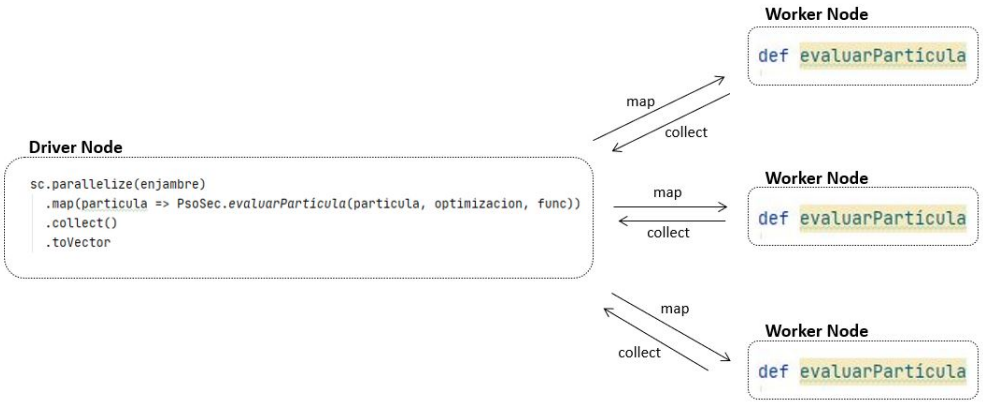


Figura 3.3: Representación esquemática de la implementación maestro-esclavo

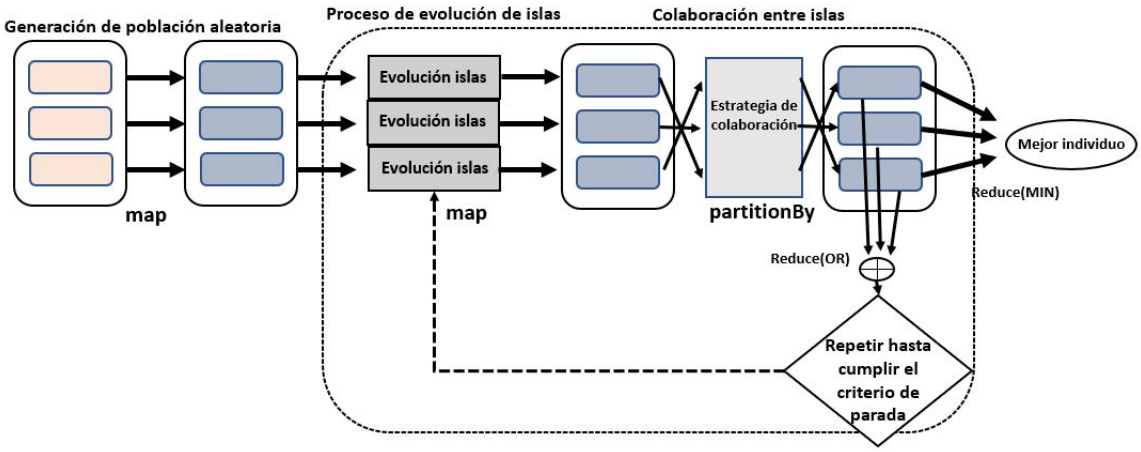


Figura 3.4: Representación esquemática del modelo basado en islas

3.2.3 Implementación basada en islas

Cuando se utiliza Spark para implementar el modelo basado en islas, el proceso *driver* reparte la población inicial entre diversos procesos que juegan el papel de islas. Cada una de estas islas ejecuta el algoritmo de forma independiente. Esta implementación modifica la lógica del algoritmo, puesto que los valores con los que trabaja cada isla son los locales durante la evolución del enjambre.

En la figura 3.4 se muestra una representación esquemática de la implementación realizada. El algoritmo empieza generando aleatoriamente la población, de manera similar a la versión secuencial, pero en este caso se opta por convertir la población inicial en un RDD para su posterior ejecución paralela. En la generación inicial se crean las islas implícitamente a través de la operación *parallelize*.

Es importante destacar que para el reparto de la población entre las islas se implementa

una clase *Partitioner* que es la encargada de asignar cada individuo de la población a una isla determinada. En el trabajo se opta por un *Partitioner* que asigna de manera aleatoria individuos a islas. La ventaja de este es que es sencillo implementarlo, pero tiene el inconveniente de que puede provocar un desbalanceo entre las diferentes islas ya que puede haber islas que acaben con un número de individuos mucho mayor que otras y que, por lo tanto, posean una carga de trabajo mayor.

Las islas ejecutan un número determinado de iteraciones del algoritmo de forma independiente. Una vez concluidas estas iteraciones, el *driver* recupera toda la población mejorada y activa la fase de colaboración, donde la población se vuelve a distribuir de forma aleatoria entre las diferentes islas. Teniendo en cuenta el tipo de acciones y transformaciones que permite Spark, se ha optado por utilizar la acción *PartitionBy()* para, en las fases de colaboración, barajar los individuos de las poblaciones actuales entre las diferentes islas. Esta estrategia permite aumentar la diversidad en la búsqueda de las islas, ya que estas vuelven a comenzar la búsqueda con una nueva población que se ha constituido a partir de las soluciones prometedoras que provienen de diferentes búsquedas, lo que beneficia especialmente a problemas difíciles que habitualmente corren peligro de quedarse estancados en óptimos locales.

Este proceso se repite hasta que se cumple el criterio de parada. En ese momento se llama a la acción *reduce()* para devolver el mejor individuo de la población, que en nuestra implementación es el que tiene el menor valor de la función objetivo.

En el proyecto se ha optado por la estrategia de barajar la población entre las distintas islas tras la ejecución de un número de iteraciones dado. No obstante, otras estrategias de colaboración suelen consistir en introducir la mejor partícula encontrada hasta el momento en el conjunto de islas, sustituyendo a la peor partícula de cada enjambre. Esta solución, si bien es simple, se ha descartado, porque habitualmente lleva a las islas a una pérdida de diversidad y a que la partícula prometedora que se introduce en todas las islas actúe como un *atractor* para el resto de partículas de los enjambres, lo que, en problemas difíciles, puede llevar al algoritmo a estancarse fácilmente en óptimos locales.

3.3 Repositorio

Todos los desarrollos de este proyecto se encuentran disponibles en el siguiente repositorio: `https://github.com/Roberto-97/ParticleSwarnOptimization`

El código del trabajo está dividido en una serie de módulos que ayudan a organizar mejor la estructura del mismo. Esta se muestra gráficamente en la figura 3.5. El módulo *Common* contiene archivos de configuración y de uso global para el programa, mientras que en los módulos *Secuencial* y *Spark* se han separado los códigos de las implementaciones secuencial y paralela. Por su parte, el módulo *Main* contiene los programas principales para ejecutar cada

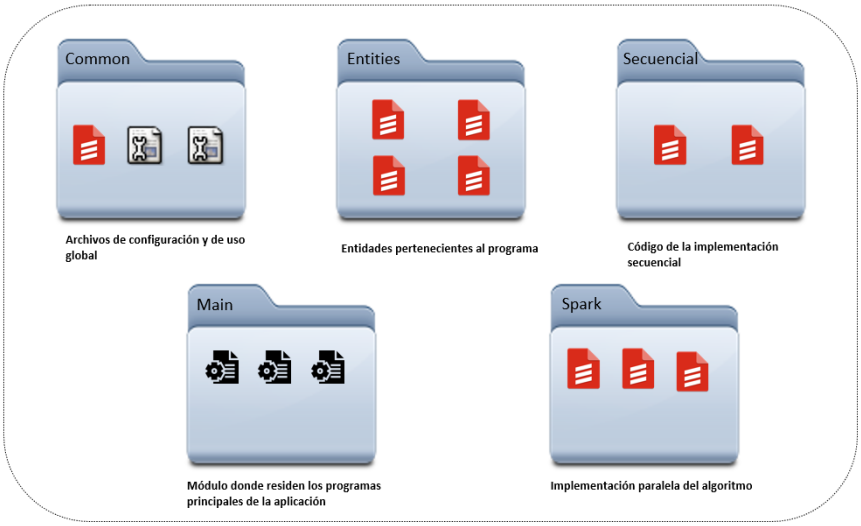


Figura 3.5: Estructura del proyecto

una de las implementaciones y el módulo *Entities* contiene todas las entidades necesarias. Esta modularidad permite mejorar el mantenimiento de código y la escalabilidad del mismo, gracias a la facilidad para introducir cambios en los distintos módulos.

Capítulo 4

Evaluación Experimental

Esta sección se dedica a la evaluación experimental de las propuestas anteriores. El capítulo comienza con una descripción del entorno de pruebas, y después se muestran los resultados y conclusiones obtenidos de los experimentos.

El principal objetivo de este capítulo es evaluar el impacto de los diferentes modelos de paralelización propuestos, comparando los resultados obtenidos con la implementación secuencial, y ofreciendo conclusiones que puedan ser útiles a otros desarrolladores interesados en soluciones paralelas para problemas similares.

4.1 Descripción del entorno de pruebas

En este apartado comenzaremos describiendo los problemas de prueba que hemos usado para la evaluación, las plataformas empleadas y finalmente los tipos de pruebas que se han realizado y la finalidad de cada una.

Los códigos que se han utilizado en los experimentos que se muestran en este capítulo se nombrarán de la siguiente manera:

- **seqPSO:** versión secuencial del *PSO*.
- **msPSO:** versión paralela basada en el modelo maestro-esclavo.
- **iscPSO:** versión paralela basada en el modelo en islas sin cooperación entre ellas.
- **iccPSO:** versión paralela basada en el modelo en islas con cooperación entre ellas.

Para facilitar la comparación de estos experimentos con otros que se puedan encontrar en propuestas similares en la bibliografía, por cada experimento se proporcionarán, en las siguientes secciones, los parámetros de configuración. Además, debido al comportamiento

estocástico del algoritmo, se realizan de forma independiente 20 ejecuciones por cada experimento, y se tiene en cuenta la distribución de los datos en la evaluación y discusión de resultados.

4.1.1 Problemas de prueba

Para la realización de los experimentos se han utilizado funciones de prueba populares a la hora de evaluar algoritmos de optimización. Una revisión de las más populares se puede encontrar en [17]. Para los experimentos de este trabajo hemos elegido 4 funciones de prueba que presentan comportamientos diferentes en los experimentos realizados, lo que nos permite una evaluación más exhaustiva y sacar más conclusiones sobre el comportamiento de las propuestas. El objetivo de estas pruebas es optimizar buscando el mínimo de cada uno de estos problemas.

