Universidad Nacional de San Antonio Abad del Cusco

Escuela de Postgrado

Maestría en Ciencias Mención Informática

Paralelización del Algoritmo Basado en el Comportamiento Social de las Arañas para Clustering

Tesis Presentado Por: Edwin Alvarez Mamani

Para optar al grado académico de
Maestro en Ciencias Mención Informática

Asesor: Dr. Lauro Enciso Rodas

Coasesor: Dr. José L. Soncco Álvarez

Tesis financiada por la UNSAAC

Cusco-Perú 2021

Dedico esta tesis a mis padres Práxides y Miguel, a mis hermanas Gloria, Rocío, Carmen y Roxana, a mis tíos Raúl y Beltrán, a mis amigos de toda la vida, a mis compañeros de la universidad, a todos ellos que confiaron en mí y me apoyaron a lo largo de todo el proceso educativo. De manera muy especial dedico esta tesis a Milda, por brindarme el aliento y la inspiración para cumplir con este objetivo…

Agradecimientos

Quiero agradecer a mis asesores Dr. Lauro Enciso Rodas y Dr. José Luis Soncco Álvarez por su ayuda, recomendaciones y sugerencias durante el proceso de investigación.

También quiero agradecer al Dr. Harley Vera Olivera, Ing. Jared León Malpartida, Mgt. Gerar F. Quispe Torres e Ing. Franco M. Rosa Alagón por su colaboración.

Índice general

Índice de figuras

Índice de tablas

Acrónimos

Acronyms / Abbreviations

ABC Artificial Bee Colony

ALU Arithmetic Logic Unit

API Application Programming Interface

APSO Adaptive Particle Swarm Optimization

BIRCH Balanced Iterative Reducing and Clustring using Hierarchies

CLIQUE Clustering In Quest

CLSPSO Line Search-Particle Swarm Optimisation

DBSCAN Density-Based Spatial Clustering of Applications with Noise

DENCLUE Density-based Clustering

FOs First Order Statistics

FPA Flower Pollination Algorithm

GA Genetic Algorithms

GLCM Gray Level Co-occurrence Matrix

IoT Internet of Things

KNN K-nearest neighbor

LBP Local Binary Pattern

PRAM Local Memory Machine

MIMD Multiple Instruction Stream Single Data Stream

MISD Multiple Instruction Stream Single Data Stream

MMM Modular Memory Machine

MPI Message Passing Interface

NUMA Nonuniform Memory Access

UMA Uniform Memory Access

OPTICS Ordering, Points to Identifythe Clustering Structure

PRAM Parallel Random Access Machine

PSO Particle Swarm Optimization

P-SSO Parallel Social Spider Optimization

RAM Random Access Machine

RGA Real-coded genetic algorithm

SFTA Segmentation-based Fractal Texture Analysis

SIMD Single Instruction Stream MultipleData Stream

SMSSO Simplex Method-based Social Spider Optimization

SISD Single Instruction Stream Single Data Stream

SSO Social Spider Optimization

STING Statistical Information Grid

UCI University of California, Irvine

URL Uniform Resource Locator

Abstract

The adaptation of digital technologies and the application of the Internet in organizations, people and devices, generate an extraordinary amount of data in various areas of science such as: data mining, big data, pattern classification, image recognition, business intelligence, bioinformatics, outliers detection and IoT. Consequently, this data needs to be analyzed, processed and stored. The analysis process usually brings computational difficulties such as execution time and quality of results. Clustering is one of the most widely used classification techniques for analyzing large and small volumes of data. In the literature you can find algorithms such as: Social Spider Optimization (SSO), K-means, Artificial Bee Colony (ABC), Particle Swarm Optimization (PSO), Genetic Algorithms (GA). In this work the parallel version of the SSO algorithm is implemented, this implementation is called *Parallel Social Spider Optimization* (P-SSO). The research aims to improve the precision of the metric and the execution time of the SSO algorithm. For the development of this implementation, the island model mechanism with static topologies and dynamic topologies was used. In the experimental stage, the proposed algorithms were executed 50 times, for which 9 datasets from the UCI repository Machine Learning Repository was used. Statistical analysis was also performed to compare the SSO algorithm with the P-SSO algorithm. The results show that the parallel models of the P-SSO algorithm on average are 15 times faster than the SSO algorithm to classify large volumes of data, and 28 times faster for small volumes of data. It was also verified that the metric generated from the sum of the Euclidean distances for the P-SSO algorithm is very similar to the resulting metric of the SSO algorithm, and for some datasets this value is more optimal. Finally, it was found that the P-SSO algorithm models converge slower than their counterpart. This constitutes a significant contribution in improving the execution time of these algorithms to solve clustering problems, with very favorable metrics that verify the solution.

Keywords: *Bio-inspired algorithm, Social Spider Optimization, Parallel algorithms, Island models, Clustering*.

Resumen

La adaptación de las tecnologías digitales y la aplicación de Internet en las organizaciones, personas y dispositivos, generan una cantidad extraordinaria de datos en diversas áreas de la ciencia como por ejemplo: minería de datos, *big data*, clasificación de patrones, reconocimiento de imágenes, inteligencia de negocios, bioinformática, detección de *outliers* e *IoT*. En consecuencia estos datos requieren ser analizados, procesados y almacenados. El proceso de análisis generalmente traen dificultades computacionales como el tiempo de ejecución y la calidad de los resultados. *Clustering* es una de las técnicas de clasificación más utilizadas para analizar grandes y pequeños volúmenes de datos. En la literatura se puede hallar algoritmos como por ejemplo: *Social Spider Optimization* (SSO), K-means, *Artificial Bee Colony* (ABC), *Particle Swarm Optimization* (PSO), *Genetic Algorithms* (GA). En este trabajo se implementa la versión paralela del algoritmo SSO, esta implementación es denominada como *Parallel Social Spider Optimization* (P-SSO). El objetivo de esta investigación es mejorar la precisión de la métrica y el tiempo de ejecución del algoritmo SSO. Para el desarrollo de la implementación se utilizó el mecanismo de modelo de isla con topologías estáticas y topologías dinámicas. En la etapa experimental los algoritmos propuestos se ejecutaron 50 veces, para lo cual se usó 9 *dataset* del repositorio UCI *Machine Learning Repository*. También se realizó un análisis estadístico para comparar el algoritmo SSO con el algoritmo P-SSO. Los resultados muestran que los modelos paralelos del algoritmo P-SSO en promedio son 15 veces más rápido que el algoritmo SSO para clasificar grandes volúmenes de datos y 28 veces más rápido para pequeños volúmenes de datos. Así mismo se verificó que la métrica generada de la suma de las distancias Euclidianas para el algoritmo P-SSO es muy similar a la métrica resultante del algoritmo SSO y para algunos *dataset* este valor es más óptimo. Finalmente, se verificó que los modelos paralelos del algoritmo P-SSO convergen más lento que el algoritmo SSO. Esto constituye un aporte significativo en mejorar el tiempo de ejecución de estos algoritmos para resolver problemas de *clustering*, con métricas muy favorables que verifican la solución.

Palabras claves: *Algoritmos Bioinspirados, Social Spider Optimization, Algoritmos Paralelos, Modelos de Isla, Clustering*.

Introducción

Clustering es una de las técnicas más utilizadas para el análisis de datos en campos como por ejemplo: *data mining*, *big data*, clasificación de patrones, reconocimiento de imágenes, inteligencia de negocios, bioinformática, detección de *outliers* e *IoT*. El objetivo del *clustering* es agrupar aquellos datos que compartan un grado de similitud entre sí. En la literatura se puede encontrar algoritmos como por ejemplo: *Social Spider Optimization* (SSO), *K-means*, *Artificial Bee Colony* (ABC), *Particle Swarm Optimization* (PSO), *Genetic Algorithms* (GA). Estos algoritmos son aplicados para resolver problemas de *clustering* de grandes y pequeños volúmenes de datos. Las organizaciones, personas y dispositivos diariamente generan una enorme cantidad de datos. Estos datos requieren ser analizados, procesados y almacenados. Generalmente, analizar estos datos traen dificultades computacionales durante la ejecución, como el tiempo y la calidad de los resultados. Por esta razón es importante seguir investigando acerca de los algoritmos para resolver problemas de *clustering* encontrados en la literatura. En este trabajo se implementa la versión paralela del algoritmo *Social Spider Optimization* (SSO), esta implementación es llamada *Parallel Social Spider Optimization* (P-SSO). El objetivo de esta investigación es mejorar la precisión de la métrica y el tiempo de ejecución del algoritmo SSO, para solucionar problemas de *clustering*. Para el desarrollo e implementación se utiliza el mecanismo de modelos de isla encontradas en la literatura, tanto para topologías estáticas (anillo-unidireccional, árbol, red 6x4, toroide y grafo completo) y para topologías dinámicas (entre similares, entre buenos y malos, entre aleatorios). En la etapa experimental se utilizó *dataset* del repositorio UCI y se realizó una comparación entre los algoritmos SSO y P-SSO, para validar estos resultados se aplicaron métodos estadísticos. Los resultados muestran que los modelos paralelos del algoritmo P-SSO en promedio son 15 veces más rápido que el algoritmo SSO para clasificar grandes volúmenes de datos y 28 veces más rápido para pequeños volúmenes de datos, en este punto el modelo paralelo con topología estática anillo unidireccional es ligeramente más rápida que los modelos paralelos estudiados en este trabajo. En cuanto a la precisión de la métrica para evaluar un *cluster*, los modelos paralelos tienen resultados similares con respecto al algoritmo SSO y en algunos casos los modelos paralelos superar al algoritmo SSO. Sin embargo, se puede inferir que el modelo de

isla con topología estática red 6x4 ($\mathcal{P}sso_{na}$ y $\mathcal{P}sso_{nb}$) son mejores que el resto de los modelos paralelos.

El resto del trabajo está organizado de la siguiente manera: En el capítulo 1, se presenta el planteamiento del problema. En el capítulo 2 se detalla el marco teórico conceptual para abordar el tema de investigación. En el capítulo 3 se describe la metodología que se aplicó. En el capítulo 4 se presenta la propuesta del algoritmo paralelo *Social Spider Optimization*. En el capítulo 5 se detallan los experimentos, resultados, discusiones y conclusiones a los que se llegaron con la realización del proyecto.

Capítulo 1

Planteamiento del Problema

1.1 Situación problemática

Las organizaciones, personas y dispositivos diariamente generan una enorme cantidad de datos. Estos datos requieren ser analizados, procesados y almacenados. Generalmente, analizar estos datos para la toma de decisiones, traen dificultades computacionales durante la ejecución, como el tiempo y la calidad de los resultados (precisión de la métrica). *Clustering* es una de las técnicas más utilizadas para el análisis de datos en campos como por ejemplo: *data mining*, *big data*, clasificación de patrones, reconocimiento de imágenes, inteligencia de negocios, bioinformática, detección de *outliers* e *IoT*. El objetivo de *clustering* es agrupar aquellos datos que compartan un grado de similitud entre sí. En la literatura se puede encontrar algoritmos como por ejemplo: *Social Spider Optimization* (SSO), *K-means*, *Artificial Bee Colony* (ABC), *Particle Swarm Optimization* (PSO), *Genetic Algorithms* (GA). Estos algoritmos son aplicados para resolver problemas de *clustering* de grandes y pequeños volúmenes de datos. Otro problema frecuente al ejecutar estos algoritmos es que no se utiliza toda la capacidad de procesamiento de los equipos de cómputo, con esto nos referimos a los múltiples núcleos que posee el computador. Aplicando el mecanismo de modelo de isla con topologías estáticas y dinámicas que ofrece la computación paralela se podría mejorar significativamente el tiempo de ejecución y posiblemente la calidad de los resultados. Un caso de estudio real de estos problemas se encuentra en los sistemas en tiempo real, donde el tiempo de respuesta debe ser el menor posible. Por ejemplo para determinar el tráfico vehicular en una ciudad sobre la base de datos geo localizados, es preferible utilizar algoritmos de *clustering* paralelos para reducir considerablemente el tiempo de respuesta del sistema.

1.2 Formulación del problema

1.2.1 Problema general

Clustering posee múltiples aplicaciones en diferentes campos de la ciencia y la computación paralela permite aprovechar de mejor manera la capacidad de procesamiento de un computador. Por esta razón es importante seguir investigando acerca de los algoritmos encontrados en la literatura, en particular el algoritmo *Social Spider Optimization* (SSO) para resolver problemas de *clustering*, aplicando computación paralela. En consecuencia se plantea la siguiente interrogante: ¿En qué porcentaje la implementación del algoritmo SSO paralelo va a mejorar la precisión de la métrica y el tiempo de ejecución del algoritmo SSO secuencial, para resolver problemas de *clustering*?

1.2.2 Problemas específicos

- ¿Cuál es el valor de la métrica que genera el algoritmo SSO?
- ¿Cuál es el tiempo de ejecución del algoritmo SSO?
- ¿Cuáles son los mejores parámetros para ejecutar el algoritmo P-SSO?
- ¿Cuál es el valor de la métrica que genera el algoritmo P-SSO?
- ¿Cuál es el tiempo de ejecución del algoritmo P-SSO?
- ¿Cuál es la diferencia entre las métricas que generan los algoritmos SSO y P-SSO?
- ¿Cuál es el *speedup* entre los algoritmos SSO y P-SSO?

1.3 Justificación de la investigación

Esta investigación tiene justificación práctica, porque pretende mejorar la calidad de los resultados (precisión de la métrica) y el tiempo de ejecución del algoritmo *Social Spider Optimization* (SSO) para resolver problemas de *clustering*; con la finalidad de analizar y clasificar pequeños y grandes volúmenes de datos de manera más eficiente y rápida.

También tiene una justificación teórica, porque este trabajo sigue lo que se está haciendo en el estado del arte; que consiste en descubrir que mecanismos de topologías (estáticas o dinámicas) proporcionan mejores resultados en cuanto a la precisión de la métrica y el tiempo de ejecución del algoritmo SSO paralelo.

1.4 Objetivos de la investigación

1.4.1 Objetivo general

Determinar en que porcentaje el algoritmo SSO paralelo para resolver problemas de *clustering*, va a mejorar la precisión de la métrica y el tiempo de ejecución con respecto a la versión secuencial del algoritmo SSO.

1.4.2 Objetivos específicos

- Obtener la métrica que genera el algoritmo SSO.
- Medir el tiempo de ejecución del algoritmo SSO.
- Determinar los parámetros más adecuados para el algoritmo P-SSO.
- Obtener la métrica que genera el algoritmo P-SSO.
- Medir el tiempo de ejecución del algoritmo P-SSO.
- Comparar y validar los resultados de la métrica entre los algoritmos SSO y P-SSO, utilizando métodos estadísticos.
- Evaluar el *speedup* resultante entre los algoritmos SSO y P-SSO, utilizando métodos estadísticos.

1.5 Delimitaciones del trabajo

- La comparación se realizará únicamente entre los algoritmos SSO y P-SSO.
- Los experimentos se realizarán sobre *datasets*, que cumplan con una tarea predeterminada de clasificación y serán tomadas del repositorio UCI *Machine Learning Repository*.

1.6 Resultados esperados

Los resultados a priori que se pretende conseguir, consiste en encontrar una mejor métrica del algoritmo P-SSO con respecto al algoritmo SSO, en un 10% y también conseguir un *speedup* superior a $10x$ sobre el algoritmo SSO.