A continuación se describen los problemas de prueba elegidos para los experimentos:

- Ackley → Función no convexa que destaca por su multimodalidad y está caracterizada por una región exterior casi plana y un gran agujero en el centro. Esta función contiene numerosos picos donde el algoritmo se puede estancar (ver figura 4.1a). Se define a partir de la siguiente expresión:

$$f(x) = -20e^{-0.2\sqrt{D^{-1}[\sum_{i=1}^{D} x_i^2]}} - e^{D^{-1}[\sum_{i=1}^{D} \cos(2\pi x_i)]} + 20 + e \tag{4.1}$$

- Quadric → Se trata de la función más simple que hemos tratado, con un único mínimo global localizado en el valor 0, como se aprecia en la figura 4.1b . Se define por:

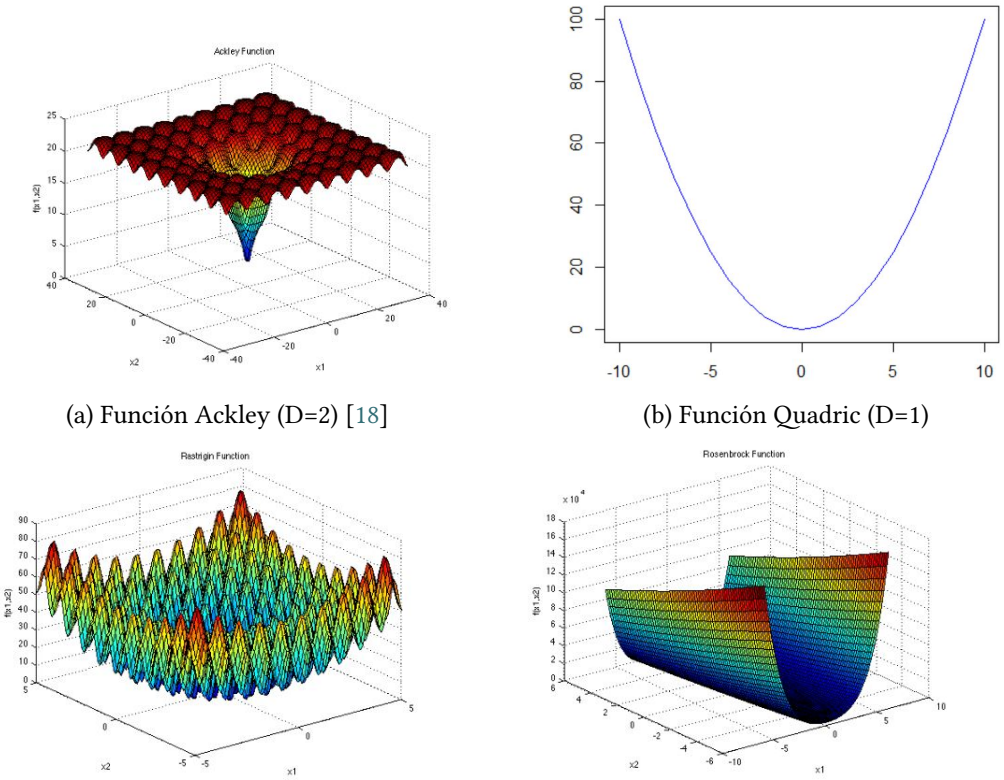
$$f(x) = \sum_{i=2}^{D} x_{i-1} * x_i \tag{4.2}$$

- Rastrigin → Esta es una función altamente multimodal (ver figura 4.1d), pero la localización de sus mínimos está distribuida regularmente. Se define con la siguiente expresión:

$$f(x) = 10D + \sum_{i=1}^{D} [x_i^2 - 10\cos(2\pi x_i)] \tag{4.3}$$

- Rosenbrock → Esta es una función unimodal (ver figura 4.1c) donde el mínimo global está localizado en el valor 0. Se define por la expresión:

$$f(x) = \sum_{i=1}^{D-1} [100(x_i + 1 - x_i^2)^2 + (x_i - 1)^2] \tag{4.4}$$



(a) Función Ackley (D=2) [18]

(b) Función Quadric (D=1)

(c) Función Rastrigin (D=2) [19]

(d) Función Rosenbrock (D=2) [20]

Figura 4.1: Representación de las funciones de prueba usadas

Se realizaron pruebas con distintas configuraciones para adecuar los parámetros y conseguir tiempos de ejecución razonables para las pruebas de este trabajo. La configuración del peso social y del peso cognitivo, parámetros que se utilizan a la hora de mover el enjambre para encontrar de una manera más rápida la solución, se estableció en el valor 2 en ambos casos por ser este uno de los valores más usados en otros trabajos de la bibliografía. Por otro lado, la dimensión del problema (D), es el número de variables que contiene el vector de posición de cada partícula. La dificultad del problema se ve incrementada con su dimensión, ya que el espacio de búsqueda aumenta exponencialmente. En las funciones de test usadas en este trabajo se ha optado por una dimensión de entre 10 y 20 variables.

4.1.2 Plataformas de prueba

En primer lugar, se realizaron pruebas locales con un procesador Intel(R) Core(TM) i7-855U compuesto por 4 cores a 1.80 GHz. Los experimentos locales se utilizaron para entender el comportamiento de cada función de prueba e ir ajustando de esta manera los parámetros utilizados, pero esas pruebas no se muestran en esta memoria.

	compute-0-0 a compute-0-17
CPU Model	2 x Intel Xeon E5-2660 Sandy Bridge-EP (0-0 a 0-16) 2 x Intel Xeon E5-2650c2 Ivy Bridge-EP (0-17)
CPU Speed/Turbo	2.20 GHz/3.0 GHz (0-0 a 0-16) 2.60 GHz/3.4 GHz (0-17)
#Cores por CPU	8
#Threads por core	2
#Cores/Threads por nodo	16/32
Cache L1/L2/L3	32 KB/256 KB/20 MB
Memoria RAM	64 GB DDR3 1600 Mhz
Discos	1 x HDD 1 TB SATA3 7.2K rpm
Aceleradoras	1 x NVIDIA Tesla Kepler K20m 5 GB GDDR5 (0-0 a 0-12) 1 x NVIDIA Tesla Kepler K40c 12 GB GDDR5 (0-13) 1 x Intel Xeon Phi 5110P 8 GB GDDR5 (0-14) 3 x NVIDIA Tesla Kepler K20m 5 GB GDDR5 (0-15) 2 x Intel Xeon Phi 5110P 8 GB GDDR5 (0-18) 1 x AMD FirePro S9150 16 GB GDDR5 (0-17)
Redes	InfiniBand FDR & Gigabit Ethernet

Tabla 4.1: Descripción de la cabina 0 del clúster Pluton

Los resultados que se muestran en la mayor parte de las secciones de este capítulo son fruto de experimentos que se realizaron en un clúster heterogéneo para computación de altas prestaciones denominado Pluton [21]. El clúster consta de un nodo front-end desde el exterior y nodos de cómputo que se encargan de proporcionar los recursos necesarios para la ejecución de los experimentos. Estos nodos se agrupan en dos cabinas que ofrecen diferentes recursos computacionales ya que poseen estructuras diferenciadas. Las pruebas realizadas en este trabajo siempre se ejecutan en los nodos de la cabina 0, cuya composición se puede ver en la tabla 4.1.

Por otro lado, aunque la mayoría de las ejecuciones para evaluar los resultados se han realizado en el clúster Pluton, también se llevaron a cabo diferentes experimentos en una plataforma cloud pública, proporcionada por el CESGA, y que está basada en OpenNebula v5.4.6 [22], que llamaremos *Nebula*. Estas pruebas se llevaron a cabo con el objetivo de comparar el rendimiento en un clúster privado y en un cloud público. OpenNebula es un middleware que permite el despliegue de cualquier tipo de cloud. Para poder ejecutar las pruebas necesarias para la comparación entre plataformas fue necesario configurar manualmente un clúster virtual Spark compuesto por 9 máquinas virtuales (8 nodos *workers* y 1 nodo *driver*). Estos se desplegaron en máquinas virtuales con el sistema operativo Ubuntu 18.04, los nodos *workers* contaron con 16 vcpus (virtual cpu) y 16 Gb de ram, mientras que el nodo *driver* contaba con 1 vcpu y 4 Gb de ram. En la figura 4.2 se puede ver el listado de las máquinas virtuales utilizadas

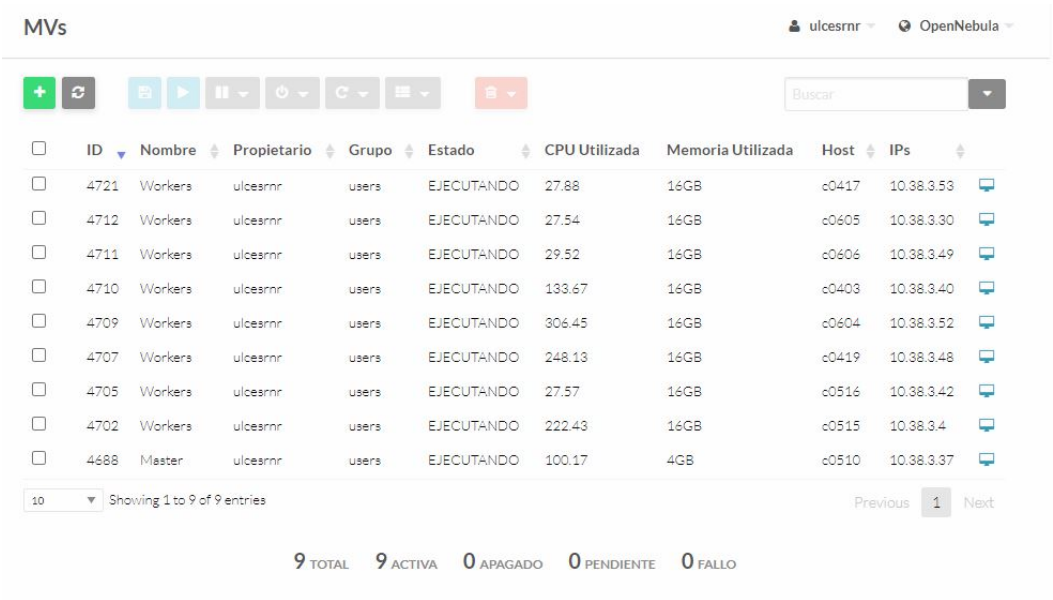


Figura 4.2: Máquinas del clúster virtual desplegado en la plataforma Nebula del Cesga

y parte de la configuración empleada.

4.1.3 Tipos de pruebas

Para evaluar exhaustivamente el comportamiento de las propuestas paralelas, se realizan pruebas desde diferentes perspectivas. En el diseño de los experimentos a realizar se siguieron los consejos y guías de [23]. En primer lugar, se realizaron las pruebas con un criterio de parada por esfuerzo. En este tipo de experimentos los benchmarks se ejecutan un número de iteraciones determinado, o un período de tiempo concreto. El objetivo de estos experimentos es determinar el valor final encontrado por cada una de las propuestas después de un esfuerzo determinado, común para todas ellas.

Por otro lado, también se realizaron las pruebas con un criterio de parada por calidad. En ellas se establece una solución a la que los diferentes experimentos deben llegar. Aquí no se compara el valor final, puesto que en todos los experimentos se llega al mismo valor, sino la rapidez con la que cada propuesta es capaz de alcanzar la solución establecida. El problema de las pruebas con criterio de parada por calidad es el tiempo que en algunos casos se necesita para completar los experimentos. Por este motivo, en las pruebas realizadas con parada por calidad se utilizó en la práctica un criterio de parada combinado, estableciéndose un valor objetivo y un esfuerzo máximo. El algoritmo termina cuando alguno de los dos criterios se cumpla.

4.2 Evaluación del modelo maestro-esclavo

El modelo maestro-esclavo, que se ha descrito en el capítulo anterior, no modifica las propiedades sistémicas del algoritmo, simplemente realiza parte del trabajo (evaluación de los individuos) de forma paralela. Por este motivo, los experimentos que se han realizado para su evaluación consisten en pruebas de esfuerzo, para determinar la rapidez con la que la versión paralela ejecuta el trabajo de la secuencial. En la tabla 4.2 se muestran los parámetros utilizados para estos experimentos con cada uno de los programas de prueba, donde D es la dimensión, NP el tamaño de la población y EM el esfuerzo realizado en número de iteraciones.

Función	D	NP	EM
Ackley	10	1200	200
Quadric	10	1200	200
Rastrigin	10	1200	200
Rosenbrock	10	1200	200

Tabla 4.2: Parámetros para la evaluación por esfuerzo en el maestro-esclavo

En la tabla 4.3 se muestran los resultados de valores alcanzados y tiempo necesario para todos los experimentos. Al no variar las propiedades del algoritmo con la paralelización, lo esperable sería que el valor alcanzado de media entre las 20 ejecuciones que se realizaron de cada experimento fuese aproximadamente el mismo, mientras que el tiempo de ejecución se esperaría que mejorase a medida que se aumenta el número de esclavos. Sin embargo, esto no es lo que ocurre en estos experimentos. Se puede ver que para 2 workers la versión de Spark empeora mucho la versión secuencial en Scala.

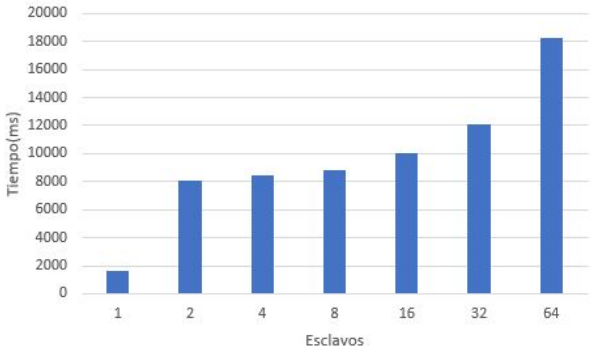
Dado que la implementación de esta estrategia, como se ha comentado en la sección 3.2.2, es muy sencilla y solo supone un cambio en el código en Scala para introducir las funciones de mapeo y recolección en Spark, estos resultados indican una sobrecarga de estas funciones muy significativa. La sobrecarga que se muestra en la tabla ha sido calculada como:

$$Sc = (T_i - T_1)/T_1 \tag{4.5}$$

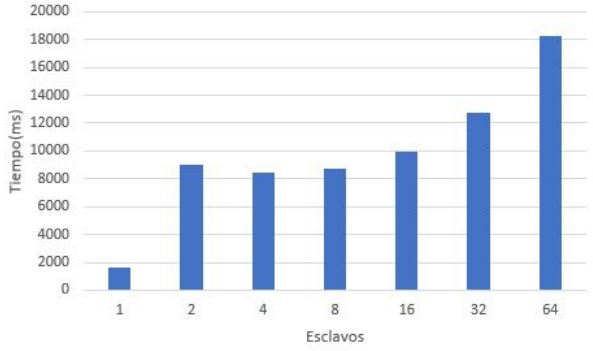
Esta sobrecarga está asociada a la cantidad de datos a repartir y al número de workers entre los que se reparte. La cantidad de datos a repartir es la misma en todos los experimentos, puesto que usamos el mismo tamaño de población para todas las versiones, secuencial y paralela. Por otra parte, cada worker, así como el driver, se asigna a un nodo físico de la máquina de hasta 16 cores, de forma que al aumentar el número de workers se necesitará más de un nodo. Esto explica que la sobrecarga se mantenga en valores más o menos constantes cuando el reparto se realiza entre cores del mismo nodo. Sin embargo, a partir de 16 cores, donde es

	Esclavos	Valor	Tiempo (ms)	Sc
Ackley	1	20.2 ± 0.1	1665 ± 176	-
	2	20.2 ± 0.1	8090 ± 512	3.9
	4	20.2 ± 0.1	8429 ± 2300	4.1
	8	20.2 ± 0.1	8786 ± 2267	4.3
	16	18.8 ± 5.2	10065 ± 2422	5.0
	32	20.2 ± 0.1	12094 ± 399	6.3
	64	20.2 ± 0.1	18301 ± 2943	9.9
Quadric	1	$(3.0 \pm 3.8)\text{e}{-10}$	1601 ± 142	-
	2	$(1.1 \pm 2.3)\text{e}{-9}$	8997 ± 2164	4.6
	4	$(6.3 \pm 11.7)\text{e}{-10}$	8433 ± 2049	4.3
	8	$(4.4 \pm 8.0)\text{e}{-10}$	8766 ± 1949	4.5
	16	$(3.7 \pm 5.1)\text{e}{-10}$	9981 ± 2264	5.2
	32	$(2.1 \pm 3.9)\text{e}{-9}$	12756 ± 2422	6.9
	64	$(1.4 \pm 2.2)\text{e}{-9}$	18250 ± 2746	10.4
Rastrigin	1	1.6 ± 0.7	1752 ± 150	-
	2	1.9 ± 0.9	8710 ± 1984	3.9
	4	2.1 ± 1.7	8328 ± 1978	3.7
	8	1.6 ± 0.9	8659 ± 1985	3.9
	16	2.4 ± 1.3	10049 ± 2171	4.7
	32	1.5 ± 0.7	12812 ± 2507	6.3
	64	1.8 ± 1.1	17888 ± 2728	9.2
Rosenbrock	1	420 ± 291	1591 ± 132	-
	2	339 ± 229	8717 ± 2033	4.5
	4	200 ± 106	8228 ± 1982	4.2
	8	249 ± 134	8835 ± 1996	4.6
	16	207 ± 179	9843 ± 2152	5.2
	32	207 ± 137	12757 ± 2475	7.0
	64	202 ± 136	18696 ± 2793	10.8

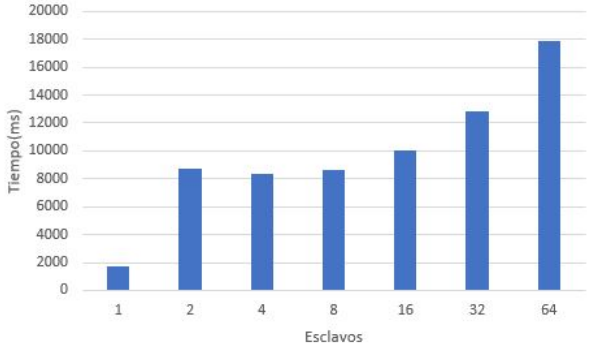
Tabla 4.3: Resultados del modelo maestro-esclavo con parada por esfuerzo



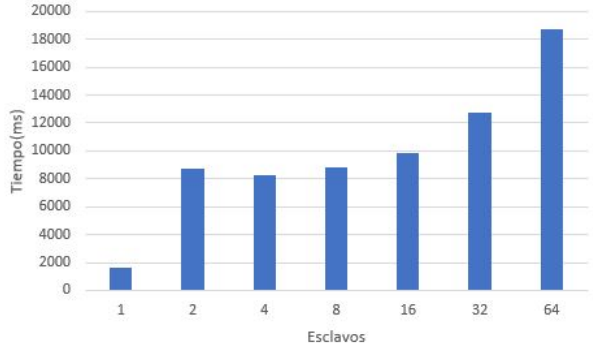
(a) Función Ackley



(b) Función Quadric



(c) Función Rastrigin



(d) Función Rosenbrock

Figura 4.3: Representación gráfica de los tiempos para el modelo maestro-esclavo

necesario el uso de más de un nodo, el coste de las comunicaciones internodo dispara esta sobrecarga. En la figura 4.3, que muestra gráficamente los resultados de tiempos de ejecución, se ve más claramente esta situación.

Estos resultados, si bien son malos, no son sorprendentes, ya que trabajos previos [24] usando el algoritmo *Differential Evolucion* (DE), que tiene características parecidas al PSO, ya demostraban la poca adecuación de Spark a este tipo de estrategias.

4.3 Evaluación del modelo basado en islas

En el modelo basado en islas, cada isla ejecuta el algoritmo secuencial en una población inicial diferente para cada una de ellas. La ejecución de cada isla se puede ver beneficiada por etapas intermedias de comunicación para colaborar entre ellas. De esta forma, el algoritmo paralelo tiene un comportamiento diferente al algoritmo secuencial, al contrario de lo que sucede en el modelo maestro-esclavo. Por eso a la hora de evaluar esta propuesta se han realizado diferentes tipos de pruebas.

En la figura 4.4 se muestran las diferentes pruebas realizadas para la evaluación del modelo basado en islas. Se llevaron a cabo pruebas con una versión cooperativa (iccPSO) y una versión no cooperativa (iscPSO) del modelo de islas para poder evaluar el efecto de la cooperación tanto en la convergencia como en el tiempo de ejecución de la propuesta. Además, se realizaron diferentes experimentos variando el tamaño de la población, para observar el efecto de la distribución entre las islas. Y para todos los casos se llevaron a cabo pruebas utilizando los criterios de parada descritos anteriormente. Estas se repitieron para la implementación secuencial y para la implementación paralela en un número creciente de islas desde 1 hasta 128, lo que nos permite comparar también la escalabilidad de la propuesta paralela. Notar que cada islas es una partición que es ejecutada en Spark por un worker que a su vez se asigna a un nodo físico con dedicación exclusiva.

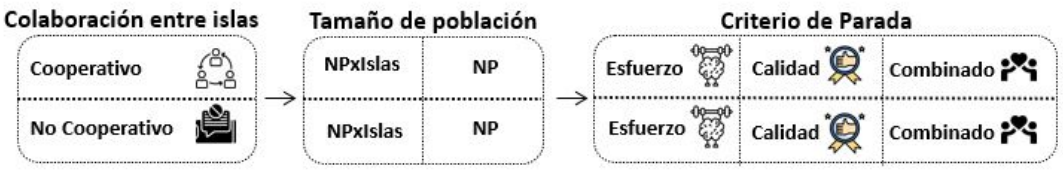


Figura 4.4: Resumen de los experimentos en el modelo basado en Islas

Para explicar mejor los resultados obtenidos y discutir de forma más ordenada las conclusiones que se desprenden de ellos, organizamos esta sección en base al criterio de parada usado en los experimentos.

4.3.1 Pruebas con parada por esfuerzo

Uno de los primeros experimentos que se realizaron tenía como objetivo valorar el efecto que puede tener el tamaño de la población si esta se distribuye al aumentar el número de islas. Los parámetros de ejecución de estas pruebas se pueden ver en la tabla 4.4, donde D es la dimensión del problema, NP es el tamaño de población, EM el esfuerzo máximo en número de iteraciones y C el número de iteraciones de cada isla entre cooperaciones.

Función	D	NP	EM	C
Ackley	10	2400	800	100
Quadric	10	2400	800	100
Rastrigin	20	1200	600	100
Rosenbrock	10	2400	800	100

Tabla 4.4: Parámetros para la evaluación por esfuerzo

Población NP

En este tipo de experimentos, la población global se divide entre el número de islas, es decir, si la población inicial consta de 2400 individuos, con 2 islas cada una tendría 1200, con 4 islas 600, y así sucesivamente. Para ilustrar el comportamiento del algoritmo se pueden ver en la figura 4.5 las curvas de convergencia de varios experimentos realizados con el problema Ackley, que muestran resultados para la ejecución secuencial y para la ejecución paralela del modelo basado en islas con cooperación para 2, 4 y 8 islas.

Se puede ver en estas curvas como el esquema paralelo converge en menos tiempo que el esquema secuencial, ya que cada isla tiene menos trabajo al dividir la población entre ellas. Por otra parte, como los enjambres más pequeños suelen optimizar peor, especialmente en problemas difíciles, se introduce la cooperación entre islas, que en nuestra propuesta consiste en barajar las islas cada cierto número de iteraciones. Estas etapas de cooperación se ven claramente en la figura, en las curvas correspondientes a las ejecuciones paralelas ya que existen puntos en estas donde la evolución empeora , momento en el que se introducen soluciones diferentes en cada isla, situación que no ocurre en la versión secuencial ya que no hay cooperación.

En la tabla 4.5 se muestran los resultados de los experimentos realizados con todos los problemas de prueba. Para cada experimento paralelo cooperativo y no cooperativo se muestra el mejor valor de la función objetivo al que ha llegado la ejecución en el esfuerzo máximo, el tiempo que ha tardado en milisegundos y la aceleración calculada respecto a la ejecución

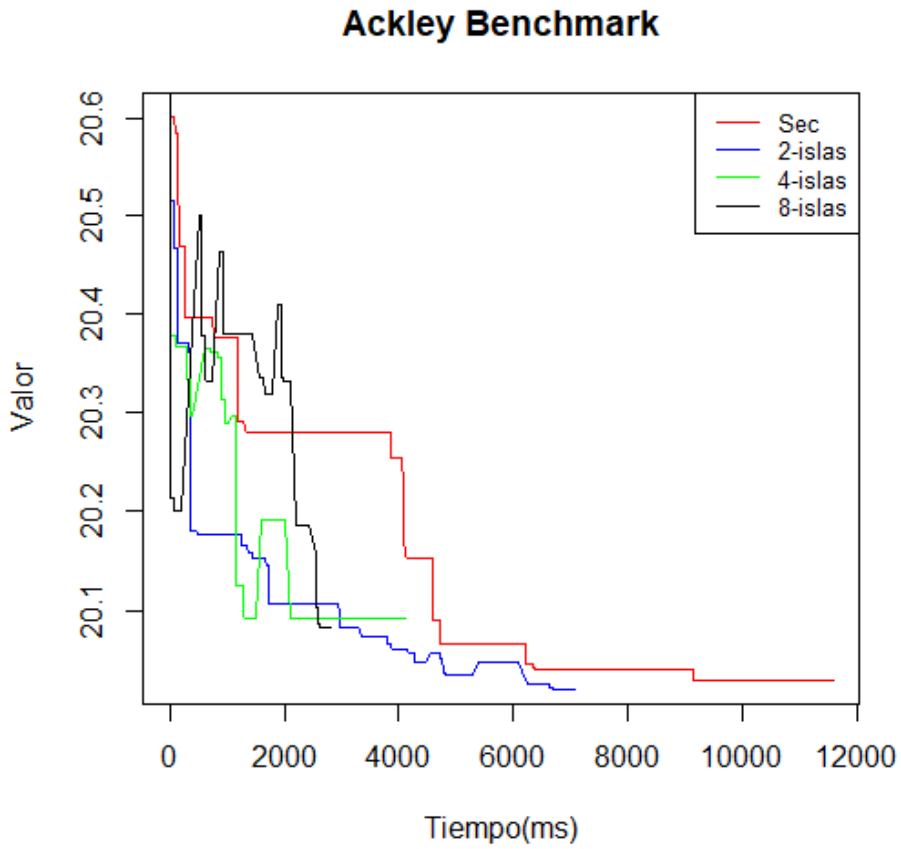


Figura 4.5: Representación de las curvas de convergencia para los esquemas paralelo y secuencial

secuencial como:

$$Sp = T_{sec}/T_i \tag{4.6}$$

Como se realizan 20 ejecuciones de cada experimento, los valores que se muestran son la media y la desviación estándar de esas ejecuciones, y la aceleración se calcula con los valores medios de los tiempos.