1.7 Contribuciones

Diseño e implementación del algoritmo P-SSO utilizando el mecanismo de modelos de isla con topologías estáticas y dinámicas encontradas en el estado del arte, con el objetivo de mejorar el valor de la métrica y el tiempo de ejecución del algoritmo SSO, para resolver problemas de *clustering* de manera más eficiente y rápida. El producto de este trabajo de investigación es un *paper* que tiene por título "*Parallel Social Spider Optimization Algorithms with Island Model for the Clustering Problem*", que fue presentado en *The International Conference on Information Management and Big Data* (SIMBig) 2020, obteniendo el reconocimiento por parte de la organización (ver anexo C), esta información también se encuentra detallado en el siguiente sitio web: https://simbig.org/SIMBig2020/. Posteriormente este *paper* fue publicando con la serie Springer CCIS (*Communications in Computer and Information Science*) y se puede encontrar en el siguiente sitio web: https://doi.org/10.1007/978-3-030-76228-5_9.

Capítulo 2

Marco Teórico Conceptual

2.1 *Clustering*

Según Han et al. (2011) *clustering* o simplemente agrupamiento es el proceso de dividir un conjunto de objetos de datos en subconjuntos. Cada subconjunto es un *cluster*, de modo que los objetos en un *cluster* son similares entre sí, pero distintos a los objetos en otros *clusters*. El conjunto de *clusters* resultante de un análisis puede denominarse agrupamiento. En este contexto, los diferentes métodos de agrupamiento pueden generar *cluters* diferentes sobre el mismo conjunto de datos. El análisis de *cluster* se utiliza ampliamente en muchas aplicaciones, como la inteligencia de negocios, el reconocimiento de patrones de imágenes, la búsqueda en la web, la biología y la seguridad. La agrupación también se denomina segmentación de datos en algunas aplicaciones porque la agrupación divide grandes conjuntos de datos en grupos según su similitud. La agrupación también puede utilizarse para la detección de valores atípicos (valores que están lejos de cualquier agrupación) pueden ser más interesantes que los casos comunes.

2.1.1 Clasificación de los métodos de agrupamiento

De acuerdo con Xu and Wunsch (2008) y Han et al. (2011) clasifican los métodos de agrupamiento en: método basado en particiones, método jerárquico, método basado en la densidad y el método *Grid-Based*.

Método basado en particiones

La versión más simple y fundamental del análisis de *cluster*, consiste en organizar los objetos de un conjunto en varios grupos o *clusters*. El número de *clusters* es el punto de partida

para este método. Los algoritmos más conocidos que utilizan este método son: *k-means* y *k-medoids*. A continuación se describe matemáticamente el agrupamiento basado en particiones (Jain et al., 1999; Xu and Wunsch, 2008).

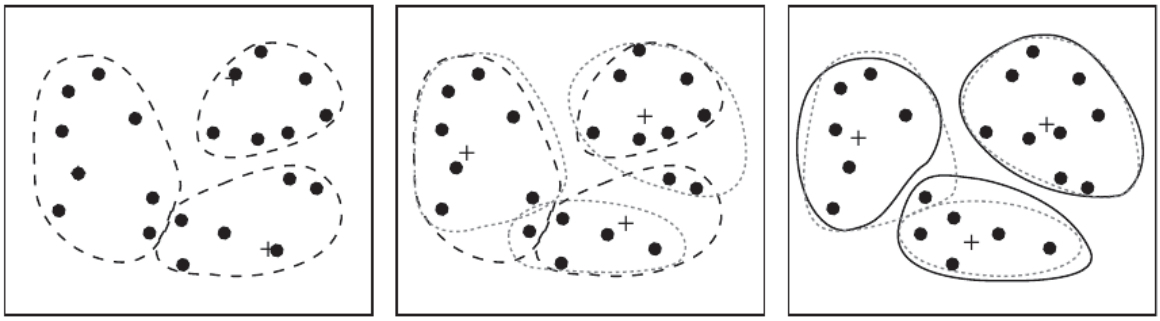
Dado un conjunto de patrones $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i, \ldots, \boldsymbol{x}_N\}$, donde $\boldsymbol{x_i} = (x_{i1}, x_{i2}, \ldots, x_{id}) \in \mathbb{R}^d$, cada medida x_{ij} determina una característica (atributo, dimensión o variable). Esta agrupación intenta buscar K-particiones de $\boldsymbol{X}$, $C = \{C_1, \ldots, C_k\}(K \leq N)$, tal que:

- $C_i \neq \emptyset, i = 1, \ldots, K$
- $\bigcup_{i=1}^{K} C_i = \boldsymbol{X}$
- $C_i \cap C_j = \emptyset; i, j = 1, \ldots, K$ y $i \neq j$

De forma general la métrica más popular para características continuas es el cálculo de la distancia Euclidiana.

$$D(\boldsymbol{X}_i, \boldsymbol{X}_j) = \sum_{l=1}^{d} |x_{il} - x_{jl}| \tag{2.1}$$

En la Figura 2.1, se muestra el proceso iterativo para generar *clusters* sobre un conjunto de datos en 2 dimensiones.



(a) Initial clustering **(b)** Iterate **(c)** Final clustering

Fig. 2.1 Método basado en particiones para generar *clusters*

Fuente: Han et al. (2011)

Método jerárquico

El método de agrupación jerárquico funciona agrupando el conjunto de objetos en una jerarquía o árbol de agrupaciones. Este método puede encontrar dificultades en la selección de puntos de fusión o división. Esa decisión es esencial, porque una vez que un grupo de

objetos se fusiona o se divide, el proceso siguiente opera sobre los nuevos *clusters* generados. Entonces no se puede deshacer lo que se ha hecho anteriormente, ni se podrá realizar el intercambio de objetos entre los *clusters*. A continuación se describe el agrupamiento basado en jerarquías (Xu and Wunsch, 2008).

Dado un conjunto de patrones $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i, \ldots, \boldsymbol{x}_N\}$, donde $\boldsymbol{x_i} = (x_{i1}, x_{i2}, \ldots, x_{id}) \in \mathbb{R}^d$, cada medida x_{ij} determina una característica (atributo, dimensión o variable), $H = \{H_1, \ldots, H_Q\}(Q \leq N)$, tal que:

- $C_i \in H_m, C_j \in H_l$ y $m > l$ implica
- $C_i \subset C_j$ o $C_i \cap C_j = \emptyset$ para todo $i, j \neq i, m, l = 1, \ldots, Q$.

Los resultados de la agrupación jerárquica suelen representarse mediante un árbol binario o *dendrogram* (Figura 2.2).

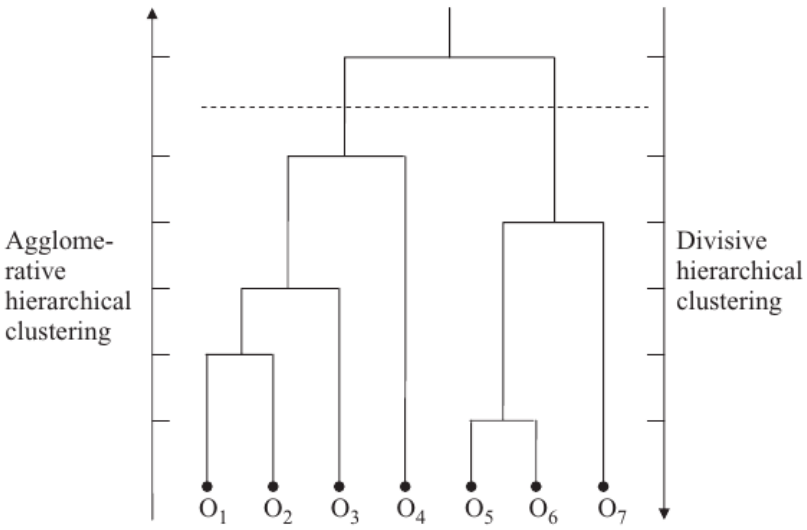


Fig. 2.2 *Dendrogram* a partir de una agrupación jerárquica

Fuente: Xu and Wunsch (2008)

El nodo raíz del *dendrogram* representa el conjunto de datos y cada nodo hoja se considera un punto de dato. Los nodos intermedios describen la proximidad de los objetos entre sí y la altura del *dendrogram* expresa la distancia entre cada par de puntos de datos o *clusters*, o entre un punto de datos y un *cluster*. Los resultados finales de la agrupación se obtienen cortando el *dendrogram* en diferentes niveles (la línea discontinua de la Figura 2.2). Los métodos más representativos son: *Balanced Iterative Reducing and Clustring using Hierarchies* (BIRCH) y Chameleon.

Método basado en la densidad

El método basado en particiones y el método jerárquico están diseñados para encontrar *clusters* dispuestos de forma esférica. Por esta razón tienen dificultades para descubrir agrupaciones de formas arbitrarias como la forma "S" y el óvalo (Ver Figura 2.3). Para encontrar *clusters* de forma arbitraria, alternativamente, se puede modelar los *clusters* como regiones densas en el espacio de datos, separadas por regiones dispersas. Esta es la principal estrategia detrás del método de agrupación basado en la densidad, que pueden descubrir *clusters* dispuestos de forma no esférica. Los métodos más representativos son: *Density-Based Spatial Clustering of Applications with Noise* (DBSCAN), *Ordering, Points to Identify the Clustering Structure* (OPTICS) y *Density-based Clustering* (DENCLUE).

Fig. 2.3 *Clusters* dispuestos de forma arbitraria

Fuente: Han et al. (2011)

Método *Grid-Based*

Los métodos de agrupación tratados hasta este punto son dirigidos por datos. El método de agrupación *Grin-based* es dirigido por el espacio al dividir el espacio de incrustación en celdas independientes a la distribución de los objetos de entrada. Este método utiliza una estructura de datos en cuadrícula de resolución múltiple. Cuantifica el espacio de objetos en un número finito de celdas que forman una estructura cuadriculada en la que se realizan todas las operaciones de agrupación. La principal ventaja de este enfoque es su rápido tiempo de procesamiento, que suele ser independiente del número de objetos de datos, pero depende del número de celdas de cada dimensión en el espacio cuantificado. Los métodos más representativos son: *Statistical Information Grid* (STING), *Clustering In Quest* (CLIQUE).

2.1.2 Medida de similitud/disimilitud

Los conceptos de *clustering* y métrica (medida de similitud), aplicados en este trabajo fue definido por Maulik and Bandyopadhyay (2000). Estas definiciones se muestran a continuación.

Sea $S = \{x_1, x_2, \ldots, x_n\}$ y $C = \{c_1, c_2, \ldots, c_k\}$ conjuntos de puntos N-dimensionales. El problema de *clustering* en un espacio N-dimensional $\mathbb{R}^N$ consiste en encontrar una partición del conjunto S en k *clusters* basados en una medida de similitud, donde cada *cluster* tiene como centro un elemento c_i de C.

Supongamos que $G_i, i = 1, \ldots, k$, representa k *clusters*, entonces se verifican las siguientes propiedades:

- $G_i \neq \emptyset$, para $i = 1, \ldots, k$
- $\bigcup_{i=1}^{k} G_i = S$
- $C_i \cap C_j = \emptyset$, para $i, j = 1, \ldots, k$ y $i \neq j$;

La métrica utilizada para evaluar una partición (*cluster*) se basa en la suma de las distancias Euclidianas. La definición de esta medida de similitud $\mathscr{M}$ para k *clusters* $G_1, G_2 \ldots, G_k$ está dado por:

$$\mathscr{M}(G_1, G_2 \ldots, G_k) = \sum_{i=1}^{k} \sum_{x_j \in G_i} ||x_j - c_i|| \tag{2.2}$$

Los algoritmos (*Social Spider Optimization* y *Parallel Social Spider Optimization*) presentados en este trabajo tratan de encontrar el conjunto de centros $\{c_1, c_2, \ldots, c_k\}$ y también un valor mínimo para la métrica $\mathscr{M}$.

2.2 Algoritmo *Social Spider Optimization* (SSO)

Este algoritmo bioinspirado fue propuesto por Cuevas et al. (2013) y adaptado por Vera-Olivera et al. (2016) para solucionar el problema de *clustering*. A continuación se muestra el algoritmo secuencial SSO (Algoritmo 1) propuesta por Vera-Olivera et al. (2016).

A continuación se explica el Algoritmo 1, en la línea 1 se lee el *dataset* el cual se analizará, el segmento de código a partir de la línea 2 a la línea 4 inicializan las estructuras de datos utilizados (*clusterCenter*, *spider* y *population*), a partir de la línea 6 hasta la línea 11 se vuelve a producir una nueva generación, de acuerdo con la adaptación del algoritmo SSO propuesto por Vera-Olivera et al. (2016).

Algoritmo 1: SSO secuencial para *clustering*

Entrada: Un *dataset* D de m puntos n-dimensionales $D = \{d_1, d_2, \ldots, d_m\}$; un entero $k > 0$ que representa el número de *clusters*; *numeroGeneraciones*

Salida: Métrica $\mathscr{M}$ de los *clusters* encontrados

```
Leer dataset D;
Generar población inicial P;
Calcular fitness de la población P;
Calcular peso de la población P;
para i ← 2 to numeroGeneraciones hacer
    Operador cooperativo para arañas hembra;
    Operador cooperativo para arañas macho;
    Operador de apareamiento;
    Substitución de arañas en P;
    Calcular fitness de la población P;
    Calcular peso de la población P;
// Retornar el fitness (métrica M) de la mejor solución
devolver M;
```

Una de las operaciones más importantes en el Algoritmo 1 es el cálculo de la función *fitness*. Según el enfoque propuesto por Cuevas et al. (2013), cada individuo (araña) recibe un peso w_i que representa la calidad de la solución. Este peso se calcula con la siguiente ecuación:

$$w_i = \frac{J(s_i) - peor_s}{mejor_s - peor_s} \tag{2.3}$$

Donde $J(s_i)$ es el valor de aptitud obtenido por la evaluación de la posición de una araña s_i con respecto a la función $J(.)$. Los valores de $mejor_i$ y $peor_i$ son considerados un problema de maximización y se definen de la siguiente manera:

$$mejor_s = \max_{k \in 1,2,\ldots,N} (J(s_k)) \tag{2.4}$$

$$peor_s = \min_{k \in 1,2,\ldots,N} (J(s_k)) \tag{2.5}$$

En la Figura 2.4 se aprecia las operaciones necesarias para el mecanismo de comunicación entre los individuos hembra y macho. También se muestra el flujo de datos para generar nuevos individuos.

Finalmente, el análisis de complejidad de este algoritmo para solucionar problemas de *clustering* es $T(n) = O(km^3n)$. Donde, k representa el número de *clusters*, m representa al

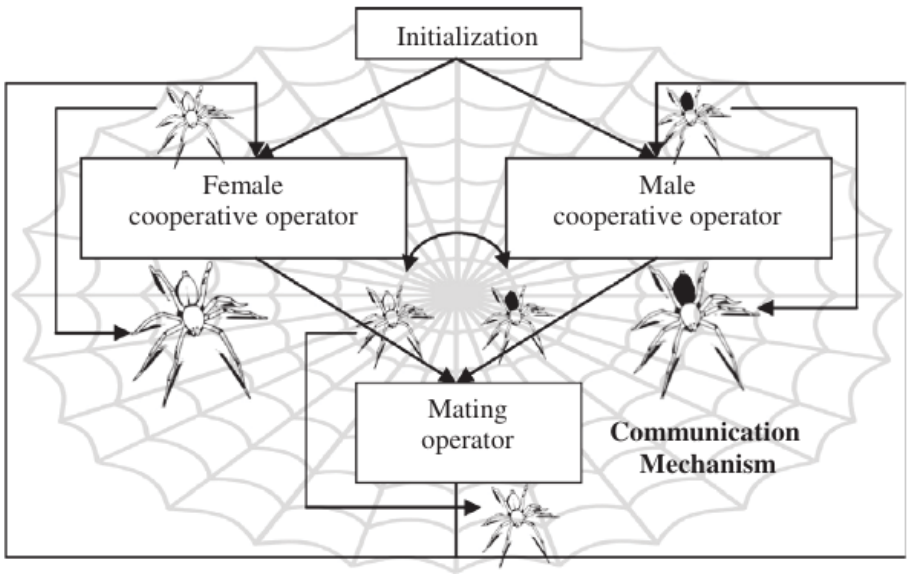


Fig. 2.4 Flujo de datos para el algoritmo SSO

Fuente: Cuevas et al. (2013)

número de individuos de una población y n representa al número de generaciones (Vera-Olivera et al., 2016).

2.3 Computación paralela

De acuerdo con Pacheco (2011), la computación paralela, es aquel programa donde múltiples tareas cooperan estrechamente para resolver un problema; existen dos enfoques ampliamente utilizados: paralelismo de tareas y paralelismo de datos. En el paralelismo de tareas, las tareas se dividen entre los núcleos para resolver el problema. En el paralelismo de datos, los datos utilizados se dividen entre los núcleos y cada núcleo lleva a cabo operaciones similares en la parte de los datos que le corresponde.

La mayoría de las CPU modernas son procesadores multinúcleo y, por tanto, constan de varias unidades de procesamiento independientes denominadas núcleos. Además, estas CPU admiten *Simultaneous Multithreading* (SMT), de modo que cada núcleo puede ejecutar (casi) simultáneamente múltiples flujos independientes de instrucciones llamados hilos. Para un programador, cada núcleo de cada procesador actúa como varios núcleos lógicos, cada uno de los cuales puede ejecutar su propio programa o un hilo dentro de un programa de forma independiente (Trobec et al., 2018).

2.4 Arquitectura de computación paralela

2.4.1 Taxonomia de Flynn

La taxonomía de procesadores más conocida fue propuesta por Flynn (1966) y está clasificado en: *Single Instruction Stream Single Data Stream* (SISD), *Single Instruction Stream Multiple Data Stream* (SIMD), *Multiple Instruction Stream Single Data Stream* (MISD) y *Multiple Instruction Stream Single Data Stream* (MIMD). En este contexto *Stream* es usado para referirse a una secuencia de datos o instrucciones.

a) SISD: Está compuesto por los computadores que poseen un único procesador y ejecutan una instrucción sobre un solo dato en una unidad de tiempo, causando con esto un cuello de botella.

b) SIMD: Consta de *n* procesadores idénticos, cada uno ejecuta instrucciones similares con su propio acceso a la memoria y a sus datos.

c) MISD: Varias unidades de procesamiento operan sobre un único conjunto de datos, donde cada unidad actúa sobre los datos de forma independiente.

d) MIMD: Consta de *n* procesadores, cada uno ejecuta sus propias instrucciones, con su propio acceso a la memoria y a sus datos.

Los sistemas SIMD son ideales para paralelizar bucles sencillos que operan sobre grandes arreglos de datos. El paralelismo que se obtiene dividiendo datos entre los procesadores y haciendo que todos los procesadores apliquen (más o menos) las mismas instrucciones a sus subconjuntos de datos se llama paralelismo de datos. El paralelismo SIMD puede ser muy eficiente en problemas paralelos de datos de gran dimensión, pero los sistemas SIMD a menudo no se desempeñan bien con otros tipos de problemas paralelos. Por otra parte, los sistemas MIMD soportan múltiples *streams* de instrucciones simultáneas que operan en múltiples *streams* de datos. Por consiguiente, los sistemas MIMD suelen estar compuestos de un conjunto de unidades o núcleos de procesamiento totalmente independientes. Cada núcleo tiene su propia unidad de control y su propia ALU. Además, a diferencia de los sistemas SIMD, los sistemas MIMD suelen ser asíncronos (Pacheco, 2011).

2.4.2 Memoria compartida y memoria distribuida

Sistema de memoria compartida

En un sistema de memoria compartida, un conjunto de procesadores se conectan de forma autónoma a un sistema de memoria a través de una red de interconexión y cada procesador

puede acceder a cada ubicación de la memoria (Figura 2.5). Los procesadores suelen comunicarse implícitamente accediendo a estructuras de datos compartidas.

En los sistemas de memoria compartida con múltiples procesadores multinúcleo, la interconexión puede conectar todos los procesadores directamente a la memoria principal o cada procesador puede tener una conexión directa a un bloque de memoria principal y los procesadores pueden acceder a los bloques de memoria principal de los demás a través de un hardware especial incorporado en los procesadores. En el primer tipo de sistema (Figura 2.6a), el tiempo de acceso a todas las ubicaciones de memoria será el mismo para todos los núcleos, a este tipo de sistema se le denomina sistema *Uniform Memory Access* (UMA).

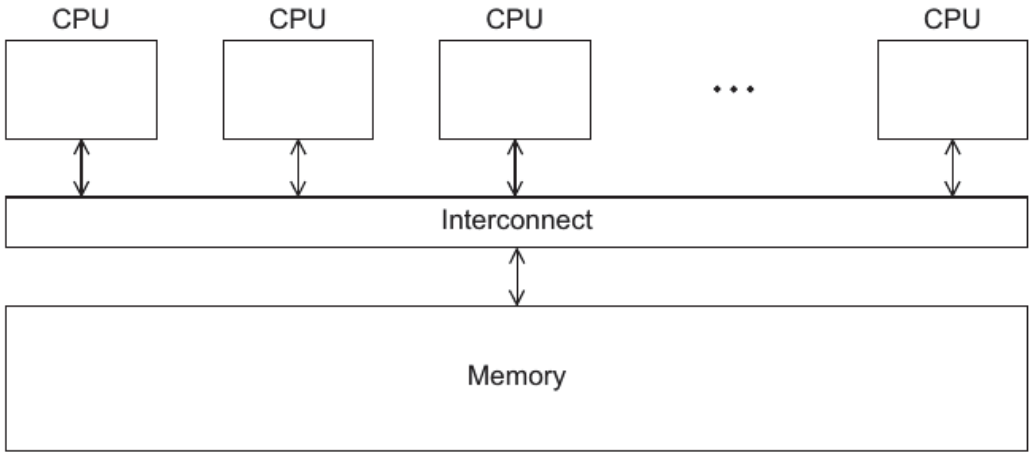
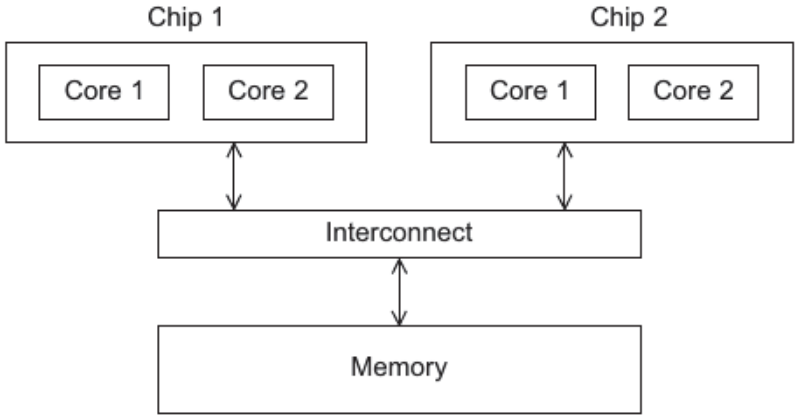


Fig. 2.5 Sistema de memoria compartida

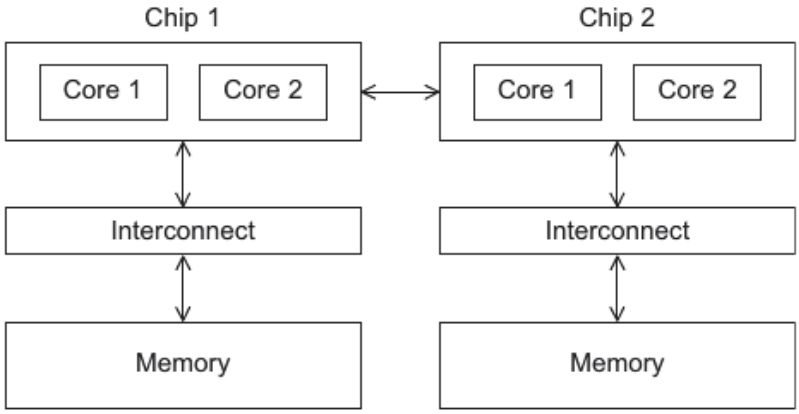
Fuente: Pacheco (2011)

Mientras que en el segundo tipo (Figura 2.6b) se puede acceder más rápidamente a una ubicación de memoria a la que un núcleo está directamente conectado que a una ubicación de memoria a la que se debe acceder a través de otro chip, a este de tipo se le denomina sistema *Nonuniform Memory Access* (NUMA).

Los sistemas UMA suelen ser más fáciles de programar, ya que el programador no tiene que preocuparse por los diferentes tiempos de acceso a las distintas posiciones de memoria. Esta ventaja puede verse compensada por el acceso más rápido a la memoria directamente conectada en los sistemas NUMA. Además, los sistemas NUMA tienen el potencial de utilizar mayores cantidades de memoria que los sistemas UMA que los sistemas UMA (Pacheco, 2011).



(a) UMA



(b) NUMA

Fig. 2.6 Sistemas multinúcleo

Fuente: Pacheco (2011)

Sistema de memoria distribuida

En un sistema de memoria distribuida, cada procesador está asociado con su propia memoria privada y los pares procesador-memoria se comunican a través de una red de interconexión (ver Figura 2.7). Los procesadores suelen comunicarse explícitamente enviando mensajes o utilizando funciones especiales que dan acceso a la memoria de otro procesador. En los programas de memoria distribuida, los núcleos pueden acceder directamente solo a sus propias memorias privadas. Para realizar estas tareas existen una variedad de APIs. Sin embargo, la más utilizada con diferencia es la de paso de mensajes *Message Passing Interface* (MPI). Una API de paso de mensajes proporciona (como mínimo) una función de envío y recepción. Los procesos se identifican típicamente por *ranks* en el rango $0, 1, \ldots, p-1$, donde p es el número de procesos.

Los sistemas de memoria distribuida más extendidos son los llamados *clusters*. Están compuestos por un conjunto de sistemas básicos (Por ejemplo un computador conectado a una red). Los nodos de estos sistemas, las unidades de cálculo individuales unidas por la red de comunicación, suelen ser sistemas de memoria compartida con uno o varios procesadores multinúcleo. Para diferenciar estos sistemas de los de memoria distribuida pura, a veces se denominan sistemas híbridos. La red proporciona la infraestructura necesaria para convertir grandes redes de ordenadores distribuidos geográficamente en un sistema unificado de memoria distribuida. En general, un sistema de este tipo será heterogéneo, es decir, los nodos individuales pueden estar construidos con diferentes tipos de hardware (Pacheco, 2011).

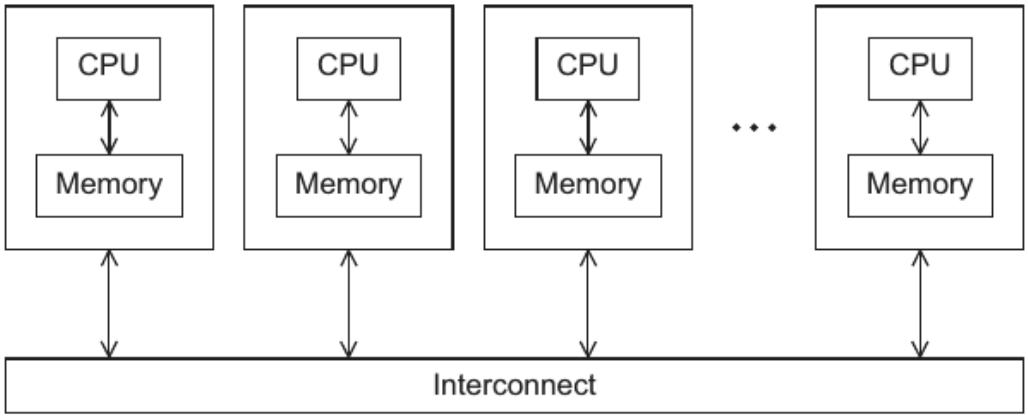


Fig. 2.7 Sistema de memoria distribuida

Fuente: Pacheco (2011)

2.4.3 Comunicación entre procesadores paralelos

Los procesadores paralelos requieren intercambiar datos entre ellos para completar las tareas que se les asignan, para lo cual existen cinco tipos de comunicación (Ver Figura 2.8): *unicast*, *multicast*, *broadcast*, *gather* y *reduce* Gebali (2011).

a) *Unicast*: Esta operación involucra un par de procesadores: al emisor y al receptor, también denominado como comunicación punto a punto. Este tipo de comunicación a menudo en máquinas (SIMD).

b) *Multicast*: Esta operación implica un procesador emisor y varios procesadores receptores.

c) *Broadcast*: Esta operación implica el envío de los mismos datos a todos los procesadores en el sistema.

d) *Gather*: Esta operación implica recopilar datos de varios o todos los procesadores del sistema.

e) *Reduce*: Esta operación es similar a la operación de *gather* excepto que se realiza alguna operación en los datos recopilados. Un ejemplo de la operación de reducción es cuando todos los datos producidos por todos los procesadores deben agregarse para producir un valor final.

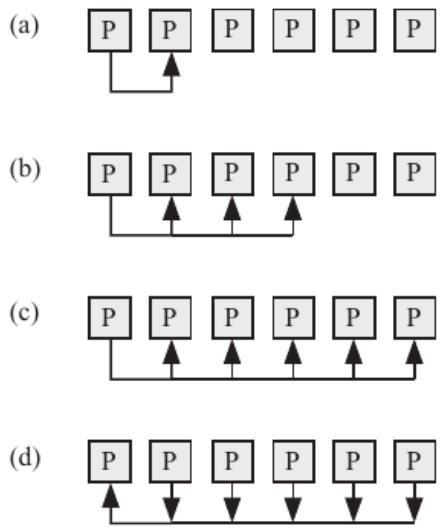


Fig. 2.8 Comunicación entre procesadores: (a) *Unicast*, (b) *Multicast*, (c) *Broadcast*, (d) *Gather*, (e) *Reduce*

Fuente: Gebali (2011)

2.4.4 Redes de interconexión

Según Rauber and Rünger (2013) y Trobec et al. (2018), las redes de interconexión son un mecanismo para transferir datos entre nodos de un procesador o entre procesadores y el módulo de memoria. Las redes de interconexión típicas se crean mediante enlaces y conmutadores. Un enlace corresponde a medios físicos como un conjunto de cables o fibras capaces de transportar información.

Una red de interconexión puede modelarse como un grafo $\mathfrak{G}(N,C)$, donde N es un conjunto de nodos de comunicación y C es un conjunto de enlaces de comunicación (o canales) entre los nodos de comunicación. Experimentos recientes demostraron que los tiempos de ejecución de la mayoría de las aplicaciones paralelas dependen cada vez más del

tiempo de comunicación en lugar del tiempo de cálculo. Entonces, a medida que aumenta el número de unidades de procesamiento, el rendimiento de las redes de interconexión se está volviendo más importante que el rendimiento de la unidad de procesamiento.

La red de interconexión tiene un gran impacto en la eficiencia y la escalabilidad de la computación paralela. En otras palabras, el alto rendimiento de una red de interconexión puede reflejarse en mayores velocidades. Tres de los más importantes factores en el rendimiento de una red de interconexión son: el enrutamiento, los algoritmos de control de flujo y la topología de red. El enrutamiento es el proceso de selección de un camino para el tráfico en una red de interconexión; el control de flujo es el proceso de gestión de la velocidad de transmisión de datos entre dos nodos para evitar que un emisor rápido abrume a un receptor lento; y topología de la red es la disposición de los diversos elementos, como nodos y canales de comunicación.