Como se puede ver, los resultados para una isla empeoran, en tiempo, en la mayoría de casos, el resultado secuencial, debido a la sobrecarga que introduce Spark y en concreto la función *PartitionBy*, a pesar de que el algoritmo que se ejecuta es idéntico al algoritmo secuencial en Scala. Sin embargo, al aumentar el número de islas, el algoritmo paralelo termina en menos tiempo que el algoritmo secuencial, ya que en cada isla el tamaño del enjambre es menor y, por lo tanto, se realizan menos evaluaciones por isla.

		IscPSO			IccPSO		
	Islas	**Valor**	**Tiempo (ms)**	**Sp**	**Valor**	**Tiempo (ms)**	**Sp**
Ackley	sec.	20.1 ± 0.1	11819 ± 513	-	20.1 ± 0.1	11819 ± 513	-
	1	19.4 ± 3.0	12142 ± 1299	0.9	19.4 ± 3.0	12142 ± 1299	0.9
	2	19.0 ± 4.5	7349 ± 1186	1.6	20.1 ± 0.1	7698 ± 1451	1.5
	4	18.0 ± 6.2	3903 ± 1154	3.0	18.1 ± 6.2	4649 ± 1438	2.5
	8	19.0 ± 4.5	2353 ± 1202	5.0	18.1 ± 6.2	3263 ± 1473	3.7
	16	19.0 ± 7.4	1807 ± 1172	6.5	17.1 ± 7.4	2911 ± 1328	4.2
	32	20.0 ± 0.0	1170 ± 1329	10.1	17.1 ± 7.4	2428 ± 1495	4.9
	64	16.0 ± 8.2	1004 ± 1631	11.8	17.1 ± 7.4	2611 ± 1768	4.6
	128	12.6 ± 9.5	1160 ± 2277	10.2	19.9 ± 0.8	2530 ± 2436	4.8
Quadric	sec.	$(1.7 \pm 7.6)\text{e}{-52}$	11653 ± 687	-	$(1.7 \pm 7.6)\text{e}{-52}$	11653 ± 687	-
	1	$(9.1 \pm 37.9)\text{e}{-54}$	12109 ± 1469	0.9	$(9.1 \pm 37.9)\text{e}{-54}$	12109 ± 1469	0.9
	2	$(1.5 \pm 4.6)\text{e}{-45}$	7684 ± 1079	1.5	$(1.1 \pm 2.9)\text{e}{-48}$	8159 ± 1310	1.4
	4	$(5.9 \pm 10.4)\text{e}{-39}$	3989 ± 1077	2.9	$(1.3 \pm 2.9)\text{e}{-41}$	4837 ± 1332	2.4
	8	$(4.7 \pm 9.4)\text{e}{-31}$	2559 ± 1049	4.6	$(4.2 \pm 1.8)\text{e}{-33}$	3416 ± 1323	3.4
	16	$(8.9 \pm 9.9)\text{e}{-25}$	1823 ± 1014	6.4	$(7.3 \pm 21.7)\text{e}{-27}$	3033 ± 1318	3.8
	32	$(2.0 \pm 2.3)\text{e}{-19}$	1121 ± 1185	10.4	$(6.7 \pm 7.9)\text{e}{-21}$	2436 ± 1501	4.8
	64	$(5.9 \pm 6.1)\text{e}{-15}$	987 ± 1537	11.8	$(1.1 \pm 1.2)\text{e}{-15}$	2545 ± 1870	4.6
	128	$(2.2 \pm 2.6)\text{e}{-10}$	1143 ± 2112	10.2	$(6.7 \pm 7.8)\text{e}{-11}$	2485 ± 2421	4.7
Rastrigin	sec.	6.5 ± 2.2	6384 ± 249	-	6.5 ± 2.2	6384 ± 249	-
	1	7.9 ± 2.8	7344 ± 1100	0.9	7.9 ± 2.8	7344 ± 1100	0.9
	2	6.6 ± 2.4	4228 ± 1017	1.5	8.0 ± 2.9	4492 ± 1201	1.4
	4	9.3 ± 2.3	2332 ± 997	2.7	7.6 ± 2.7	2901 ± 1170	2.2
	8	9.4 ± 2.5	1512 ± 980	4.2	10.7 ± 4.3	2198 ± 1194	2.9
	16	12.1 ± 2.9	1203 ± 1022	5.3	12.3 ± 4.4	1996 ± 1243	3.2
	32	13.5 ± 3.2	826 ± 1123	7.7	16.3 ± 4.9	1769 ± 1508	3.6
	64	17.6 ± 3.9	821 ± 1511	7.8	23.1 ± 5.0	1632 ± 1718	3.9
	128	31.6 ± 4.7	1018 ± 2083	6.3	26.9 ± 7.6	1847 ± 2383	3.5
Rosenbrock	sec.	67.0 ± 88.7	11237 ± 523	-	67.0 ± 88.7	11237 ± 523	-
	1	50.7 ± 74.7	10801 ± 1341	1.0	50.7 ± 74.7	10801 ± 1341	1.0
	2	39.4 ± 42.6	6461 ± 1025	1.7	63.3 ± 73.7	7276 ± 1347	1.5
	4	32.8 ± 34.2	3526 ± 1025	3.2	40.7 ± 46.9	4383 ± 1267	2.6
	8	26.8 ± 18.5	2229 ± 1043	5.0	43.2 ± 29.8	3275 ± 1330	3.4
	16	14.5 ± 20.9	1806 ± 1020	6.2	27.4 ± 28.8	2907 ± 1311	3.9
	32	15.6 ± 12.2	1098 ± 1189	10.2	24.3 ± 20.9	2388 ± 1503	4.7
	64	23.1 ± 16.5	973 ± 1527	11.5	19.5 ± 18.7	2643 ± 1894	4.3
	128	17.6 ± 14.3	1112 ± 2151	10.1	22.1 ± 19.7	2452 ± 2468	4.6

Tabla 4.5: Resultados repartiendo la población NP entre las islas con parada por esfuerzo

La aceleración mejora cuando se aumenta el número de islas, no obstante, llega un punto donde la escalabilidad no es buena, porque la sobrecarga del reparto de datos en Spark crece mientras el tiempo total de ejecución disminuye, por lo que la relación comunicación/computación hace que la eficiencia decrezca.

Además, el algoritmo en cada isla empeora, porque el tamaño del enjambre, al aumentar el número de islas y repartir la población inicial entre ellas, se reduce y tiene menos capacidad de optimizar. Esto se puede observar en los valores finales alcanzados. Algunos problemas de prueba se ven más afectados que otros por esta circunstancia, por ejemplo Quadric y Rastrigin empeoran considerablemente al reducir el enjambre en cada isla, frente a Ackley o Rosenbrock que mantienen valores finales similares con enjambres más pequeños.

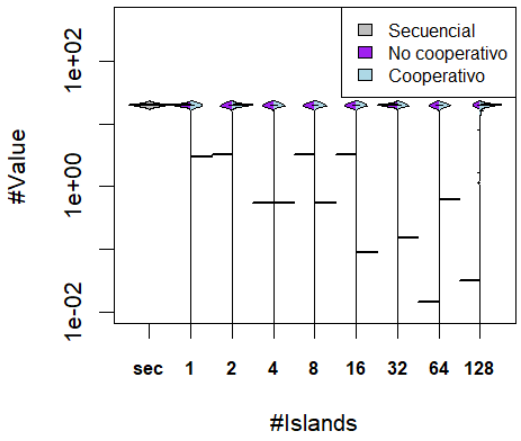
Aunque en la tabla 4.5 se muestran resultados de medias y desviaciones, resulta especialmente interesante ver la distribución de los resultados de las 20 ejecuciones por experimento. Una forma de representar gráficamente esto son los llamados *beanplots*. En la figura 4.6 se muestra esta representación donde en cada *bean* para los experimentos paralelos se muestra a la izquierda la distribución de los resultados de los experimentos no cooperativos y a la derecha la de los cooperativos, para una fácil comparación. En el eje y se encuentra el valor final obtenido por cada uno de los experimentos y en el eje x el número de islas utilizado. La linea horizontal que corta los beans representa la mediana del conjunto de valores obtenido.

En primer lugar, se puede ver como la función Ackley (ver figura 4.6a), que en la tabla obtenía valores medios que disminuían a medida que se aumentaban las islas, en realidad termina la mayoría de sus ejecuciones con valores en torno a 20 (donde existe un mínimo local) pero a medida que aumentan las islas, más ejecuciones en cada experimento consiguen salir de ese mínimo, lo que hace que la media disminuya, como se veía en la tabla.

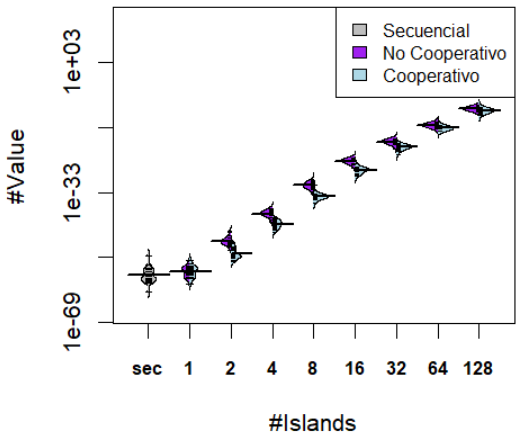
En cuanto a la función Quadric (ver figura 4.6b) se aprecia un empeoramiento del valor a medida que se aumentan las islas ya que al reducir el tamaño del enjambre se empeora la capacidad de optimización para este problema de prueba. De todos modos, se observa como la versión cooperativa optimiza en todos los casos mejor que la versión no cooperativa, es decir, compensa ligeramente esa disminución de la eficiencia del algoritmo con enjambres más pequeños, a través de la cooperación entre las diferentes islas.

Por su lado, la función Rastrigin (ver figura 4.6c) empeora a medida que se incrementan el número de islas, de forma análoga a la función Quadric. Y al igual que aquel, también la versión cooperativa obtiene mejores resultados que la función no cooperativa.

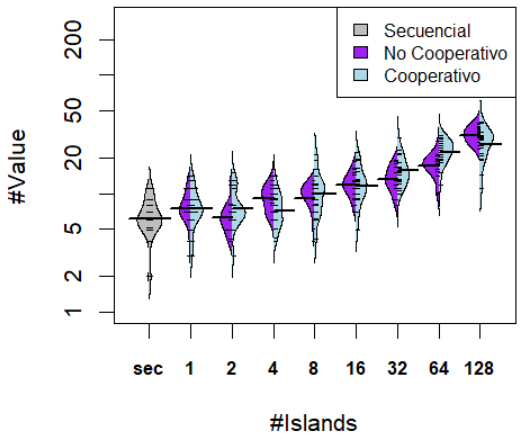
Por último, la función Rosenbrock (ver figura 4.6d) se comporta de forma parecida a la función Ackley, ya que los experimentos no cambian demasiado el valor conseguido a medida que se aumentan los recursos. En este problema de prueba, al contrario de lo que ocurre con Quadric, se puede ver que la versión no cooperativa ofrece un comportamiento mejor que la versión cooperativa.