Clasificación de las redes de interconexión

Las redes de interconexión se pueden clasificar como dinámicas y estáticas (Figura 2.9). Las redes estáticas (directas) consisten en enlaces de comunicación punto a punto entre nodos del procesador. Las redes dinámicas (indirectas) se construyen utilizando conmutadores y enlaces de comunicación. Los enlaces de comunicación están conectados entre sí dinámicamente por los conmutadores para establecer rutas entre los nodos del procesador y los módulos de memoria (Grama et al., 2003).

Se dice que una red es directa cuando cada nodo está conectado directamente con sus nodos vecinos. Una red indirecta conecta los nodos a través de conmutadores. Por lo general, conecta las unidades de procesamiento de un extremo de la red con los módulos de memoria en el otro extremo de la red. Su ventaja es que puede establecer una conexión entre las unidades de procesamiento y los módulos de memoria de forma arbitraria (Trobec et al., 2018).

Topologías de redes de interconexión

La topología de una red de interconexión se suele representar con un grafo, los nodos representan los conmutadores o procesadores y las aristas representan los enlaces de comunicación entre los conmutadores o procesadores. Existen muchas topologías de red capaces de interconectar unidades de procesamiento y módulos de memoria. Sin embargo, no todas las topologías de red son capaces de transmitir solicitudes de memoria con la suficiente rapidez para respaldar eficazmente el cálculo paralelo. Además, resulta que la topología de red tiene una gran influencia en el rendimiento de la red de interconexión y en consecuencia,

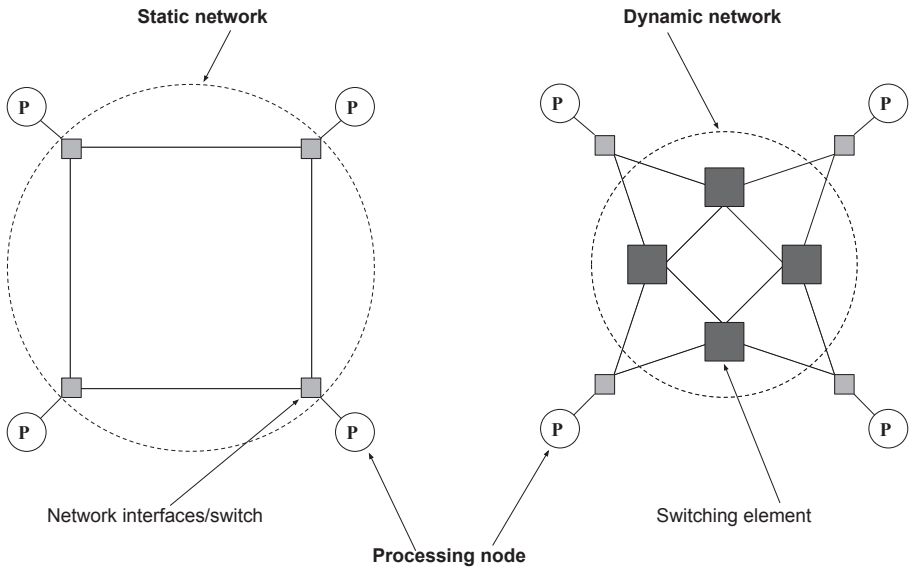


Fig. 2.9 Redes de interconexión Estática y Dinámica

Adaptado de: Grama et al. (2003)

de la computación paralela. Finalmente, la topología de la red puede plantear dificultades considerables en la construcción real de la red y su costo (Trobec et al., 2018).

Existen topologías de redes de interconexión muy conocidas que los investigadores han propuesto, analizado, construido, probado y utilizado. A continuación se enumera las topologías más notables o populares: *bus*, red (malla), red 3D, toroide, hipercubo, árbol, árbol binario, grafo completo, anillo, multietapa, estrella y topologías aleatorias. (Grama et al., 2003; Pacheco, 2011; Gebali, 2011; Rauber and Rünger, 2013); Trobec et al., 2018).

2.5 Modelos de computación paralela

Teniendo en cuenta a Trobec et al. (2018), los computadores paralelos varían mucho en su organización. Por ejemplo, sus unidades de procesamiento pueden o no estar conectadas directamente entre sí; algunas unidades de procesamiento pueden compartir una memoria, mientras que otras pueden poseer únicamente memorias privadas (locales); el funcionamiento de las unidades de procesamiento pueden estar sincronizados por un reloj en común, o pueden funcionar cada una a su propio ritmo. Además, suele haber detalles de arquitectura y especificaciones de los componentes de hardware, todo aparece durante el diseño y uso real de un ordenador. Finalmente, hay diferencias tecnológicas, que se manifiestan a diferentes velocidades de reloj, tiempos de acceso a la memoria, etc.

Un modelo computacional puede utilizarse para evaluar algoritmos independientemente de una implementación en un lenguaje de programación específico y del uso de un sistema informático en concreto. Para ser útil, un modelo computacional debe abstraerse de muchos detalles de un sistema informático específico, mientras que, por otro lado, debe capturar aquellas características de una amplia clase de sistemas informáticos que tienen una mayor influencia en el tiempo de ejecución de los algoritmos (Rauber and Rünger, 2013).

El modelo *Random Access Machine* (RAM) extrae las propiedades importantes de los ordenadores de propósito general secuenciales y que de hecho se han tomado como base conceptual para el modelado de la computación paralela y los computadores paralelos. En la Figura 2.10 se muestra el modelo de computación RAM, donde se visualiza una memoria **M** (que contiene instrucciones de programa y datos) y una unidad de procesamiento **P** (que ejecuta instrucciones sobre los datos).

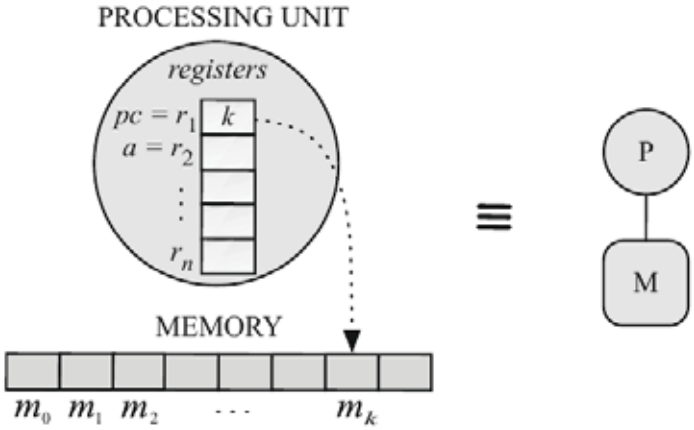


Fig. 2.10 Modelo de computación RAM

Fuente: Trobec et al. (2018)

En la RAM, la memoria es una secuencia potencialmente infinita de ubicaciones de igual tamaño $m_0, m_1, \ldots$ El índice i se denomina dirección de m_i. La unidad de procesamiento puede acceder directamente a cada ubicación: dada una i arbitraria, la lectura de m_i o la escritura en m_i se realiza en tiempo constante. Los registros son una secuencia $r_1, \ldots, r_n$ de ubicaciones en la unidad de procesamiento. Los registros son directamente accesibles y dos de ellos tienen funciones especiales. El contador de programa $pc(= r_1)$ contiene la dirección de la posición de la memoria que contiene la siguiente instrucción a ejecutar. El acumulador a $(= r_2)$ interviene en la ejecución de cada instrucción. Al resto de los registros se les asignan funciones según sea necesario. El programa es una secuencia finita de instrucciones (similar que en los ordenadores reales) (Trobec et al. ,2018).

Un modelo de multiprocesador es un modelo de computación paralela que se basa en el modelo de computación RAM, lo que da lugar a tres modelos de multiprocesador diferentes. Cada uno de los tres modelos tiene algún número $p(\geq 2)$ de unidades de procesamiento, pero los modelos difieren en la organización de sus memorias y en la forma en que las unidades de procesamiento acceden a las memorias. Los modelos se denominan: *Parallel Random Access Machine*, *Local Memory Machine* y *Modular Memory Machine*.

2.5.1 *Parallel Random Access Machine* (PRAM)

El modelo PRAM, tiene p unidades de procesamiento conectadas a una memoria compartida no limitada (Figura 2.11). Cada unidad de procesamiento puede acceder en un paso a cualquier ubicación (palabra) de la memoria compartida emitiendo una petición de memoria directamente a la memoria compartida. En este modelo no existe una red de interconexión para transferir las peticiones de memoria y los datos de ida y vuelta entre las unidades de procesamiento y la memoria compartida. Sin embargo, la suposición de que cualquier unidad de procesamiento puede acceder a cualquier ubicación de memoria en un solo paso no es realista. Suponga que las unidades de procesamiento P_i y P_j emiten simultáneamente instrucciones I_i e I_j en las que ambas instrucciones pretenden acceder (para leer o escribir) en la misma posición de memoria L (Figura 2.12). Incluso si un acceso físico simultáneo a L hubiera sido posible, dicho acceso podría haber dado lugar a contenidos impredecibles para L. Por lo tanto, es razonable suponer que, eventualmente, los accesos reales de I_i e I_j a L son de alguna manera ejecutados secuencialmente por el hardware para que I_i e I_j accedan físicamente a L uno tras otro.

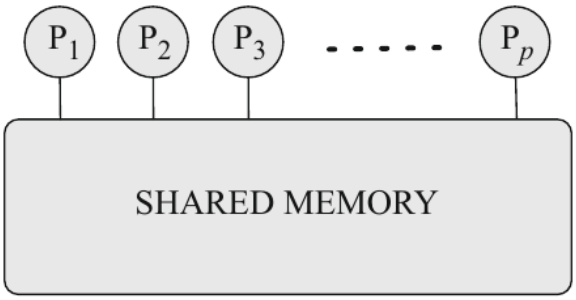


Fig. 2.11 Modelo de computación paralela PRAM

Fuente: Trobec et al. (2018)

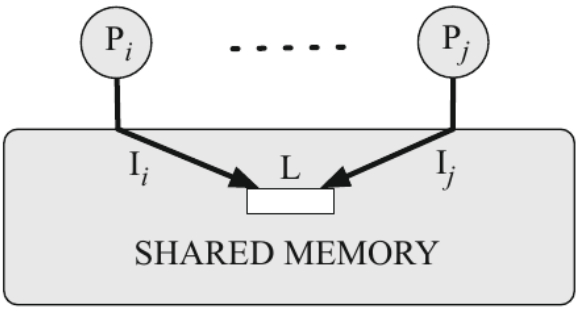


Fig. 2.12 Acceso simultáneo a una ubicación en el modelo PRAM

Fuente: Trobec et al. (2018)

2.5.2 *Local Memory Machine* (LMM)

El modelo LMM consta de p unidades de procesamiento, cada una con su propia memoria local (Figura 2.13). Las unidades de procesamiento están conectadas a una red de interconexión en común. Cada unidad de procesamiento puede acceder directamente a su propia memoria local. En cambio, para acceder a una memoria no local (es decir, a la memoria local de otra unidad de procesamiento) se realiza enviando una solicitud de memoria a través de la red de interconexión. Se supone que todas las operaciones locales, incluido el acceso a la memoria local, requieren un tiempo unitario. En consecuencia, el tiempo necesario para acceder a una memoria no local depende de la capacidad de la red de interconexión y del patrón de accesos a la memoria no local desde las otras unidades de procesamiento, ya que los accesos pueden congestionar la red de interconexión.

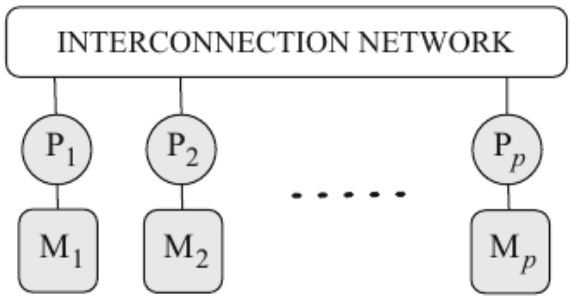


Fig. 2.13 Modelo de computación paralela LMM

Fuente: Trobec et al. (2018)

2.5.3 *Modular Memory Machine* (MMM)

El modelo MMM (Figura 2.14) consta de p unidades de procesamiento y m módulos de memoria a los que puede acceder cualquier unidad de procesamiento mediante una red de interconexión. No hay memorias locales para las unidades de procesamiento. Una unidad de procesamiento puede acceder al módulo de memoria enviando una solicitud de memoria a través de la red de interconexión. Se supone que las unidades de procesamiento y los módulos de memoria están dispuestos de tal manera que cuando no hay accesos el tiempo para que cualquier unidad de procesamiento acceda a cualquier módulo de memoria es aproximadamente uniforme. Sin embargo, cuando hay accesos a memoria, el tiempo de acceso depende de la capacidad de la red de interconexión y del patrón de accesos a la memoria.

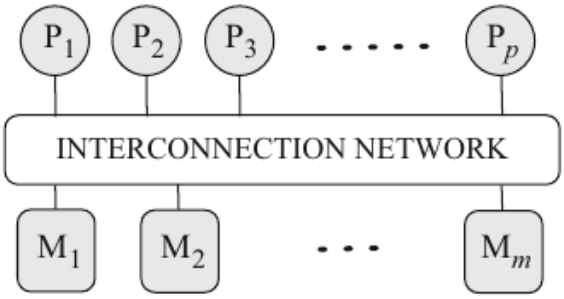


Fig. 2.14 Modelo de computación paralela MMM

Fuente: Trobec et al. (2018)

2.6 Métricas de rendimiento para programas paralelos

Un criterio importante para la utilidad de un programa paralelo es su tiempo de ejecución. El tiempo de ejecución paralelo $T_{par}(n)$ de un programa es el tiempo que transcurre entre el inicio del programa y el final de la ejecución en todos los procesadores participantes. El tiempo de ejecución en paralelo suele expresarse para un número específico p de procesadores en función del tamaño del problema n. Donde, el tamaño del problema viene dado por el tamaño de los datos de entrada. Los conceptos presentados en esta sección son desarrollados tomando en cuenta a Pacheco (2011), Rauber and Rünger (2013) y Trobec et al. (2018).

2.6.1 *Speedup* y eficiencia

Al ejecutar un programa P con p núcleos y un hilo o proceso en cada núcleo, entonces el programa paralelo se ejecutará p veces más rápido que el programa secuencial. Cuando esto sucede, se dice que el programa paralelo tiene un *speedup* lineal.

$$T_{par}(P) = \frac{T_{sec}(P)}{p} \tag{2.6}$$

Los programas de memoria distribuida casi siempre necesitan transmitir datos a través de la red, esto suele ser mucho más lento que el acceso a la memoria local. Por otro lado, los programas secuenciales no tendrán estos gastos generales. Por esta razón, será muy inusual encontrar que los programas paralelos tengan un *speedup* lineal. Además, es probable que las sobrecargas aumenten a medida que se aumenta el número de procesos o hilos. Si se define el *speedup* de un programa paralelo como:

$$S(P) = \frac{T_{sec}(P)}{T_{par}(P)} \tag{2.7}$$

Entonces, el *speedup* lineal es $S = p$, lo cual es inusual. También, a medida que p aumenta, se espera que S sea una fracción cada vez más pequeña. Otra forma de decir esto es que S/p probablemente se reducirá cada vez más a medida que p aumente. A este valor S/p, se le llama eficiencia de un programa paralelo.

$$E(P) = \frac{S}{p} = \frac{\frac{T_{sec}(P)}{T_{par}(P)}}{p} = \frac{T_{sec}(P)}{p.T_{par}(P)} \tag{2.8}$$

2.6.2 Teorema de Brent

El teorema de Brent permite cuantificar el rendimiento de un programa paralelo cuando se reduce el número de unidades de procesamiento. Sea M una PRAM arbitraria y suponemos que el número de unidades de procesamiento es siempre suficiente para cubrir todas las necesidades de cualquier programa paralelo. Cuando un programa paralelo P se ejecuta en M, diferentes números de operaciones de P se ejecuta, en cada paso, por diferentes unidades de procesamiento de M. Supongamos que un total de W operaciones se realizan durante la ejecución paralela de P en M (W también se llama el trabajo de P) y denotemos el tiempo de ejecución paralelo de P en M por $T_{par,M}(P)$. Ahora reduzcamos el número de unidades de procesamiento de M a un número fijo p y denotamos la máquina obtenida con el número reducido de unidades de procesamiento por R. R es una PRAM del mismo tipo que M la cual puede utilizar, en cada paso de su operación, a lo sumo p unidades de procesamiento.

Ejecutemos ahora P en R. Si p unidades de procesamiento no pueden soportar, en cada paso de la ejecución, todo el paralelismo potencial de P, entonces el tiempo de ejecución paralelo de P en R ($T_{par,R}(P)$), puede ser mayor que $T_{par,M}(P)$. Ahora se plantea la pregunta: ¿Se puede cuantificar $T_{par,R}(P)$? La respuesta viene dada por el Teorema de Brent que establece lo siguiente.

$$T_{par,R}(P) = O\left(\frac{W}{p} + T_{par,M}(P)\right) \tag{2.9}$$

2.6.3 Ley de Amdahl

Intuitivamente, se espera que al duplicar el número de unidades de procesamiento, el tiempo de ejecución en paralelo se reduce a la mitad. En otras palabras, se espera que el aumento de velocidad de la paralelización sea una función lineal del número de unidades de procesamiento. Sin embargo, el aumento de velocidad lineal de la paralelización es solo un óptimo deseable. La mayoría de los programas paralelos tienen un aumento de velocidad casi lineal para un número pequeño de elementos a procesar y para procesar un gran número de elementos la velocidad converge a un valor constante.

En general, un programa P ejecutado por un computador paralelo puede dividirse en dos partes, la parte P_1, que no se beneficia de las unidades de procesamiento múltiples y la parte P_2, que sí se beneficia de las unidades de procesamiento múltiples. Además, del beneficio de P_2, los tiempos de ejecución secuencial de P_1 y P_2 influyen en el tiempo de ejecución paralela de todo P y en consecuencia, en el aumento de velocidad de P. A continuación, se evalúa cuantitativamente cómo el aumento de la velocidad de P depende de los tiempos de ejecución secuencial de P_1 y P_2 y de su capacidad de paralelización y explotación de múltiples unidades de procesamiento.

Sea $T_{seq}(P)$ el tiempo de ejecución secuencial de P. Como $P = P_1P_2$, una secuencia de partes P_1 y P_2, se tiene

$$T_{sec}(P) = T_{sec}(P_1) + T_{sec}(P_2) \tag{2.10}$$

donde $T_{sec}(P_1)$ y $T_{sec}(P_2)$ son los tiempos de ejecución secuencial de P_1 y P_2, respectivamente (Ver Figura 2.15).

Cuando realmente se utilizan las unidades de procesamiento adicionales en la ejecución paralela de P, es la ejecución de P_2 la que se acelera en algún factor $s > 1$, mientras que la ejecución de P_1 no se beneficia de las unidades de procesamiento adicionales. Es decir, el tiempo de ejecución de P_2 se reduce de $T_{seq}(P_2)$ a $\frac{1}{s}T_{seq}(P_2)$, mientras que el tiempo de ejecución de P_1 sigue siendo el mismo, $T_{seq}(P_1)$. Así, tras la utilización de las unidades de

procesamiento adicionales el tiempo de ejecución en paralelo $T_{par}(P)$ de todo el programa P es

$$T_{par}(P) = T_{sec}(P_1) + \frac{1}{s} T_{sec}(P_2) \tag{2.11}$$

El *speedup* $S(P)$ del programa completo P, se calcula a partir de la definición descrita en la Ecuación 2.7. Adicionalmente, $S(P)$ se puede expresar en términos de b, la fracción de $T_{seq}(P)$ durante la cual la paralelización de P es beneficiosa. Del cual se obtiene lo siguiente

$$b = \frac{T_{sec}(P_2)}{T_{sec}(P)} \tag{2.12}$$

Sustituyendo la Ecuación 2.12 en la Ecuación 2.7, se obtiene la Ley de Amdahl.

$$S(P) = \frac{1}{1 - b + \frac{b}{s}} \tag{2.13}$$

Para finalizar, el aumento de velocidad en la Ley de Amdahl es una función de tres variables denotada por $S(P,b,s)$. Donde, b es la fracción de tiempo durante la cual la ejecución secuencial de P puede beneficiarse de múltiples unidades de procesamiento. Si las unidades de procesamiento múltiples están realmente disponibles y son explotadas por P, la parte de P que las explota se acelera por el factor $s > 1$. Dado que s es solo el aumento de velocidad de una parte del programa P, el aumento de velocidad de todo P no puede ser mayor que s; en concreto, viene dado por $S(P)$ de la Ley de Amdahl, de donde se deduce.

$$S < \frac{1}{1 - b} \tag{2.14}$$

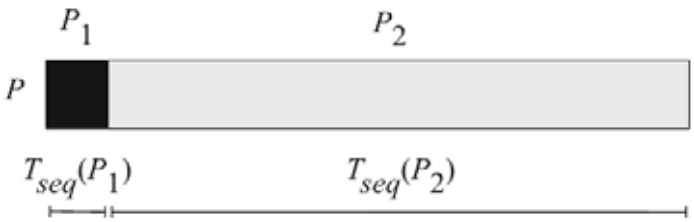


Fig. 2.15 El programa P se compone de un P_1 no paralelizable y un P_2 paralizable

Fuente: Trobec et al. (2018)

2.7 Modelos de isla

En el modelo de isla, también conocido como *distributed evolutionary algorithm*, *coarse-grained model* o *multi-deme model*, donde la población de cada ejecución se considera como una isla (Sudholt, 2015). Esté modelo también puede ser aplicado en la computación paralela para algoritmos bioinspirados y evolutivos. El proceso de comunicación de las islas se lleva a cabo entre las islas que se encuentran conectadas, esta conexión (unidireccional o bidireccional) puede ser de topología dinámica o estática (Cantú-Paz, 1998). La finalidad de dicha conexión es para intercambiar individuos (migración) entre las islas. Este proceso es un factor importante para mejorar la precisión y el rendimiento del algoritmo bioinspirado paralelo, porque determina que tan rápido o lento se propaga una buena solución.

En el modelo de isla hay un operador fundamental llamado migración que se aplica periódicamente para transferir soluciones de una isla a otra. Este movimiento de soluciones está condicionado a un conjunto de reglas conocido como *política de migración* y comprende de parámetros que son definidos por el usuario. También este movimiento de soluciones impacta en el resultado final del problema porque promueve la interacción entre las soluciones candidatas. (Duarte et al., 2017).

En la literatura se puede encontrar varios trabajos que aplican los modelos de isla. Andalon-Garcia and Chavoya (2012) utilizan las topologías estáticas (estrella, anillo unidireccional, anillo bidireccional), aplicado al algoritmo genético para *clustering*. da Silveira et al. (2017) utilizan las topologías estáticas (grafo completo y anillo unidireccional), estas topologías son aplicados sobre un algoritmo genético con intercambio de individuos para genomas no firmados. Lynn et al. (2018) utilizan las topologías estáticas (anillo, grafo completo, *wheels*, von Neumann (red), estrella) y topologías dinámicas con una comunicación entre los nodos de forma aleatoria, estas topologías son aplicados para paralelizar el algoritmo PSO. da Silveira et al. (2018) emplean las topologías estáticas (toroide, árbol, red $4x3$) y topologías dinámicas (comunicación entre similares, comunicación entre buenos-malos y comunicación aleatoria), estas topologías son aplicados sobre un algoritmo genético para la clasificación de genomas no firmados. da Silveira et al. (2019a) utilizan las topologías estáticas (anillo, toroide, árbol, grafo completo, red XxY) y topologías dinámicas (comunicación entre similares, comunicación entre buenos-malos y comunicación aleatoria), estas topologías son aplicados sobre algoritmos genéticos para los casos de estudio relacionados con el problema de distancia de reversión, problema de distancia de translocación, mapeo y programación de tareas y para el problema de las N-reinas. da Silveira et al. (2019c) emplean las topologías estáticas (toroide, árbol, red $4x3$) y topologías dinámicas (comunicación entre similares, comunicación entre buenos-malos y comunicación aleatoria), estas topologías son aplicados sobre un algoritmo genético para problemas NP-*hard*.

A continuación se describe los parámetros de migración encontrados en la literatura para que los modelos de isla intercambien soluciones entre las islas.

- Número de islas: Número de procesadores del computador o subpoblaciones.
- Topología de migración: Define como están conectadas las islas para establecer una comunicación. Esta conexión puede ser unidireccional o bidireccional.
- Tipo de individuo emigrante: Son los individuos seleccionados para la emigración, quienes serán enviados a la isla objetiva. Estos tipos de individuos pueden ser: **mejores**, **peores** o **aleatorios**.
- Tipo de individuo inmigrante: Son los individuos seleccionados en la isla local para ser reemplazados por los individuos inmigrantes. Estos tipos de individuos pueden ser: **mejores**, **peores** o **aleatorios**.
- Política de emigración: Indica que los individuos emigrantes se pueden **clonar** o **remover**.
- Política de inmigración: Indica que los individuos inmigrantes pueden **reemplazar** a los individuos de la isla local. Si se elige la política de emigración **remover**, entonces los individuos inmigrantes sustituyen los espacios libres existentes en la isla local. Por el contrario, si se elige la política de migración **clonar**, entonces los individuos inmigrantes reemplazan a los individuos de la isla local según el tipo de individuo inmigrante.
- Número de emigrantes/inmigrantes: Se refiere a la cantidad de individuos que pueden ser enviados y recibidos de una población a otra.
- Intervalo de migración: Es la frecuencia con la que se puede procesar un intercambio de individuos entre una población y otra, tomando en consideración la política de emigración e inmigración.

2.8 *Message Passing Interface* (MPI)

Es una interfaz de paso de mensajes estándar que define correctamente la sintaxis y la semántica completa de una biblioteca. Por lo tanto, permite construir programas paralelos que intercambian datos, enviando y recibiendo mensajes con estos datos encapsulados. Las comunicaciones pueden ser síncronas o asíncronas, almacenadas o no almacenadas y se

pueden definir barreras (*MPI_Barrier*) de sincronización donde todos los procesos tienen que esperar el uno al otro antes de continuar con las tareas (Nielsen, 2016).

Los mecanismos de comunicación tienen cuatro modos: *standard*, *buffered*, *synchronous* y *ready*. Estos modos pueden ser bloqueantes y no bloqueantes (Trobec et al., 2018). El mecanismo de comunicación básico en modo *standard*, dispone de las operaciones de enviar (*MPI_Send*) y recibir (*MPI_Recv*), estas operaciones son bloqueantes porque la operación de envío espera a que los datos del mensaje se hayan almacenado de forma segura y la operación de recepción hasta que encuentre un mensaje almacenado en el *buffer* (Pacheco, 2011; Message Passing Interface Forum, 2012; Trobec et al., 2018).

2.8.1 Tipos de datos en MPI

Las operaciones de comunicación de MPI especifican la longitud de los datos del mensaje en términos de número de elementos de datos, no en términos de número de bytes. Dado que los tipos de datos en el lenguaje de programación C (*int*, *double*, *char*, etc.) no se pueden pasar como argumentos a las funciones (operadores) de MPI, por lo cual MPI define un tipo especial de datos denominado MPI *data type* (Pacheco, 2011; Trobec et al., 2018). En la Tabla 2.1 se enumera los tipos de datos predeterminados en MPI.

Tabla 2.1 Tipos de datos en MPI correspondiente a los tipos de datos en C

MPI *data type*	**C *data type***
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	usigned char
MPI_UNSIGNED_SHORT	usigned short int
MPI_UNSIGNED	usigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	N/A
MPI_PACKED	N/A

2.8.2 Operadores básicos en MPI

En MPI un comunicador (*communicator*) es una colección de procesos que pueden enviar mensajes entre ellos. Existen cuatro operaciones básicas en MPI: iniciar (*MPI_Init*), terminar el entorno MPI (*MPI_Finalize*), determinar el total de procesos (*MPI_Comm_Size*) y determinar el identificador de un proceso (*MPI_Comm_Rank*). Estas cuatro operaciones son llamadas desde todos los procesos del comunicador actual (Pacheco, 2011; Trobec et al., 2018).

a) *MPI_Init*: La operación inicializa la biblioteca y un entorno MPI, los argumentos *argc* y *argv* solo se requieren para la vinculación con el lenguaje C. A continuación de muestra la sintaxis.

```
int MPI_Init (
  int* argc,          /* in/out */
  char*** arbv        /* in/out */
);
```

b) *MPI_Finalize*: La operación cierra el entorno de MPI, ninguna rutina MPI puede ser llamada antes de la operación *MPI_Init* o después de *MPI_Finalize*. A continuación de muestra la sintaxis.

```
int MPI_Finalize (void);
```

c) *MPI_Comm_Size*: La operación determina el número de procesos en el comunicador actual. El argumento de entrada *comm* es el *handle* del comunicador; el argumento de salida *size* devuelto por la operación *MPI_Comm_Size* es el número de procesos en el grupo de *communicator*. A continuación de muestra la sintaxis.

```
int MPI_Comm_size (
  MPI_Comm communicator,          /* in  */
  int* size                       /* out */
);
```

d) *MPI_Comm_Rank*: La operación determina el identificador del proceso actual dentro de un comunicador. El argumento de entrada *communicator* es el *handle* del comunicador; el argumento de salida *rank* es un identificador del proceso desde el *communicator*, que está en el rango de 0 a $size - 1$. A continuación de muestra la sintaxis.

```
int MPI_Comm_Rank (
  MPI_Comm communicator,          /* in  */
  int* rank                       /* out */
);
```

2.8.3 Comunicación proceso a proceso con MPI

El modelo de paso de mensajes formaliza la comunicación entre procesos que tienen espacios de dirección separados. La comunicación de proceso a proceso tiene que llevar a cabo dos tareas esenciales: el movimiento de datos y la sincronización de los procesos; por lo tanto, requiere la cooperación de los procesos emisores y receptores. Por consiguiente, toda operación de envío espera una operación de emparejamiento/recepción (Pacheco, 2011; Trobec et al., 2018). En la Figura 2.16 se muestra en proceso de envío y recepción de mensajes entre el procesador emisor (*Process_0*) y el proceso receptor (*Process_1*).