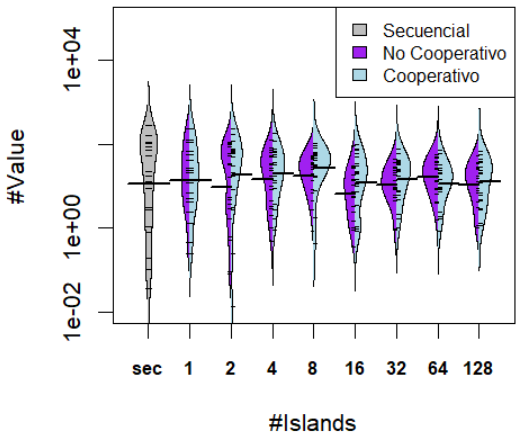
(a) Función Ackley



(b) Función Quadric



(c) Función Rastrigin



(d) Función Rosenbrock

Figura 4.6: Representación gráfica de las distribuciones de los datos resumidos en la tabla 4.5

Población NPxIslas

Para comparar con los resultados obtenidos cuando la población inicial se reparte entre las islas, y por lo tanto el tamaño de las islas decrece con el número de recursos, a continuación se analizaron los resultados de los experimentos donde cada isla tiene el mismo número de individuos que el algoritmo secuencial, es decir, el tamaño del enjambre global crece con el número de islas, siendo NP por el número de islas empleado.

En primer lugar, en la tabla 4.6 se muestran los resultados obtenidos clasificados como en la tabla 4.5 por experimentos cooperativos y no cooperativos. Se pueden ver las medias y desviaciones de los valores obtenidos con el esfuerzo máximo, y el tiempo necesario. También se muestra la sobrecarga, calculada como:

$$Sc = (T_i - T_{sec})/T_{sec} \tag{4.7}$$

en lugar de la aceleración. Cada isla está ejecutando un algoritmo secuencial sobre una población NP, sin embargo, a medida que las islas aumentan hay una sobrecarga debido al reparto de la población inicial y otras comunicaciones, como pueden ser las cooperaciones en la versión cooperativa, que hacen que el tiempo aumente. Se puede ver como los resultados de valor alcanzado mejoran a medida que el número de islas aumenta, y como, en general, la versión cooperativa obtiene mejores valores que la versión no cooperativa. Además, también se puede ver una sobrecarga en las versiones paralelas debido a las comunicaciones en el caso del cooperativo y a la propia sobrecarga de Spark en la versión no cooperativa.

Para ver mejor la distribución de los valores obtenidos se pueden estudiar los *beanplots* representados en la figura 4.7. En esta se representan los resultados de la distribución de los valores finales obtenidos en cada experimento.

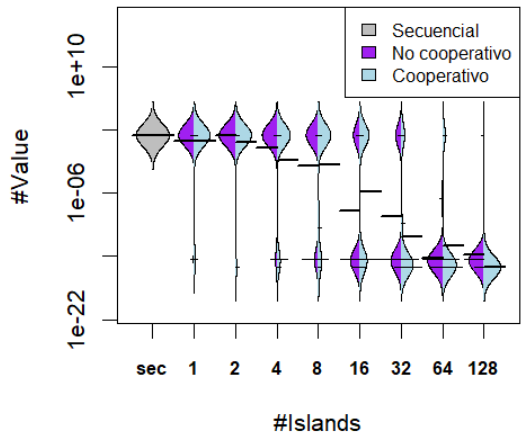
Para la función Ackley (ver figura 4.7a), se puede apreciar la mejora sustancial del valor obtenido por el algoritmo a medida que se incrementan el número de islas. Para este problema, la versión cooperativa mejora a la versión no cooperativa. En la gráfica se aprecia la existencia de mínimos locales (zonas más abultadas en las figuras localizadas en la parte superior e inferior de la gráfica). Como se puede ver, a medida que aumenta el número de islas el algoritmo tiene más probabilidad de salir del primer mínimo local. Además, la versión cooperativa consigue salir más veces de ese mínimo.

Por su parte, la función Quadric (ver figura 4.7b) también obtiene una gran mejora al incrementar el número de islas. En este caso se aprecia como la versión cooperativa tiene un comportamiento significativamente mejor frente a la versión no cooperativa en todas las situaciones, por lo que la inyección de diversidad en este problema de prueba favorece en gran medida su optimización.

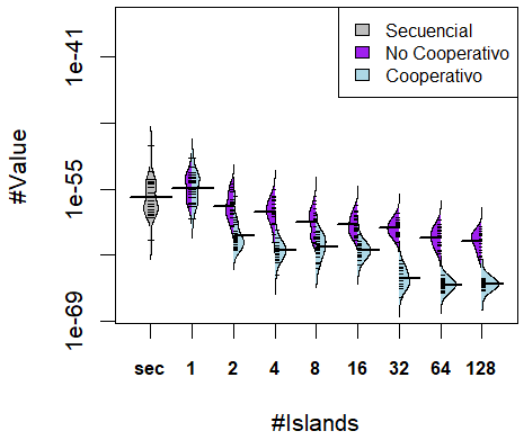
En cuando a la función Rastrigin (ver figura 4.7c) también muestra una importante me-

		IscPSO			IccPSO		
	Islas	**Valor**	**Tiempo (ms)**	**Sc**	**Valor**	**Tiempo (ms)**	**Sc**
Ackley	sec.	20.1 ± 0.1	11819 ± 513	-	20.1 ± 0.1	11819 ± 513	-
	1	19.4 ± 3.0	12142 ± 1299	0.0	19.4 ± 3.0	12142 ± 1299	0.0
	2	20.0 ± 0.0	12150 ± 1205	0.0	19.1 ± 4.5	13661 ± 1440	0.2
	4	18.0 ± 6.2	14757 ± 1340	0.2	16.0 ± 8.2	16615 ± 1607	0.4
	8	15.0 ± 8.9	25513 ± 740	1.2	15.0 ± 8.9	17536 ± 1524	0.5
	16	8.9 ± 10.2	25022 ± 1286	1.1	11.0 ± 10.2	23573 ± 1687	0.9
	32	6.9 ± 9.8	25780 ± 1258	1.2	4.0 ± 8.2	26931 ± 1932	1.3
	64	4.1 ± 7.5	27611 ± 1341	1.3	3.0 ± 7.3	23145 ± 1873	0.9
	128	0.9 ± 4.5	51426 ± 2914	3.4	$(4.4 \pm 0)\text{e}{-16}$	43045 ± 2761	2.6
Quadric	sec.	$(1.7 \pm 7.6)\text{e}{-52}$	11653 ± 687	-	$(1.7 \pm 7.6)\text{e}{-52}$	11653 ± 687	-
	1	$(9.1 \pm 37.9)\text{e}{-54}$	12109 ± 1469	0.0	$(9.1 \pm 37.9)\text{e}{-54}$	12109 ± 1469	0.0
	2	$(3.6 \pm 7.9)\text{e}{-56}$	13087 ± 1225	0.1	$(8.4 \pm 37.1)\text{e}{-58}$	14228 ± 1416	0.2
	4	$(6.6 \pm 18.4)\text{e}{-57}$	14938 ± 1358	0.3	$(3.3 \pm 1.1)\text{e}{-60}$	17164 ± 1654	0.5
	8	$(1.3 \pm 3.8)\text{e}{-57}$	18806 ± 1328	0.6	$(5.4 \pm 8.9)\text{e}{-61}$	18359 ± 1512	0.6
	16	$(1.7 \pm 4.3)\text{e}{-58}$	48526 ± 4738	3.2	$(1.4 \pm 3.4)\text{e}{-61}$	25074 ± 1827	1.2
	32	$(3.1 \pm 6.5)\text{e}{-59}$	37249 ± 1436	2.2	$(2.7 \pm 6.1)\text{e}{-64}$	52457 ± 1750	3.5
	64	$(5.0 \pm 10.9)\text{e}{-59}$	49955 ± 4317	3.3	$(1.4 \pm 1.8)\text{e}{-65}$	52744 ± 1881	3.5
	128	$(1.4 \pm 2.4)\text{e}{-60}$	51662 ± 3271	3.5	$(1.2 \pm 86.2)\text{e}{-65}$	56322 ± 3037	3.8
Rastrigin	sec.	6.5 ± 2.2	6384 ± 249	-	6.5 ± 2.2	6384 ± 249	-
	1	7.9 ± 2.8	7344 ± 1100	0.2	7.9 ± 2.8	7344 ± 1100	0.2
	2	5.7 ± 1.7	7179 ± 1074	0.1	4.9 ± 2.3	7859 ± 1335	0.2
	4	4.5 ± 1.7	8391 ± 1157	0.3	4.4 ± 1.5	8638 ± 1418	0.4
	8	3.7 ± 1.1	8963 ± 1250	0.4	3.7 ± 2.1	10086 ± 1485	0.6
	16	3.1 ± 1.2	11356 ± 1125	0.8	2.0 ± 1.2	12657 ± 1588	0.9
	32	2.6 ± 0.9	20750 ± 354	2.3	1.8 ± 1.7	16472 ± 1738	1.6
	64	2.0 ± 0.7	21815 ± 177	2.4	1.6 ± 1.2	24129 ± 1789	2.8
	128	1.8 ± 0.6	24120 ± 796	2.8	0.9 ± 0.7	26883 ± 2378	3.2
Rosenbrock	sec.	67.0 ± 88.7	11237 ± 523	-	67.0 ± 88.7	11237 ± 523	-
	1	50.7 ± 74.7	10801 ± 1341	0.0	50.7 ± 74.7	10801 ± 1341	0.0
	2	50.6 ± 61.3	11781 ± 1236	0.0	11.8 ± 23.7	12519 ± 1459	0.1
	4	6.8 ± 12.0	14398 ± 1304	0.3	2.2 ± 4.9	16216 ± 1720	0.4
	8	1.1 ± 2.6	19256 ± 826	0.7	2.7 ± 6.4	16480 ± 1508	0.5
	16	0.3 ± 0.9	47021 ± 4681	3.2	0.5 ± 1.1	22623 ± 1629	1.0
	32	0.1 ± 0.1	36879 ± 1931	2.3	1.2 ± 2.4	25465 ± 2230	1.3
	64	0.0 ± 0.0	37589 ± 1601	2.3	0.6 ± 1.2	39226 ± 1886	2.5
	128	0.0 ± 0.0	51180 ± 3425	3.6	0.1 ± 0.1	43862 ± 2569	2.9

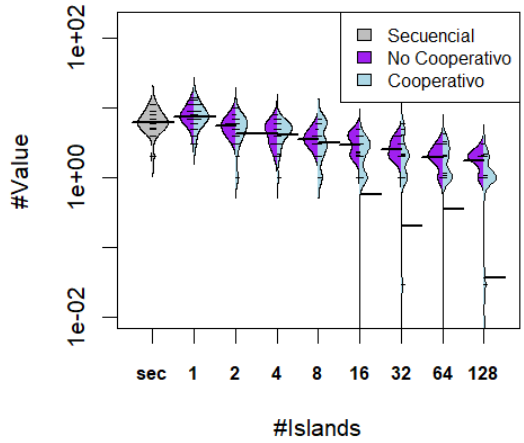
Tabla 4.6: Resultados para experimentos donde cada isla tiene una población NP igual para todas, con parada por esfuerzo



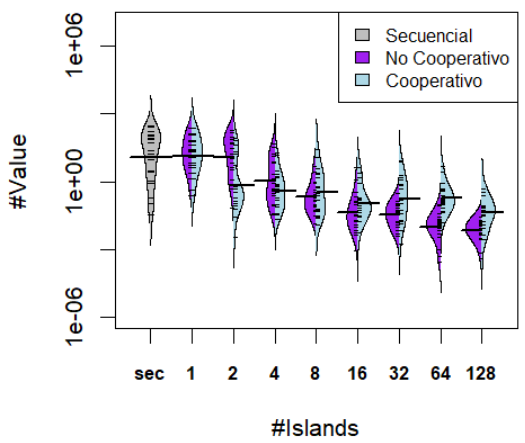
(a) Función Ackley



(b) Función Quadric



(c) Función Rastrigin



(d) Función Rosenbrock

Figura 4.7: Representación gráfica de las distribuciones de los datos resumidos en la tabla 4.6

jora al incrementar las islas utilizadas. La versión cooperativa en este caso mejora siempre a la versión no cooperativa, de hecho, se puede ver como, a partir de las 16 islas, la versión cooperativa en muchas ocasiones consigue salir de los mínimos locales y llegar a converger al valor mínimo de la función, que es 0, cuando tanto la versión no cooperativa como aquellas con pocas islas, no lo consiguieron en ninguna de las ejecuciones de nuestros experimentos.