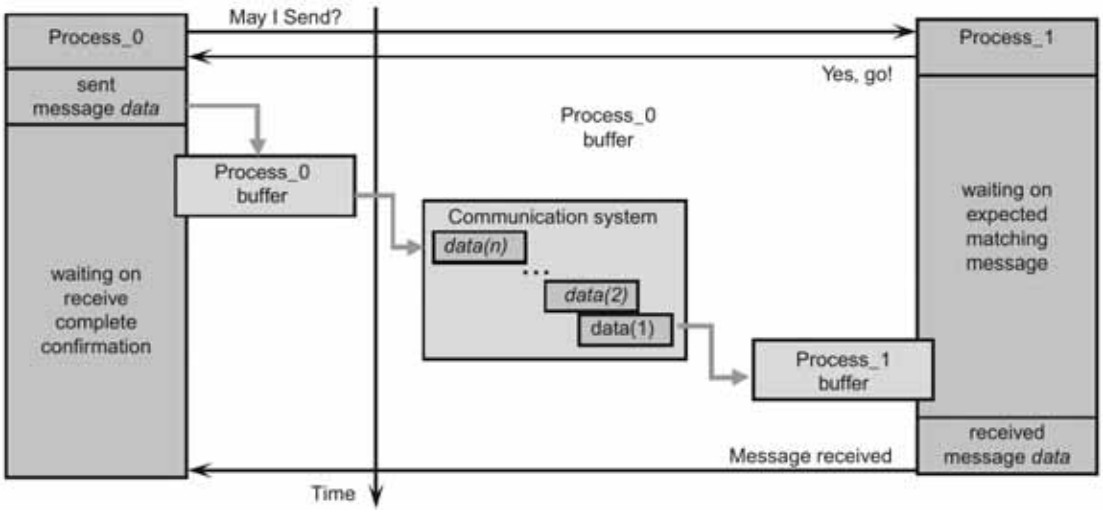


Fig. 2.16 Comunicación básica entre dos procesos

Fuente: Trobec et al. (2018)

a) *MPI_Send*: La operación bloqueante *MPI_Send* en el proceso emisor, no se completará hasta que haya un *MPI_Recv* coincidente (*matching*) en el proceso receptor, identificado con un *rank* correspondiente. La operación *MPI_Send* retornará cuando los datos del mensaje hayan sido entregados al sistema de comunicación y el *buffer* de envío del proceso emisor pueda ser reutilizado. El primer argumento, *msg_buffer*, es un puntero al bloque de memoria que contiene el contenido del mensaje. El segundo y

tercer argumento, *msg_size* y *msg_type*, determinan la cantidad y el tipo de dato que se enviara. El cuarto argumento, *dest*, es el identificador del proceso receptor. El quinto argumento, *tag*, proporciona un mecanismo para distinguir entre diferentes mensajes para el mismo proceso receptor. Finalmente, el argumento *communicator* alberga a todos los procesos en el comunicador. A continuación de muestra la sintaxis.

```
int MPI_Send (
  void* msg_buffer,          /* in */
  int msg_size,              /* in */
  MPI_Datatype msg_type,     /* in */
  int dest,                  /* in */
  int tag,                   /* in */
  MPI_Comm communicator      /* in */
);
```

b) *MPI_Recv*: Esta operación espera hasta que el sistema de comunicación entregue un mensaje con el tipo de dato, la fuente, la etiqueta y el comunicador correspondiente. Entonces, los tres primeros argumentos especifican la memoria disponible para recibir el mensaje: *msg_buffer* puntero al bloque de memoria, el *buffer_size* determina el número de objetos que pueden almacenarse en el bloque y el *buffer_type* indica el tipo de los objetos. Los siguientes tres argumentos identifican al mensaje. El argumento *source* especifica el proceso desde el cual se debe recibir el mensaje. El argumento *tag* debe coincidir con el argumento *tag* del mensaje que se está enviando y finalmente, el argumento *communicator* debe coincidir con el comunicador utilizado por el proxy que lo envía. A continuación de muestra la sintaxis.

```
int MPI_Recv (
  void* msg_buffer,          /* out */
  int buffer_size,           /* in */
  MPI_Datatype buffer_type,  /* in */
  int source,                /* in */
  int tag,                   /* in */
  MPI_Comm communicator,     /* in */
  MPI_Status* status         /* out */
);
```

2.8.4 Comunicación colectiva MPI

Las operaciones colectivas de MPI son llamadas por todos los procesos en un comunicador. Las tareas típicas que pueden realizarse de esta manera son las siguientes: sincronización

global, recepción de datos locales desde todos los procesos que cooperan en el comunicador, envió de datos locales a todos los procesos que cooperan en el comunicador (Pacheco, 2011; Trobec et al., 2018).

a) *MPI_Barrier*: Esta operación se utiliza para sincronizar la ejecución de un grupo de procesos específicos dentro del comunicador *communicator*. Cuando un proceso llega a esta operación, tiene que esperar hasta que todos los demás procesos hayan llegado a la *MPI_Barrier*. A continuación de muestra la sintaxis.

```
int MPI_Barrier (
  MPI_Comm communicator      /* in */
);
```

b) *MPI_Bcast*: En esta operación el proceso emisor con identificador *source*, envía el contenido de la memoria referenciada por *data* a todos los procesos en el comunicador *communicator*, incluso a sí misma. Cada proceso recibe estos datos del proceso *source*, este proceso puede tener cualquier identificador (*rank*). Como *data* se utiliza como un argumento de entrada en el proceso *source*, pero como un argumento de salida en todos los procesos restantes, entonces *data* es del tipo INOUT (entrada y salida). A continuación de muestra la sintaxis.

```
int MPI_Bcast (
  void* data,                /* in/out */
  int count,                 /* in */
  MPI_Datatype datatype,     /* in */
  int source,                /* in */
  MPI_Comm communicator      /* in */
);
```

MPI implementa más operaciones colectivas, pero por la naturaleza de este trabajo solo se utilizó los mencionados en la sección anterior. Sin embargo, a continuación enumeramos los operadores encontrados en la literatura: *MPI_Gatter*, *MPI_Allgatter*, *MPI_Scatter*, *MPI_Reduce*, *MPI_Allreduce*.

2.9 Antecedentes de la investigación

En la literatura se puede encontrar algoritmos como por ejemplo: *Social Spider Optimization* (SSO), *K-means*, *Artificial Bee Colony* (ABC), *Particle Swarm Optimization* (PSO), *Genetic Algorithms* (GA). Estos algoritmos son aplicados para resolver problemas de *clustering* de grandes y pequeños volúmenes de datos.

Inicialmente Cuevas et al. (2013), proponen un algoritmo que consiste en simular el comportamiento e interacción de las arañas hembra ($65\% - 90\%$ de la población) y macho basado en las leyes biológicas de una colonia de arañas. El objetivo de esta investigación es solucionar tareas de optimización. Las pruebas se realizaron con 19 funciones *benchmark*, los resultados obtenidos en la comparación con PSO y ABC confirmaron un rendimiento aceptable del método propuesto en términos de calidad de la solución. Posteriormente León et al. (2020) propuso un algoritmo (EMAX) basado en el algoritmo de *k-means*, para solucionar problemas similares.

Vera-Olivera et al. (2016), proponen una adaptación del algoritmo bioinspirado SSO, con el objetivo de solucionar problemas de *clustering*. Este nuevo enfoque toma como métrica la suma de distancias Euclidianas. Las pruebas se realizaron con 5 *dataset* tomados del repositorio UCI *Machine Learning Repository*, los resultados obtenidos en la comparación demuestran la superioridad del algoritmo propuesto con respecto al algoritmo de *k-means* y al algoritmo genético, estos resultados se obtuvieron realizando un análisis estadístico.

Chandran et al. (2016), realizan un estudio del algoritmo SSO orientado a resolver problemas de *clustering* en documentos de texto, con el objetivo de minimizar la distancia entre *clusters* de documentos. Los experimentos se realizaron con 4 *dataset* de la colección *Patent Corpus*. También efectuaron una comparación entre los algoritmos SSO para *clustering* con el algoritmo de *K-means*. Los resultados demostraron que el algoritmo SSO es más óptimo que el algoritmo de *K-means*, pero se requiere mayor número de iteraciones. Sin embargo, el algoritmo *K-means* converge más rápido que su contra parte.

En su investigación ElSoud and Anter (2016), efectúan un estudio acerca de la detección de tumores cancerígenos, con la finalidad de ayudar a los radiólogos a detectar con precisión la anormalidad. Para detectar estas anormalidades plantearon 5 fases: extracción de características, normalización, selección de características, clasificación, análisis y evaluación. En la fase de selección de características proponen un algoritmo basado en el algoritmo SSO, donde el cálculo de la función *fitness* se determina utilizando el algoritmo *K-nearest neighbor* (KNN). En los experimentos se tomaron imágenes de rayos-X y se comparó el algoritmo propuesto con los algoritmos: *Gray Level Co-occurrence Matrix* (GLCM), *First Order Statistics* (FOs), *Local Binary Pattern* (LBP) y *Segmentation-based Fractal Texture Analysis* (SFTA). Llegando a la conclusión que el algoritmo SSO brinda mejores características sin ruido y una precisión óptima.

Zhou et al. (2017), proponen el algoritmo *Simplex Method-Based Social Spider Optimization* (SMSSO), este método aumenta la diversidad de una población a medida que mejora la capacidad de búsqueda en el tiempo, también ayuda a mejorar la tasa de convergencia del algoritmo propuesto. Las pruebas se realizaron con 11 *dataset* del repositorio UCI *Machine*

Learning Repository, en las comparaciones con los algoritmos SSO, ABC, *Line Search-Particle Swarm Optimisation* (CLSPSO), *Flower Pollination Algorithm* (FPA) y *k-means*; dieron como resultado que SMSSO genera un mejor rendimiento.

Thalamala et al. (2018), aplican el algoritmo *Social Spider Optimization* (SSO) para solucionar problemas de *clustering* en grandes volúmenes de datos. En la etapa experimental ejecutaron comparaciones del algoritmo SSO con los algoritmos de *k-means*, PSO, *Ant Colony Optimization* (ACO) e *Improved Bee Colony Optimization* (IBCO). También realizaron una comparación de los siguientes algoritmos híbridos: SSO + *k-means* (SSOKC), *Integrated SSOKC* (ISSOKC), e *Interleaved* SSOKC (ILSSOKC) con *k-means* + PSO (KPSO), *k-means* + *Genetic Algorithm* (KGA), *k-means* + *Artificial Bee Colony* (KABC) e *Interleaved k-means* + IBCO (IKIBCO). Comparando la precisión entre los algoritmos SSO, *k-means*, PSO y SSOKC, se encontró que SSOKC supera a los otros tres métodos de agrupamiento debido a su capacidad de explorar un espacio amplio de búsqueda para producir una solución óptima.

En cuanto a los algoritmos paralelos Sudholt (2015), propone los siguientes modelos para paralelizar algoritmos evolutivos: modelo maestro-esclavo, modelo de isla (grafo dirigido), modelo celular (toroide), con el objetivo de reducir el tiempo de ejecución. Los resultados obtenidos en este trabajo plantean que efectivamente estos algoritmos paralelos pueden reducir el tiempo de ejecución, al mismo tiempo conducir a una mayor exploración de individuos y a mejorar la diversidad de la población; en comparación con los algoritmos evolutivos secuenciales. Los resultados también llevaron al descubrimiento de un esquema adaptativo, simple y poderoso para elegir el número de islas.

Shukla and Nanda (2016), implementan una versión paralela del algoritmo SSO denominada como PSSO, esta implementación utiliza una topología maestro-esclavo para realizar el proceso de actualización de la posición de las arañas (hembra, macho dominante y no dominante). Las pruebas se ejecutaron con *dataset* de grandes y pequeñas dimensiones tomadas del repositorio UCI. El algoritmo PSSO se comparó con la versión paralela de *Real-Coded Genetic Algorithm* (RGA) que utiliza una topología maestro-esclavo, también se comparó con la versión paralela de *Adaptive Particle Swarm Optimization* (APSO). Los resultados obtenidos para los *dataset* de pequeñas dimensiones demostró que el algoritmo SSO, presenta un mejor rendimiento en "términos de porcentaje de error de clasificación" con respecto a los algoritmos APSO, ABC, RGA y *k-means*. Para los *datasets* de gran volumen el algoritmo PSSO es aproximadamente 10 veces más rápido que la versión secuencial de SSO.

Capítulo 3

Hipótesis, Variables y Metodología

Las definiciones y los términos utilizados en este capítulo fueron tomados de acuerdo a Arias Gonzales and Covinos (2021).

3.1 Hipótesis

3.1.1 Hipótesis general

El algoritmo *Social Spider Optimizatión* paralelo (P-SSO) para resolver problemas de *clustering*, mejora la precisión de la métrica y el tiempo de ejecución.

3.1.2 Hipótesis específicas

- Los modelos paralelos del algoritmo P-SSO proporcionan mejor precisión de la métrica y tiempo de ejecución que el algoritmo SSO, para resolver problemas de *clustering* en *datasets* de grandes dimensiones.
- Los modelos paralelos del algoritmo P-SSO proporcionan mejor precisión de la métrica y tiempo de ejecución que el algoritmo SSO, para resolver problemas de *clustering* en *datasets* de dimensiones pequeñas.
- En promedio los modelos paralelos del algoritmo P-SSO se ejecuta 2 veces más rápido que el algoritmo SSO, para un mismo conjunto de datos.
- El modelo paralelo con topología estática grafo completo, genera mejores resultados en cuanto a la precisión de la métrica.

3.2 Identificación de variables

Las variables identificadas de acuerdo al objetivo general y a la hipótesis general son las siguientes:

- Variable 1: Tamaño del *dataset*.
- Variable 2: Valor de la métrica.
- Variable 3: Tiempo de ejecución.

En la Tabla 3.1 se muestra el tipo de variable según su naturaleza, complejidad y según su función o finalidad. Estas variables permitirán verificar si se cumplió con el objetivo general y también aceptar o refutar la hipótesis general.

Tabla 3.1 Identificación de variables

Variables	**Según su naturaleza**	**Según su complejidad**	**Según su función**
Tamaño del *dataset*	cuantitativa discreta	simple	independiente
Valor de la métrica	cuantitativa continua	simple	dependiente
Tiempo de ejecución	cuantitativa continua	simple	dependiente

3.3 Tipo y nivel de la investigación

3.3.1 Tipo de investigación

El tipo de investigación de este proyecto, según su finalidad es una investigación aplicada, porque mediante el uso de la teoría se encarga de descubrir y solucionar lo planteado en el objetivo del estudio.

3.3.2 Nivel de investigación

El nivel de investigación de este proyecto, por su alcance es explicativo, porque establece una característica de causa-efecto entre la variable independiente (causa) y las variables dependientes (efectos) identificadas en la sección 3.2. Para establecer esta causalidad se requiere realizar un análisis estadístico entre estas variables.

3.4 Técnicas de recolección de información

La captura de datos para la etapa experimental del proyecto se tomó del repositorio UCI *Machine Learning Repository* (Dua and Graff, 2017). La característica principal de estos datos es que cumpla una tarea predeterminada de clasificación. Para descargar estos *datasets* se puede visitar la siguiente dirección web: https://archive.ics.uci.edu/ml/datasets.php.

3.5 Metodología de investigación

La metodología aplicada a una investigación, consta de estrategias, procedimientos y pasos ordenados que se realiza para cumplir con la solución del problema general y también cumplir con el objetivo general. Este trabajo consta de los siguientes pasos divididos en cinco fases: revisión de la literatura, diseño e implementación de los algoritmos, experimentos, análisis estadísticos y finalmente la fase de discusión de los resultados. En la Figura 3.1, se muestra la metodología planteada para abordar el proyecto de investigación.

Revisión de la literatura

En esta fase se realiza una exhaustiva revisión de la literatura, con base en las palabras claves: *Bio-inspired algorithm*, *Social Spider Optimization*, *Parallel algorithms*, *Island models* y *clustering*. El objetivo de esta fase es encontrar en que estado se encuentra el tema que se quiere investigar. El resultado de esta fase permitirá tener un marco conceptual adecuado y unos antecedentes que respalden el proyecto de investigación.