Por último, la función Rosenbrock (ver figura 4.7d) se aprecia un comportamiento similar a las funciones anteriores en cuanto a la mejora del algoritmo al aumentar el número de islas. Sin embargo, a partir de las 8 islas, la versión cooperativa deja de ser la más eficiente. Hay que tener en cuenta que la estrategia cooperativa propuesta consiste en barajar los individuos de todas las islas y repartirlos de nuevo entre estas. Esta estrategia inyecta diversidad en las islas, al introducir individuos de enjambres diferentes para el siguiente paso del algoritmo. Introducir diversidad en las búsquedas es algo que trabajos previos de la bibliografía han demostrado que puede mejorar la convergencia de algunas metaheurísticas con ciertos problemas difíciles, en concreto, en aquellos que tienden a quedarse estancados en óptimos locales. Sin embargo, esta estrategia no siempre es buena para todos los problemas. En concreto, cuando son muchas islas las que colaboran, mezclar los enjambres cuando en alguna isla el enjambre ya se movía de forma eficaz hacia el valor mínimo, puede ralentizar al algoritmo. Esto es lo que sucede en este caso.

Como se aprecia en los experimentos comentados anteriormente, los experimentos donde la población entre islas se mantiene constante al tamaño inicial de la ejecución secuencial logran valores mucho mejores, ya que exploran un espacio de búsqueda mayor, gracias a los recursos paralelos distribuidos, y consiguen, en general, una mejor convergencia gracias a la cooperación. Además, estos experimentos demuestran una mejor escalabilidad, ya que los resultados mejoran a medida que se aumenta el número de islas, al contrario que los experimentos donde la población inicial secuencial se reparte entre las islas. En conclusión, para problemas difíciles es conveniente usar todos los recursos computacionales disponibles usando el modelo basado en islas cooperativo y utilizando como tamaño de población en cada isla el tamaño adecuado a los problemas secuenciales, mientras que para problemas fáciles podemos intentar ejecutarlos con experimentos donde la población secuencial inicial se reparta entre las islas y de esta forma obtener los resultados más rápidamente.

4.3.2 Prueba con parada por calidad

En estos experimentos se establecen dos criterios de parada comentados anteriormente: uno, el que nos interesa, por calidad de la solución obtenida, y otro para limitar el tiempo máximo de ejecución del algoritmo en estas pruebas, por esfuerzo máximo. El algoritmo finaliza cuando se cumple alguno de ellos. Los parámetros para la ejecución de estas pruebas se pueden ver en la tabla 4.7, donde NP es el tamaño del enjambre para cada isla, D es la dimensión

Función	NP	D	EM	VO	C
Ackley	2400	10	800	1e−3	100
Quadric	2400	20	800	5e−11	100
Rastrigin	1200	20	600	5e−1	100
Rosenbrock	2400	10	800	5e−2	100

Tabla 4.7: Parámetros para la evaluación con parada por calidad de la solución

del problema, EM es el esfuerzo máximo expresado en iteraciones, VO es el valor objetivo establecido y C es el número de iteraciones de cada isla entre cooperaciones. Además, en estos experimentos cada isla tiene el mismo número de individuos que el algoritmo secuencial, por lo tanto, el tamaño del enjambre global es NP por el número de islas empleado.

En la tabla 4.8 se muestran los experimentos divididos en versión cooperativa y no cooperativa, como en las tablas anteriores. Se pueden ver las medias y desviaciones de los valores obtenidos y el tiempo necesario para alcanzar el valor. Se muestra también la media de las iteraciones necesarias para finalizar la prueba y el porcentaje de experimentos que finalizan por cumplir con el criterio de calidad ($\%h$). En esta tabla se ve como a medida que el número de islas aumenta también lo hace el porcentaje de experimentos que acaban por calidad, y como, menos con Rosenbrock, la versión cooperativa supone una gran mejora frente a la versión no cooperativa. Como se comentó en los experimentos anteriores, hay algoritmos que no favorecen la diversidad en la búsqueda de la solución, como es el caso de Rosenbrock. Además, al aumentar el número de experimentos que acaban por calidad a medida que se incrementan las islas, se reduce el tiempo de sobrecarga impuesto al usar un gran número de islas.

Por otro lado, en la figura 4.8 se puede ver la distribución de los valores obtenidos en cada uno de los experimentos realizados. Para empezar, en la función Ackley (ver figura 4.8a), se aprecian perfectamente los óptimos locales de este problema de prueba, y como se superan a medida que se incrementan los recursos utilizados. Además, se puede ver como la versión cooperativa mejora a la versión no cooperativa en la mayoría de los experimentos.

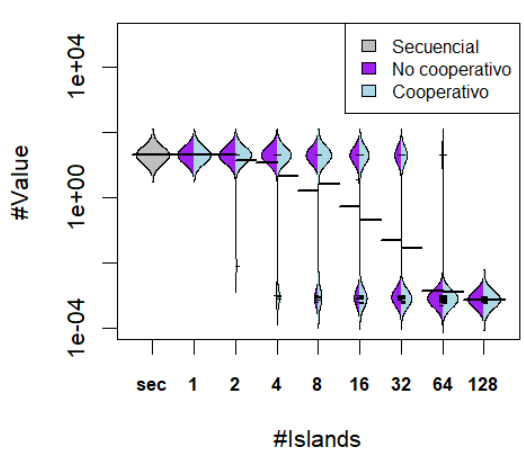
En la función Quadric (ver figura 4.8b) se puede ver la gran dispersión en los resultados obtenidos para este problema de prueba y como también se mejoran los valores a medida que se aumenta el número de islas utilizadas. Además, con este problema de prueba se ve todavía de forma más clara como la colaboración entre islas mejora la optimización del algoritmo.

Por su lado, la función Rastrigin (ver figura 4.8c) manifiesta un comportamiento similar al problema de prueba anterior, donde el valor obtenido va mejorando cuando aumentan los recursos utilizados. También en este caso la versión cooperativa supone una gran mejora frente a la versión donde las islas se ejecutan de forma independiente.

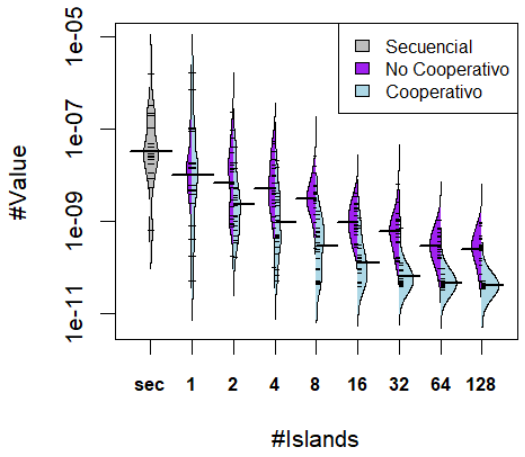
Por último, la función Rosenbrock (ver figura 4.8d) presenta un comportamiento diferente a los problemas de prueba anteriores, ya que, a pesar de que el valor obtenido mejora también

		IscPSO				IccPSO			
	Islas	**Valor**	**Tiempo (ms)**	**Iter**	**%h**	**Valor**	**Tiempo (ms)**	**Iter**	**%h**
Ackley	sec.	20.1 ± 0.1	12096 ± 555	800	0	20.1 ± 0.1	12096 ± 555	800	0
	1	20.1 ± 0.1	13459 ± 1268	800	0	20.1 ± 0.1	13459 ± 1268	800	0
	2	20.0 ± 0.1	13709 ± 1269	800	0	19.1 ± 4.5	15139 ± 1450	800	0
	4	19.0 ± 4.5	14097 ± 2932	769	5	17.0 ± 7.3	13106 ± 5325	691	15
	8	15.0 ± 8.9	13057 ± 5819	640	25	16.0 ± 8.2	14268 ± 6773	656	20
	16	12.2 ± 9.9	14470 ± 7926	584	35	11.0 ± 10.2	14231 ± 10122	509	45
	32	8.0 ± 10.1	13545 ± 9896	442	60	7.0 ± 9.8	12895 ± 12397	382	65
	64	1.0 ± 4.5	14097 ± 2932	185	95	1.0 ± 4.5	5662 ± 8426	162	95
	128	0.0 ± 0.0	4268 ± 547	157	100	0.0 ± 0.0	2503 ± 336	92	100
Quadric	sec.	$(1.5 \pm 3.5)\text{e}{-07}$	16818 ± 327	800	0	$(1.5 \pm 3.5)\text{e}{-07}$	16818 ± 327	800	0
	1	$(1.3 \pm 3.8)\text{e}{-07}$	24270 ± 1438	799	5	$(1.3 \pm 3.8)\text{e}{-07}$	24270 ± 1438	799	5
	2	$(2.8 \pm 5.4)\text{e}{-08}$	25426 ± 1029	800	0	$(7.1 \pm 12.1)\text{e}{-09}$	26020 ± 1434	800	0
	4	$(1.2 \pm 1.5)\text{e}{-08}$	23816 ± 1021	800	0	$(3.1 \pm 5.4)\text{e}{-09}$	23542 ± 1506	800	5
	8	$(5.1 \pm 6.1)\text{e}{-09}$	26536 ± 1102	800	0	$(5.1 \pm 5.2)\text{e}{-10}$	26115 ± 1562	796	20
	16	$(1.3 \pm 1.0)\text{e}{-09}$	33279 ± 1114	800	0	$(2.3 \pm 3.1)\text{e}{-10}$	33120 ± 2109	793	35
	32	$(9.9 \pm 13.9)\text{e}{-10}$	44145 ± 1144	800	0	$(9.2 \pm 14.8)\text{e}{-11}$	56314 ± 2727	785	45
	64	$(3.6 \pm 2.5)\text{e}{-10}$	52929 ± 6162	798	5	$(4.7 \pm 1.4)\text{e}{-11}$	59121 ± 2537	781	85
	128	$(3.3 \pm 2.5)\text{e}{-10}$	54793 ± 9079	800	10	$(39.9 \pm 3.8)\text{e}{-12}$	62474 ± 2694	764	100
Rastrigin	sec.	8.4 ± 2.5	6490 ± 216	600	0	8.4 ± 2.5	6490 ± 216	600	0
	1	7.3 ± 2.6	8198 ± 1135	600	0	7.3 ± 2.6	8198 ± 1135	600	0
	2	7.5 ± 2.1	8160 ± 1015	600	0	5.9 ± 2.3	8692 ± 1246	600	0
	4	4.5 ± 1.4	9031 ± 1094	600	0	4.1 ± 1.8	9706 ± 1441	600	0
	8	3.8 ± 1.0	9804 ± 1257	600	0	3.0 ± 1.7	12040 ± 1395	598	5
	16	3.0 ± 0.9	15622 ± 246	600	0	2.1 ± 1.1	17624 ± 2069	599	5
	32	2.9 ± 0.9	21063 ± 348	600	0	2.4 ± 1.9	15040 ± 2362	587	25
	64	1.9 ± 0.6	21600 ± 307	600	0	1.2 ± 1.2	23217 ± 2874	580	40
	128	1.6 ± 0.7	22170 ± 2696	598	5	1.3 ± 1.1	26609 ± 3086	583	40
Rosenbrock	sec.	76.2 ± 100.5	10740 ± 446	800	0	76.2 ± 100.5	10740 ± 446	800	0
	1	42.5 ± 70.4	12747 ± 1074	800	0	42.5 ± 70.4	12747 ± 1074	800	0
	2	31.7 ± 51.1	12345 ± 2233	768	10	38.3 ± 75.8	13659 ± 1645	781	15
	4	3.1 ± 5.8	12464 ± 2674	737	25	5.4 ± 13.8	14797 ± 593	786	10
	8	0.9 ± 1.9	15861 ± 3151	750	15	4.8 ± 8.7	16838 ± 3160	756	20
	16	0.1 ± 0.1	29668 ± 6718	682	55	0.6 ± 1.9	19511 ± 4080	741	35
	32	0.1 ± 0.0	35346 ± 1931	737	65	0.7 ± 1.3	24563 ± 4244	751	20
	64	0.0 ± 0.0	21848 ± 12666	656	85	0.1 ± 0.1	42667 ± 9501	674	55
	128	0.0 ± 0.0	13853 ± 2390	554	100	0.1 ± 0.1	47942 ± 10192	663	65