Diseño e implementación de los algoritmos

En esta fase se realiza el diseño e implementación de los algoritmos SSO y P-SSO, para lo cual se utilizara el lenguaje de programación C y la API *Message Passing Interface* (MPI). Esta API permitirá realizar la interconexión entre los procesadores de acuerdo a la topología de interconexión elegida (estática o dinámica).

Experimentos y resultados

En esta fase se realizan los experimentos con *datasets* de grandes y pequeñas dimensiones, estos datos serán tomados del repositorio UCI *Machine Learning Repository*. El objetivo de esta fase es verificar la hipótesis planteada en la sección 3.1. La comparación entre los algoritmos SSO y P-SSO se efectuará mediante un análisis estadístico sobre los resultados.

Discusión de los resultados

En esta fase se presenta formalmente los resultados y conclusiones a los que se llegó en el proyecto de investigación. Esta fase estará validado por el análisis estadístico efectuado en la fase anterior.

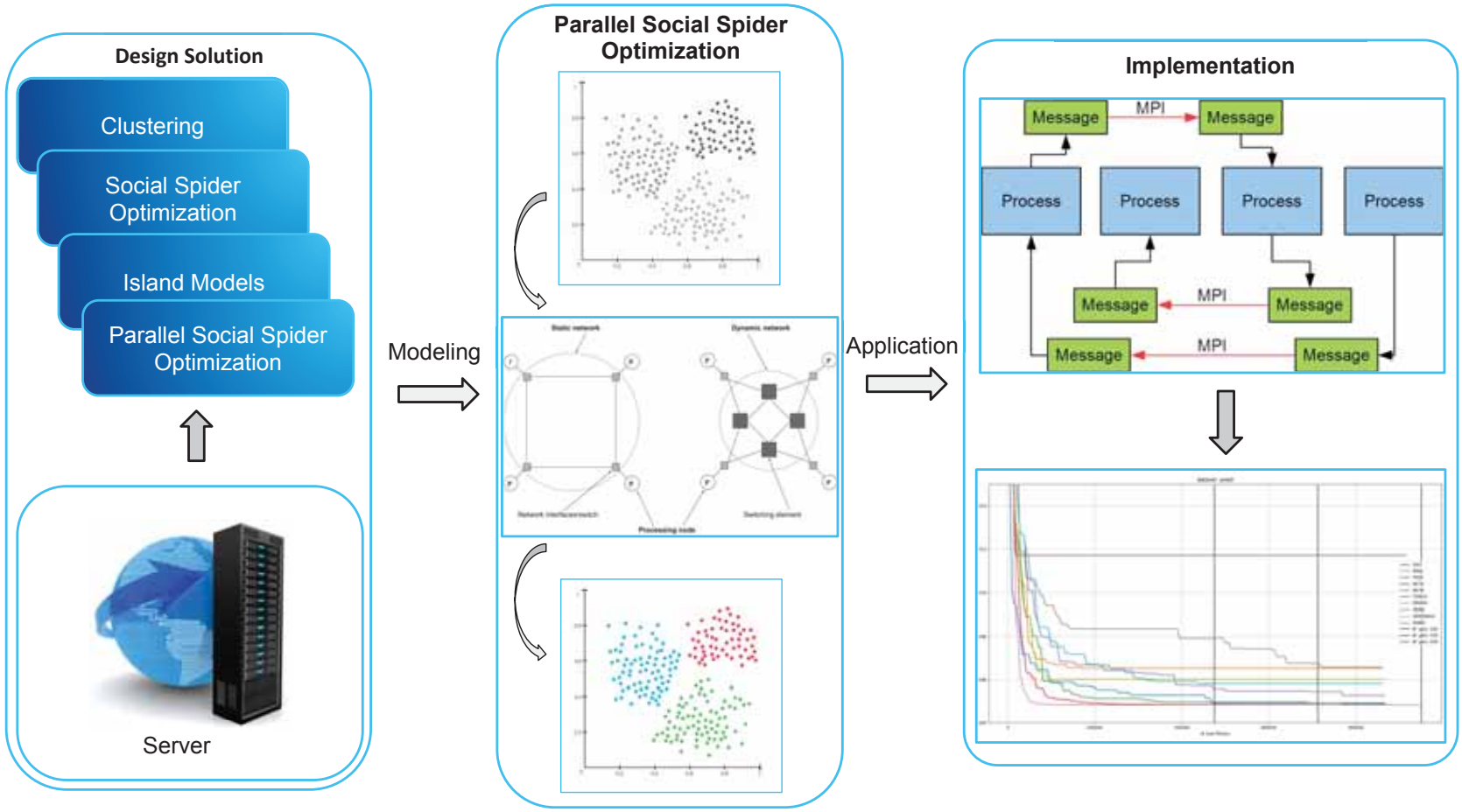


Fig. 3.1 Diagrama ilustrativo de la metodología propuesta

Fuente: Elaboración propia

Capítulo 4

Algoritmo *Social Spider Optimization* Paralelo

4.1 Algoritmo SSO paralelo (P-SSO)

En esta etapa del trabajo de investigación se explica de manera detalla, los algoritmos, procedimientos, procesos, los cuales se llevaron a cabo para cumplir con los objetivos propuestos. Este contenido está dividido en dos partes, modelos de isla para topologías estáticas y para topologías dinámicas. El modelo de isla aplicado para la implementación de los algoritmos consta de 24 islas (procesadores). Finalmente, el mecanismo para identificar a los mejores o peores individuos de una población, se determinó de acuerdo con la métrica generada de la suma de las distancias Euclidianas (Vera-Olivera et al., 2016).

Para la implementación del algoritmo P-SSO, se utilizó la API *Message Passing Interface* (MPI) en el lenguaje de programación C. Cada isla ejecuta el mismo algoritmo con una población inicial distinta para cada isla. Este Algoritmo 2, es la versión paralela del Algoritmo 1 y por consiguiente son similares. A continuación se describen las líneas de código que permiten paralelizar el Algoritmo 2.

En la línea 1 se inicializa el bloque a paralelizar, la línea 2 obtiene el identificador del proceso actual, en la línea 3 se obtiene el número total de procesos, la línea 15 bloquea a todos los procesos hasta que el comunicador haya alcanzado esta rutina, en la línea 16 se realiza el proceso de migración de acuerdo con los parámetros de migración (Tabla 5.1), en el bloque de código a partir de la línea 17 hasta la 20 el proceso 0 recibe la mejor métrica que generó el resto de los procesos y selecciona la mejor métrica para retornar como resultado, en la línea 22 los procesos restantes envían su mejor métrica hacia el proceso 0, finalmente en la línea 23 se cierra el bloque a paralelizar.

Algoritmo 2: SSO paralelo (P-SSO)

Entrada: Un *dataset D* de m puntos n-dimensionales $D = \{d_1, d_2, \ldots, d_m\}$; un entero $k > 0$ que representa el número de *clusters*; *numeroGeneraciones*; *parametros* de migración

Salida: Métrica $\mathscr{M}$ de los *clusters* encontrados

```
1  MPI_Init;
2  MPI_Comm_rank obtener el identificador del proceso (rank);
3  MPI_Comm_size obtener el número de procesos (size);
4  Leer dataset D;
5  Generar población inicial P;
6  Calcular fitness de la población P;
7  Calcular peso de la población P;
8  para i ← 2 hasta numberGenerations hacer
9      Operador cooperativo para arañas hembra;
10     Operador cooperativo para arañas macho;
11     Operador de apareamiento;
12     Substitución de arañas en P;
13     Calcular fitness de la población P;
14     Calcular peso de la población P;
15     MPI_Barrier;
16     Ejecutar las migraciones según los parametros (Alg. 5, 6. 7, 8, 9, 10, 12, 13, 14);
17 si rank == 0 entonces
18     para i ← 1 hasta size − 1 hacer
19         MPI_Recv la mejor métrica M desde el proceso i;
20     Seleccionar la mejor métrica M para devolver;
21 en otro caso
22     MPI_Send la mejor métrica M al proceso 0;
23 MPI_Finalize;
   // Retornar el fitness (métrica M) de la mejor solución
24 devolver M;
```

Además para el intercambio de individuos que requieren los algoritmos ya sea del modelo de isla con topologías estáticas (Alg. 5, 6. 7, 8, 9, 10) o dinámicas (Alg. 12, 13, 14), se implementó los Algoritmos 3 y 4, el Algoritmo 3 cumple la función de empaquetar los individuos para la emigración de acuerdo con los parámetros establecidos en la Tabla 5.3. Estos individuos seleccionados (buenos, malos o aleatorios) se almacenan en el arreglo (*puntos*), cuyo tamaño está definido por: $length = numEI * numC * numCD$, donde *numEI* representa al número de individuos para la migración, *numC* hace referencia a la cantidad de *clusters* y finalmente *numCD* es la cantidad de columnas que tiene el *dataset* que se está analizando.

Una vez que se haya enviado estos individuos a una determinada isla, el Algoritmo 4 se encarga de desempaquetar estos individuos y de acuerdo con los parámetros de migración previamente establecidos, los individuos del proceso actual pueden ser remplazados o sustituidos con los individuos que emigran.

Algoritmo 3: Función para empaquetar individuos para la emigración

Entrada: Una *poblacion* de arañas; un arreglo *puntos* de individuos; *parametros* de migración
Salida: Arreglo *puntos* de individuos

```
1 si emigracion == Remover entonces
2     para k ← 0 hasta numeroEmiInm − 1 hacer
3         Remover individuos seleccionados (mejores, peores o aleatorios) del proceso
            actual;
4         Agregar individuos seleccionados a puntos;
5 en otro caso
      // emigración = Clonar
6     para k ← 0 hasta numeroEmiInm − 1 hacer
7         Clonar individuos seleccionados (mejores, peores o aleatorios) del proceso
            actual;
8         Agregar individuos seleccionados a puntos;
9 devolver puntos
```

Algoritmo 4: Procedimiento para desempaquetar individuos en la inmigración

Entrada: Una *poblacion* de arañas; un arreglo *puntos* de individuos; *parametros* de migración
Salida: Reemplazar o restaurar individuos

```
1 si emigracion == Remover entonces
2     para k ← 0 hasta numeroEmiInm − 1 hacer
3         Restaurar individuos seleccionados (mejores, peores o aleatorios) en el
            proceso actual;
4 en otro caso
      // emigración = Reemplazar
5     para k ← 0 hasta numeroEmiInm − 1 hacer
6         Reemplazar individuos seleccionados (mejores, peores o aleatorios) en el
            proceso actual;
```

4.2 P-SSO con topologías estáticas

Las topologías estáticas que se utilizaron fueron: anillo unidireccional ($\mathcal{P}sso_{ur}$), árbol ($\mathcal{P}sso_{tr}$), red-A ($\mathcal{P}sso_{na}$), red-B ($\mathcal{P}sso_{nb}$), toroide ($\mathcal{P}sso_{hasta}$) y grafo completo, en estas cinco últimas topologías se aplicó una comunicación bidireccional ($\mathcal{P}sso_{cg}$) (Andalon-Garcia and Chavoya, 2012; da Silveira et al., 2017, 2018, 2019a, 2019b, 2019c; Lynn et al., 2018).

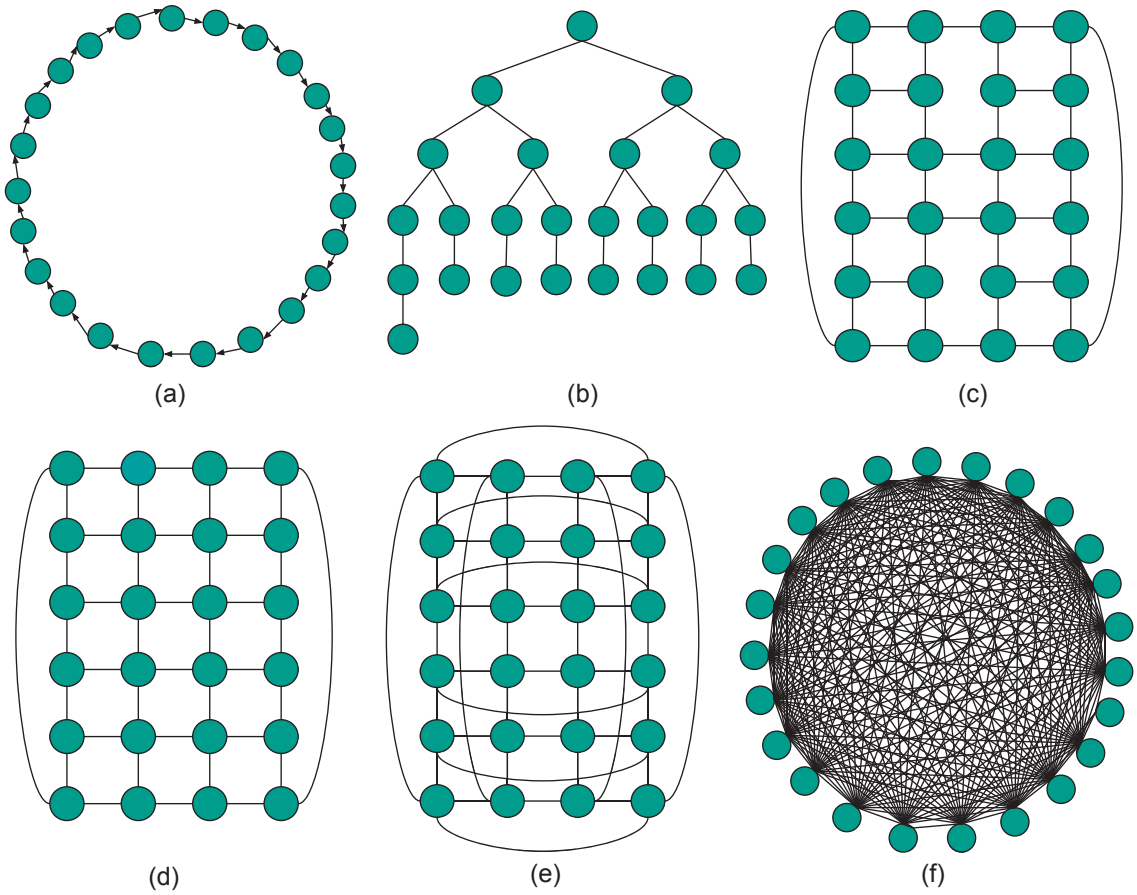


Fig. 4.1 Topologías estáticas: (a) anillo unidireccional, (b) árbol, (c) red-A, (d) red-B, (e) toroide, (f) grafo completo

En la Figura 4.1 se aprecia la comunicación existente entre los nodos y cada uno de estos nodos posee un grado de acuerdo con la topología. En la topología anillo unidireccional los 24 nodos tienen grado 2; la topología árbol tiene 8 nodos con grado 1, 10 nodos con grado 2 y 6 nodos con grado 3; la topología red-A tiene 20 nodos con grado 3 y 4 nodos con grado 4; la topología red-B tiene 16 nodos con grado 3 y 8 nodos con grado 4; en la topología toroide los 24 nodos tienen grado 4; finalmente, la topología grafo completo muestra que sus 24 nodos tienen grado 23 (Ver Tabla 4.1).