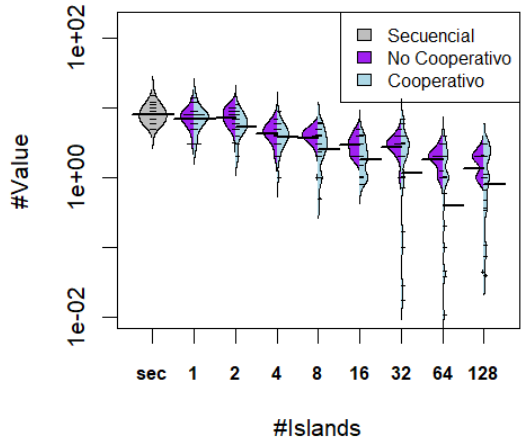
Tabla 4.8: Resultados para experimentos con población igual a NP en cada isla, con parada combinada



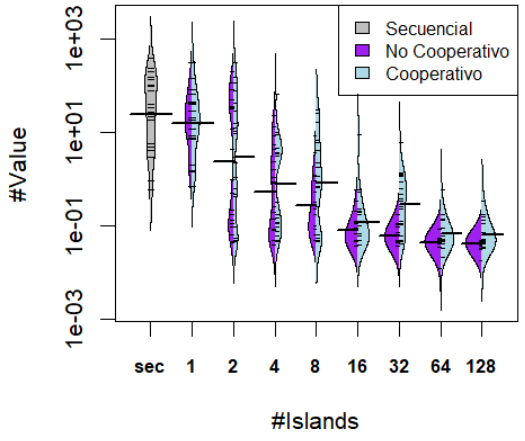
(a) Función Ackley



(b) Función Quadric



(c) Función Rastrigin



(d) Función Rosenbrock

Figura 4.8: Representación gráfica de las distribuciones de los datos resumidos en la tabla 4.8

a medida que aumentan el número de islas, en este caso la versión no cooperativa obtiene mejores valores que la versión cooperativa en todos los experimentos realizados.

4.4 Comparación entre plataformas

En esta sección se comparan dos plataformas diferentes utilizadas para ejecutar los experimentos, por un lado el clúster privado Pluton, donde se obtienen los resultados para el análisis de cara a la memoria, por otro, un cloud público en el CESGA, denominado Nebula.

En la tabla 4.9 se muestran los resultados de tiempo y valor alcanzado para las versiones cooperativas del modelo basado en islas sobre el algoritmo PSO, ejecutadas en las plataformas comentadas anteriormente. La sobrecarga mostrada en esta tabla se calcula en comparación con el tiempo secuencial en la misma plataforma:

$$Sc = (T_i - T_{sec})/T_{sec}$$

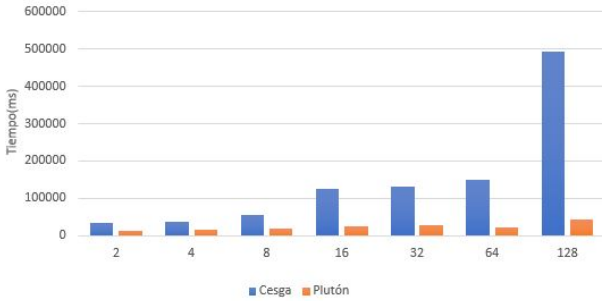
En estos resultados se puede ver como el valor alcanzado por el algoritmo no cambia de una plataforma a otra para los mismos experimentos, ya que el número de recursos es el mismo. Sin embargo, el tiempo de ejecución es muy diferente en ambas plataformas, necesitando para los mismos experimentos más tiempo de ejecución en Nebula. Aunque el tiempo absoluto es menor en Pluton, la sobrecarga que existe si comparamos las ejecuciones paralelas con la secuencial en la misma plataforma es similar en los dos casos cuando los experimentos no utilizan muchos recursos. En este caso, recordemos, cada isla es una partición que se asigna a un core físico. Cuando el número de islas aumenta, es necesario usar más de un nodo, y las comunicaciones internodo penalizan la ejecución. Si bien la penalización ya aumentaba en Pluton, ahora en los experimentos de Nebula se incrementa de forma notable, destacando especialmente en las pruebas realizadas con 128 islas.

En la figura 4.9 se muestran los tiempos de ejecución para cada una de las plataformas y el diferente número de islas utilizadas. En estos gráficos se aprecia la gran cantidad de tiempo que necesita Nebula para obtener los resultados. Esto se debe, principalmente, a la virtualización de los recursos, especialmente en el caso de las comunicaciones, que supone una gran penalización. Otro aspecto a considerar para comparar los tiempos es la diferencia entre los procesadores usados en ambas plataformas. La diferencia de tiempos en la versión secuencial se explica por el rendimiento de los procesadores en cada plataforma. Además, el uso no exclusivo de los recursos en Nebula afecta negativamente a los tiempos, especialmente al aumentar mucho el número de recursos.

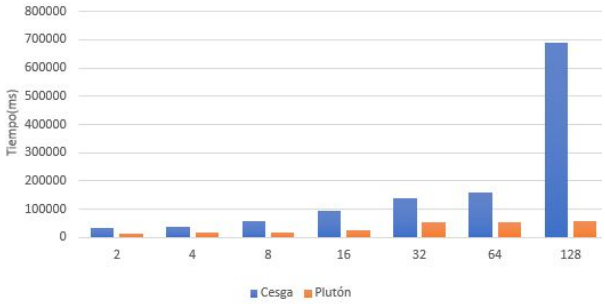
En conclusión, si bien el comportamiento, en cuanto a convergencia del algoritmo, es similar, los tiempos de ejecución son claramente mejores en Pluton que en Nebula. Sin embargo, ante la imposibilidad de acceder a un clúster privado, la opción de ejecutar en un cloud pú-

		Nebula			Pluton		
	Islas	**Valor**	**Tiempo (ms)**	**Sc**	**Valor**	**Tiempo (ms)**	**Sc**
Ackley	sec.	18.6 ± 4.8	30024 ± 1586	-	20.1 ± 0.1	11819 ± 513	-
	1	20.1 ± 0.1	23656 ± 2064	0.0	19.4 ± 3.0	12142 ± 1299	0.0
	2	19.1 ± 4.5	33669 ± 3334	0.1	19.1 ± 4.5	13661 ± 1440	0.2
	4	19.0 ± 4.5	36750 ± 3348	0.2	16.0 ± 8.2	16615 ± 1607	0.4
	8	19.0 ± 4.5	55369 ± 3492	0.8	15.0 ± 8.9	17536 ± 1524	0.5
	16	11.0 ± 10.2	123920 ± 5229	3.1	11.0 ± 10.2	23573 ± 1687	0.9
	32	3.2 ± 7.3	132000 ± 4815	3.4	4.0 ± 8.2	26931 ± 1932	1.3
	64	1.5 ± 4.8	150173 ± 6084	4.0	3.0 ± 7.3	23145 ± 1873	0.9
	128	(4.4 ± 0)e-16	493576 ± 113081	15.4	(4.4 ± 0)e-16	43045 ± 2761	2.6
Quadric	sec.	(5.2 ± 10.6)e-54	30047 ± 1857	-	(1.7 ± 7.6)e-52	11653 ± 687	-
	1	(1.3 ± 5.0)e-53	25736 ± 2079	0.0	(9.1 ± 37.9)e-54	12109 ± 1469	0.0
	2	(1.2 ± 2.5)e-58	33683 ± 2980	0.1	(8.4 ± 37.1)e-58	14228 ± 1416	0.2
	4	(7.7 ± 15.6)e-61	37715 ± 3436	0.3	(3.3 ± 1.1)e-60	17164 ± 1654	0.5
	8	(1.9 ± 3.1)e-62	58487 ± 3045	0.9	(5.4 ± 8.9)e-61	18359 ± 1512	0.6
	16	(1.8 ± 3.3)e-63	95513 ± 5105	2.2	(1.4 ± 3.4)e-61	25074 ± 1827	1.2
	32	(1.1 ± 1.4)e-64	140546 ± 5110	3.7	(2.7 ± 6.1)e-64	52457 ± 1750	3.5
	64	(4.2 ± 10.2)e-65	158534 ± 5868	4.3	(1.4 ± 1.8)e-65	52744 ± 1881	3.5
	128	(4.0 ± 3.2)e-66	691489 ± 307162	22.0	(1.2 ± 86.2)e-65	56322 ± 3037	3.8
Rastrigin	sec.	7.1 ± 2.6	13725 ± 310	-	6.5 ± 2.2	6384 ± 249	-
	1	7.1 ± 3.6	14176 ± 1704	0.0	7.9 ± 2.8	7344 ± 1100	0.2
	2	5.8 ± 2.6	18572 ± 2738	0.3	4.9 ± 2.3	7859 ± 1335	0.2
	4	3.9 ± 1.7	21007 ± 2977	0.5	4.4 ± 1.5	8638 ± 1418	0.4
	8	2.9 ± 1.5	29924 ± 2780	1.2	3.7 ± 2.1	10086 ± 1485	0.6
	16	3.1 ± 1.3	54116 ± 3548	2.9	2.0 ± 1.2	12657 ± 1588	0.9
	32	1.8 ± 1.5	57276 ± 4413	3.2	1.8 ± 1.7	16472 ± 1738	1.6
	64	1.3 ± 1.4	64053 ± 4494	3.7	1.6 ± 1.2	24129 ± 1789	2.8
	128	1.1 ± 0.8	88978 ± 9986	5.5	0.9 ± 0.7	26883 ± 2378	3.2
Rosenbrock	sec.	45.4 ± 53.5	27672 ± 1408	-	67.0 ± 88.7	11237 ± 523	-
	1	90.1 ± 143.8	22517 ± 2111	0.0	50.7 ± 74.7	10801 ± 1341	0.0
	2	23.2 ± 57.6	29096 ± 3696	0.1	11.8 ± 23.7	12519 ± 1459	0.1
	4	14.0 ± 41.3	34131 ± 3482	0.2	2.2 ± 4.9	16216 ± 1720	0.4
	8	3.8 ± 7.4	54990 ± 4054	0.9	2.7 ± 6.4	16480 ± 1508	0.5
	16	0.6 ± 1.1	85563 ± 4784	2.1	0.5 ± 1.1	22623 ± 1629	1.0
	32	0.1 ± 0.2	130604 ± 4883	3.7	1.2 ± 2.4	25465 ± 2230	1.3
	64	0.0 ± 0.0	149644 ± 4918	4.4	0.6 ± 1.2	39226 ± 1886	2.5
	128	0.0 ± 0.0	393933 ± 28390	13.2	0.1 ± 0.1	43862 ± 2569	2.9

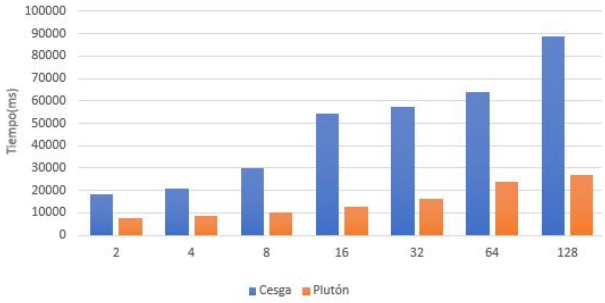
Tabla 4.9: Comparación de resultados entre Nebula y Pluton



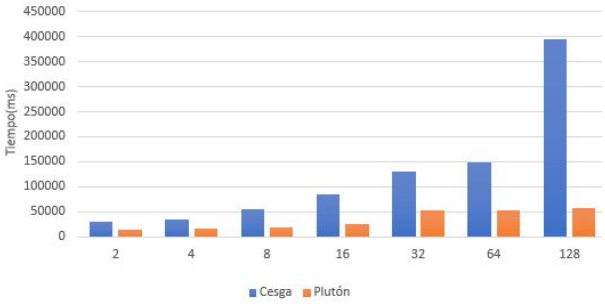
(a) Función Ackley



(b) Función Quadric



(c) Función Rastrigin



(d) Función Rosenbrock

Figura 4.9: Representación gráfica de los tiempos de ejecución resumidos en la tabla 4.9

blico resulta muy atractiva e incluso conveniente para un número de recursos razonable, que dependerá, lógicamente, del problema entre manos.

Capítulo 5

Conclusiones y trabajo futuro

En este apartado se describen las conclusiones sacadas tras la realización de este trabajo desde dos puntos de vista diferentes: en el plano académico, el conocimiento adquirido gracias al trabajo realizado y, en el plano de investigación, las conclusiones extraídas de la paralelización del algoritmo PSO.

En el plano académico

En primer lugar, este proyecto me ha dado la oportunidad de adquirir nuevos conocimientos, tanto teóricos como prácticos, en múltiples ámbitos. He aprendido un nuevo lenguaje de programación, Scala, y después de esta experiencia, entiendo mejor los lenguajes funcionales, ya que en el transcurso del grado se suele trabajar más con lenguajes imperativos como Java, C o C#.

Por otro lado, he tenido la oportunidad de trabajar con un framework para computación distribuida como es Spark. Tuve que aprender su arquitectura y la notación empleada, así como a gestionar las operaciones para optimizar el tiempo empleado en cada una de ellas.

Además, también he tenido la oportunidad de trabajar en diferentes infraestructuras, en concreto en un clúster y un cloud público. En el caso del clúster, Pluton, tuve que aprender sobre su configuración y la forma de usar las colas. En el caso del cloud del CESGA fue necesario desplegar manualmente el clúster en las máquinas virtuales. Todas estas tareas han supuesto un reto para mí.

Por último, el trabajo llevado a cabo en el proyecto me ha permitido introducirme en los métodos de optimización, más en particular en las metaheurísticas, y especialmente en su paralelización.

En el plano de investigación

Desde el punto de vista de los resultados obtenidos de la paralelización del algoritmo PSO usando Spark, se pueden comentar conclusiones muy interesantes. En primer lugar, en

modelos como el maestro-esclavo donde se requiera un gran número de comunicaciones por cada iteración del algoritmo, la relación computación/comunicación hace que la paralelización no sea rentable, ya que la sobrecarga de Spark a la hora de distribuir los datos no compensa la computación que posteriormente se realiza con ellos. Por este motivo, los resultados obtenidos con este modelo no son los deseados.

En cambio, modelos donde se mejora la relación computación/comunicación serán más rentables, como es el caso del modelo en islas. Este modelo cambia las propiedades del algoritmo secuencial, lo que podría conseguir una convergencia más rápida gracias a la cooperación entre las distintas islas que están ejecutando el algoritmo. En este trabajo se ha propuesto una estrategia de cooperación que consiste en barajar cada cierto tiempo los enjambres de las diferentes islas entre sí, de modo que se inyecte diversidad en las búsquedas, ayudando a que el algoritmo pueda salir de mínimos locales en casos difíciles. La cooperación entre islas y la inyección de diversidad de la estrategia propuesta favorece la convergencia en la mayoría de los problemas de prueba, ya que se obtienen mejores resultados tanto en la parada por esfuerzo como en la parada por calidad.

En este trabajo se ha realizado una evaluación exhaustiva de las diferentes estrategias. En el caso del modelo de islas se han realizado experimentos con diferente tamaño de población global. Se han encontrado diferencias entre los experimentos que reparten el tamaño inicial del enjambre entre las islas y los experimentos en los que cada isla mantiene el tamaño del enjambre usado en la ejecución secuencial. En el segundo caso se obtienen mejores valores ya que el espacio que se explora es mayor, mientras que, en el primer caso se reduce en gran medida el tiempo empleado porque el tamaño del enjambre va disminuyendo en cada isla a medida que el número de estas aumenta, aunque tamaños de enjambre demasiado pequeños puedan afectar significativamente a la calidad de la solución alcanzada. En conclusión, para problemas difíciles, donde interesa mejorar la calidad de la solución, se recomendaría utilizar todos los recursos de computación disponibles lanzando islas con poblaciones estándar aleatorias y haciendo que a medida que progresen en la búsqueda cooperen entre sí. Mientras que, para problemas fáciles repartir la población inicial entre las islas, podría ser mejor solución si lo que buscamos es minimizar el tiempo de resolución.

Por último, en los experimentos realizados en este trabajo hemos comparado dos plataformas diferentes, un clúster privado y un cloud público y hemos obtenido diferentes conclusiones tras las pruebas realizadas. Los resultados obtenidos muestran que las ejecuciones en el cloud sufren una penalización en los tiempos debido a la virtualización de los recursos, no obstante, este tipo de clouds suelen gozar de una mayor accesibilidad que los clúster privados como Pluton, por lo tanto, cada plataforma tiene sus ventajas e inconvenientes.

5.1 Trabajo Futuro

Este proyecto supone el comienzo de una línea de investigación que se podría completar con varios trabajos futuros. En primer lugar, sería muy interesante implementar otras estrategias de cooperación entre islas, como por ejemplo, compartir el mejor individuo a nivel global entre todas las islas, o barajar solo una parte de la población en cada isla, y evaluar para que tipo de problemas sería más conveniente cada una de las estrategias. También se podrían incluir tácticas adicionales para facilitar la salida de óptimos locales, como reiniciar parte de la población de cada isla una vez que se detecte que el algoritmo no está optimizando.

Por otra parte, como hemos visto durante el desarrollo de este proyecto, en el caso de las versiones básicas de las metaheurísticas, los parámetros de configuración del algoritmo, como puede ser el tamaño de la población, o en el caso del PSO los coeficientes que emplea el algoritmo en el movimiento del enjambre, los elige el usuario. Sin embargo, su elección no es fácil y su eficiencia depende de cada problema en cuestión. Existen trabajos en la bibliografía que estudian qué parámetros resultan más eficientes para diferentes grupos de problemas. También hay otros trabajos que proponen técnicas de *autotuning* al comienzo de la ejecución del código. Sin embargo, no encontramos trabajos que hayan estudiado el impacto de la selección de estos parámetros en versiones paralelas. Sería interesante como trabajo futuro abordar este estudio.

Lista de acrónimos

ACO *Ant Colony Optimization*

API *Application Programming Interface*

CESGA *Centro de Supercomputación de Galicia*

DE *Differential Evolution*

ES *Evolution Strategy*

GAs *Genetic Algorithms*

GRASP *Greedy Randomized Adaptative Search Procedure*

HPC *High Performance Computing*

IaaS *Infrastructure as a Service*

PaaS *Platform as a Service*

PSO *Particle Swarn Optimization*

RDD *Resilient Distributed Datasets*

SA *Simulated Annealing*

SS *Scatter Search*

SaaS *Software as a Service*

SQL *Structured Query Language*

TI *Tecnologías de la Información*

TS *Tabu Search*

Glosario

Apache Spark Motor de procesamiento *open-source* para sistemas distribuidos que permite procesar grandes conjuntos de datos sobre un conjunto de recursos computacionales proporcionando escalabilidad y tolerancia a fallos.

Cloud Conjunto de tecnologías que permiten ofrecer servicios de computación a través de la red.

Computación de altas prestaciones Consiste en utilizar el procesamiento paralelo para resolver problemas complejos y costosos en ciencia y/o ingeniería.

Hadoop Framework *open-source* para programar aplicaciones distribuidas que gestionan grandes cantidades de datos.

Map Reduce Paradigma que se basa en ejecutar varias instancias de un par de funciones de mapeo y reducción en paralelo.

Metaheurísticas Método de optimización que usa procedimientos genéricos de una manera que se espera eficiente.

Multimodal Se utiliza para definir funciones que poseen un gran número de óptimos locales.

Optimización de enjambre de partículas Algoritmo de optimización basado en población cuyo funcionamiento está inspirado en el comportamiento de los enjambres de pájaros o de los bancos de peces.

Optimización Global Rama de la matemática aplicada que se ocupa de la optimización de una función o funciones de acuerdo a diferentes criterios.

Scala Lenguaje multi-paradigma de alto nivel que combina la orientación a objetos con la programación funcional.

Sistemas distribuidos Sistemas en los que los componentes hardware o software están unidos mediante una red.

Bibliografía

[1] M. Cohn, "An introduction scrum," 2003. [En línea]. Disponible en: https://www.mountaingoatsoftware.com/uploads/presentations/Introduction-Scrum-SQuAD-2003.pdf

[2] K. Schwaber and J. Sutherland, *The Scrum Guide*, 2016.

[3] MásQueNegocio, "Desarrollar una app en españa cuesta la mitad que en alemania," 2017. [En línea]. Disponible en: https://www.masquenegocio.com/2017/07/19/desarrollar-app-espana/

[4] P. M. Pardalos and H. E. Romeijn, *Handbook of Global Optimization.* Springer Science and Business Media Dordrecht, 2013.

[5] E. M. T. Hendrix, *Global Optimization at work*, 1998.

[6] E.-G. Talbi, *Metaheuristics: from design to implementation, volume 74.* John Wiley and Sons, 2009.

[7] J. Kennedy and R. Eberhart, *Particle Swarn Optimization*, 1995, vol. 4.

[8] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers.* CRC Press, 2010.

[9] V. Kumar, A. Grama, A. Gupta, and G.Karypis, *Introduction to parallel computing: design and analysis of algorithms.* Benjamin/CummingsRedwood City, CA, 1994, vol. 400.

[10] R. Buyya, J. Broberg, and A. M. Goscinski, *Cloud computing: Principles and paradigms.* John Wiley and Sons, 2010, vol. 87.

[11] K. Hwang, J. Dongarra, and G. C. Fox, *Distributed and Cloud Computing:From Parallel Processing to the Internet of Things.* Morgan Kaufmann, 2013.

[12] "Cloud computing," 2020. [En línea]. Disponible en: https://networkencyclopedia.com/cloud-computing/

[13] J. Dean and S. Ghemawat, *Mapreduce: Simplified data processing on large clusters.* Communication ACM, 2008.

[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing.* Usenix Association, 2012.

[15] *Hadoop in Action.* Manning Publications Co., 2010.

[16] "Página oficial de scala," 2020. [En línea]. Disponible en: https://docs.scala-lang.org/tour/tour-of-scala.html

[17] N. Hansen, A. Auger, S. Finck, and R. Ros, *Real-parameter black-box optimization benchmarking 2010: Experimental setup. Technical Reports Rapports de Recherche RR-6828.* Institut National de Recherche en Informatique et en Automatique (INRIA), 2009.

[18] S. Surjanovic and D. Bingham, "Ackley function," 2013. [En línea]. Disponible en: https://www.sfu.ca/~ssurjano/ackley.html

[19] ——, "Rastrigin function," 2013. [En línea]. Disponible en: https://www.sfu.ca/~ssurjano/rastr.html

[20] ——, "Rosenbrock function," 2013. [En línea]. Disponible en: https://www.sfu.ca/~ssurjano/rosen.html

[21] R. R. Expósito, "Pluton cluster," 2020. [En línea]. Disponible en: http://pluton.des.udc.es/

[22] "Opennebula," 2014. [En línea]. Disponible en: https://archives.opennebula.org/documentation:archives:rel2.0

[23] J. A. Rodrigo, "Optimización por enjambre de partículas," 2019. [En línea]. Disponible en: https://rpubs.com/Joaquin_AR/475474

[24] D. Teijeiro, X. C. Pardo, P. González, J. R. Banga, and R. Doallo, "Implementing parallel differential evolution on Spark," ser. Applications of Evolutionary Computation. Lecture Notes in Computer Science, Vol. 9598. Springer, 2016, pp. 75–90.