Tabla 4.1 Grado de conectividad para topologías estáticas

Topología	**Nro. de nodos**	**Grado**
anillo unidireccional	24	2
árbol	8	1
	10	2
	6	3
red-A	20	3
	4	4
red-B	16	3
	8	4
toroide	24	4
grafo completo	24	23

Cada una de estas topologías envía y recibe a los individuos seleccionados, la comunicación para este intercambio se efectúa entre las islas que mantienen una conexión. Por ello los segmentos de código más importantes en los Algoritmos 5, 6, 7, 8, 9 y 10, son las secciones de envío y recepción de individuos seleccionados de acuerdo con los parámetros de migración. Por ejemplo en el Algoritmo 7 entre las líneas 2 y 6 se ejecutan las rutinas de empaquetar, enviar, recibir y desempaquetar a los individuos seleccionados; si el proceso actual tiene un identificador igual a uno ($rank = 1$), entonces este proceso $\mathtt{P}_1$ enviara sus individuos seleccionados a los procesos $\mathtt{P}_{(rank-1+size)\%size}$, $\mathtt{P}_{(rank+1)\%size}$ y $\mathtt{P}_{(rank+4)\%size}$ donde, $size = 24$ y representa al número total de islas. Como la comunicación es bidireccional, entonces el proceso $\mathtt{P}_1$ recibirá a los individuos seleccionados de los procesos $\mathtt{P}_0$, $\mathtt{P}_2$ y $\mathtt{P}_5$ respectivamente.

Algoritmo 5: Topología Anillo Unidireccional

Entrada: Una *poblacion* de arañas; un arreglo *puntos* de individuos; *parametros* de migración

Salida: Intercambio de individuos entre islas

1 Empaquetar individuos para la migración (Alg. 3);
2 MPI_Send *puntos* al proceso $\mathtt{P}_{(rank+1)\%size}$;
3 MPI_Barrier;
4 MPI_Recv *puntos* desde el proceso $\mathtt{P}_{(rank+size-1)\%size}$;
5 Desempaquetar individuos de la inmigración (Alg. 4);

Algoritmo 6: Topología Árbol

Entrada: Una *poblacion* de arañas; un arreglo *puntos* de individuos; *parametros* de migración

Salida: Intercambio de individuos entre islas

si $rank == 0$ **entonces**
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $\mathtt{P}_{(2*rank)+1}$, $\mathtt{P}_{(2*rank)+2}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $\mathtt{P}_{(2*rank)+1}$, $\mathtt{P}_{(2*rank)+2}$;
  Desempaquetar individuos de la inmigración (Alg. 4);
si no, si $rank \geq 1 \wedge rank \leq 6$ **entonces**
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $\mathtt{P}_{(rank-1)/2}$, $\mathtt{P}_{(2*rank)+1}$, $\mathtt{P}_{(2*rank)+2}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $\mathtt{P}_{(rank-1)/2}$, $\mathtt{P}_{(2*rank)+1}$, $\mathtt{P}_{(2*rank)+2}$;
  Desempaquetar individuos de la inmigración (Alg. 4);
si no, si $rank \geq 7 \wedge rank \leq 15$ **entonces**
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $\mathtt{P}_{(rank-1)/2}$, $\mathtt{P}_{rank+8}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $\mathtt{P}_{(rank-1)/2}$, $\mathtt{P}_{rank+8}$;
  Desempaquetar individuos de la inmigración (Alg. 4);
en otro caso
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $\mathtt{P}_{rank-8}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $\mathtt{P}_{rank-8}$;
  Desempaquetar individuos de la inmigración (Alg. 4);

Algoritmo 7: Topología Red-A

Entrada: Una *poblacion* de arañas; un arreglo *puntos* de individuos; *parametros* de migración

Salida: Intercambio de individuos entre islas

si $rank == 1 \vee rank == 2$ **entonces**
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
  Desempaquetar individuos de la inmigración (Alg. 4);
si no, si $rank == 21) \vee rank == 22$ **entonces**
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$;
  Desempaquetar individuos de la inmigración (Alg. 4);
si no, si $rank\%4 == 0$ **entonces**
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $P_{(rank-4+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $P_{(rank-4+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
  Desempaquetar individuos de la inmigración (Alg. 4);
si no, si $(rank+1)\%4 == 0$ **entonces**
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+4)\%size}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+4)\%size}$;
  Desempaquetar individuos de la inmigración (Alg. 4);
en otro caso
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
  Desempaquetar individuos de la inmigración (Alg. 4);

Algoritmo 8: Topología Red-B

Entrada: Una *poblacion* de arañas; un arreglo *puntos* de individuos; *parametros* de migración

Salida: Intercambio de individuos entre islas

si $rank == 1 \vee rank == 2$ **entonces**
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
  Desempaquetar individuos de la inmigración (Alg. 4);
si no, si $rank == 21) \vee rank == 22$ **entonces**
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$;
  Desempaquetar individuos de la inmigración (Alg. 4);
si no, si $rank\%6 == 0 \vee rank\%4 == 0$ **entonces**
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $P_{(rank-4+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $P_{(rank-4+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
  Desempaquetar individuos de la inmigración (Alg. 4);
si no, si $rank == 9 \vee rank == 10 \vee rank == 13 \vee rank == 14$ **entonces**
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
  Desempaquetar individuos de la inmigración (Alg. 4);
en otro caso
  Empaquetar individuos para la migración (Alg. 3);
  MPI_Send *puntos* al proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+4)\%size}$;
  MPI_Barrier;
  MPI_Recv *puntos* desde el proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+4)\%size}$;
  Desempaquetar individuos de la inmigración (Alg. 4);

Algoritmo 9: Topología Toroide

Entrada: Una *poblacion* de arañas; un arreglo *puntos* de individuos; *parametros* de migración

Salida: Intercambio de individuos entre islas

1 **si** $rank\%4 == 0$ **entonces**
2 Empaquetar individuos para la migración (Alg. 3);
3 MPI_Send *puntos* al proceso $P_{(rank-4+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+3)\%size}$, $P_{(rank+4)\%size}$;
4 MPI_Barrier;
5 MPI_Recv *puntos* desde el proceso $P_{(rank-4+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+3)\%size}$, $P_{(rank+4)\%size}$;
6 Desempaquetar individuos de la inmigración (Alg. 4);
7 **si no, si** $rank\%4 == 3$ **entonces**
8 Empaquetar individuos para la migración (Alg. 3);
9 MPI_Send *puntos* al proceso $P_{(rank-4+size)\%size}$, $P_{(rank-3+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+4)\%size}$;
10 MPI_Barrier;
11 MPI_Recv *puntos* desde el proceso $P_{(rank-4+size)\%size}$, $P_{(rank-3+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+4)\%size}$;
12 Desempaquetar individuos de la inmigración (Alg. 4);
13 **en otro caso**
14 Empaquetar individuos para la migración (Alg. 3);
15 MPI_Send *puntos* al proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
16 MPI_Barrier;
17 MPI_Recv *puntos* desde el proceso $P_{(rank-4+size)\%size}$, $P_{(rank-1+size)\%size}$, $P_{(rank+1)\%size}$, $P_{(rank+4)\%size}$;
18 Desempaquetar individuos de la inmigración (Alg. 4);

Algoritmo 10: Topología Grafo Completo

Entrada: Una *poblacion* de arañas; un arreglo *puntos* de individuos; *parametros* de migración

Salida: Intercambio de individuos entre islas

```
1 Empaquetar individuos para la migración (Alg. 3);
2 para i ← 0 hasta size − 1 hacer
3     si rank ≠ i entonces
4         MPI_Send puntos al proceso P_i;
5 MPI_Barrier;
6 para i ← 0 hasta size − 1 hacer
7     si rank ≠ i entonces
8         MPI_Recv puntos desde el proceso P_i;
9 Desempaquetar individuos de la inmigración (Alg. 4);
```

4.3 P-SSO con topologías dinámicas

En las islas de comunicación con topologías dinámicas, las islas se conectan punto a punto de manera dinámica, cualquier isla del modelo debe tener la posibilidad de ser elegida como fuente o destino según sea su atractivo. La elección de la isla destino para cada solución migratoria se basa en los pesos de las conexiones. (Duarte et al., 2017). En esta etapa del proyecto se aplicó lo utilizado en Duarte et al., 2017; da Silveira et al., 2018, 2019a, 2019b, 2019c. Donde, una isla puede ser calificada como: **bueno**, **medio** o **malo**, este valor está asociado a la diversidad de cada isla. Para determinar la calificación de cada isla se utilizó un *ranking* generado por la desviación estándar y el promedio de las mejores métricas.

Las topologías dinámicas utilizadas fueron: comunicación entre similares ($\mathcal{P}sso_{sa}$), comunicación entre buenos y malos ($\mathcal{P}sso_{gb}$), comunicación entre aleatorios ($\mathcal{P}sso_{ra}$) (Duarte et al., 2017; da Silveira et al., 2018, 2019a, 2019b, 2019c); cada una de estas comunicaciones son bidireccionales como se aprecia en la Figura 4.2. Con respecto a la comunicación entre aleatorios, primero se eligen dos grupos de forma aleatoria, entonces se establece una comunicación punto a punto entre estos dos grupos y los nodos del grupo restante también se comunican punto a punto. Por ejemplo en la Figura 4.2 (c) se eligieron los grupos con calificación de buenos y medios, entonces la comunicación punto a punto se realiza entre estos dos grupos y finalmente, el grupo restante (malos) se comunican entre sí.

Con el propósito de determinar la calidad de las islas, se implementó el Algoritmo 11, este algoritmo es ejecutado desde los Algoritmos 12, 13 y 14 cada vez que se requiere efectuar una migración, la frecuencia para ejecutar una migración está definida por el intervalo de migración (Tabla 5.1). A continuación se describe el Algoritmo 11. En las líneas 2 y 3, el

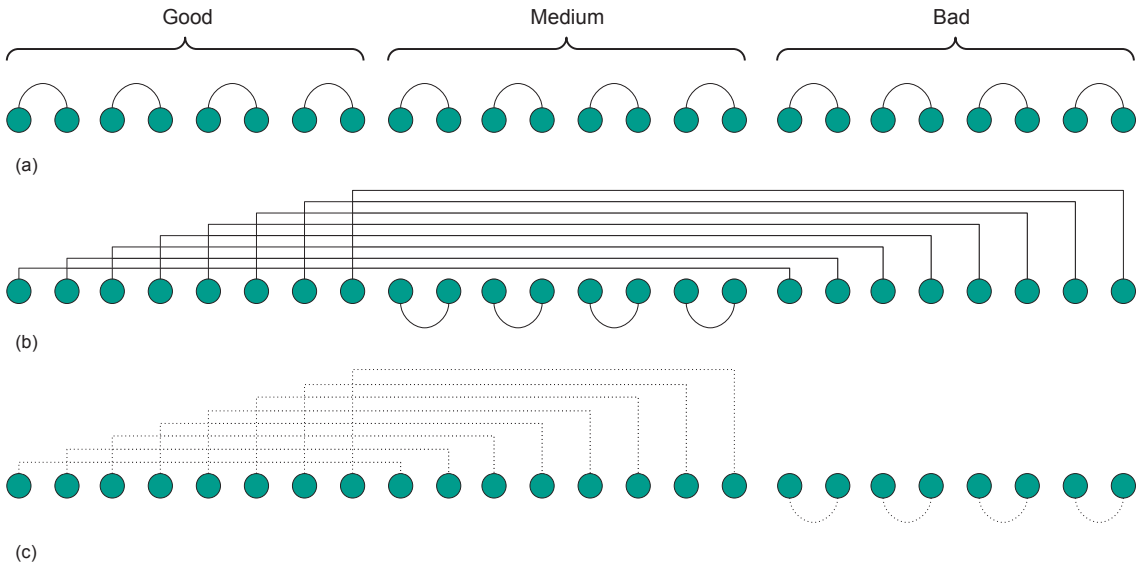


Fig. 4.2 Topologías dinámicas: (a) entre similares, (b) entre buenos y malos, (c) entre aleatorios

proceso $\mathtt{P}_0$ ($rank = 0$) recibe el promedio y la desviación estándar en función del valor de la métrica desde los procesos restantes, con estos datos en la línea 4 se crea un *ranking* de calificación para las 24 islas, este *ranking* se generó a partir de almacenar y ordenar los promedios en un arreglo P de objetos (Ejemplo: $P = [\langle \mathbf{4} : 97.279\rangle, \langle \mathbf{3} : 97.353\rangle, \langle \mathbf{21} : 97.369\rangle, \ldots, \langle \mathbf{22} : 97.696\rangle, \langle \mathbf{9} : 97.745\rangle, \langle \mathbf{18} : 97.748\rangle]$) y en un arreglo S las desviaciones estándar (Ejemplo: $S = [\langle \mathbf{4} : 0.186\rangle, \langle \mathbf{19} : 0.186\rangle, \langle \mathbf{23} : 0.201\rangle, \ldots, \langle \mathbf{21} : 0.315\rangle, \langle \mathbf{20} : 0.321\rangle, \langle \mathbf{14} : 0.33\rangle]$), cada uno de estos valores está asociado con un identificador de isla en particular.

Después se generó un arreglo (*ranking*) a partir de la suma de posiciones que una isla ocupa, tanto en el arreglo P y S ($ranking = [\langle \mathbf{0} : 20\rangle, \langle \mathbf{1} : 29\rangle, \langle \mathbf{2} : 28\rangle, \ldots, \langle \mathbf{21} : 6\rangle, \langle \mathbf{22} : 34\rangle, \langle \mathbf{23} : 27\rangle]$). Finalmente, se ordenó este arreglo de forma ascendente de acuerdo con la suma de posiciones ($ranking = [\langle \mathbf{20} : 6\rangle, \langle \mathbf{21} : 6\rangle, \langle \mathbf{3} : 11\rangle, \ldots, \langle \mathbf{9} : 40\rangle, \langle \mathbf{15} : 41\rangle, \langle \mathbf{18} : 44\rangle]$), donde los 8 primeros elementos son calificados como **buenos**, los siguientes 8 elementos como **medios** y los restantes como **malos**. En las líneas 5 y 6 el proceso $\mathtt{P}_0$ envía el arreglo *ranking* a las demás islas, en las líneas 8 y 9 los procesos con identificador a partir del 1 al 23 envían al proceso $\mathtt{P}_0$, el promedio y la desviación estándar de su respectiva métrica. Sobre la base del arreglo *ranking* se realizó la comunicación punto a punto entre las islas, de acuerdo con las topologías de comunicación: entre similares, entre buenos y malos, entre aleatorios.

Para las topologías dinámicas la comunicación entre islas depende básicamente de la calificación que posee cada isla y de los parámetros de migración. Por consiguiente, los Algoritmos 12, 13 y 14 mantienen cierta similitud. Por esta razón a continuación se explica

Algoritmo 11: Función para generar un *ranking* de islas

Entrada: Una *poblacion* de arañas
Salida: Arreglo *ranking* de islas

```
1  si rank == 0 entonces
2      para i ← 1 hasta size − 1 hacer
3          MPI_Recv avg y stdev desde el proceso P_i;
4      Generar ranking;
5      para i ← 1 hasta size − 1 hacer
6          MPI_Send arreglo ranking al proceso P_i;
7  en otro caso
8      MPI_Send Avg(poblacion) y Stdev(poblacion) al proceso P_0;
9      MPI_Recv arreglo ranking desde el proceso P_0;
10 devolver ranking
```

únicamente el Algoritmo 12. En la línea 1 se genera el *ranking* para calificar a cada isla (esto se explicó en los párrafos anteriores), la línea 2 obtiene el índice correspondiente al proceso actual dentro del arreglo *ranking*, este valor ayuda a encontrar el proceso al cual se envía y del cual se recibe individuos, en la línea 4 se verifica que el índice del proceso actual sea un número par. Por ejemplo, si este índice fuese 20 entonces en la línea 5 el proceso actual envía individuos al proceso con índice 21 (P_9), en la línea 6 el proceso actual recibe individuos del proceso P_9. Para terminar, las líneas 8 y 9 ejecutan la misma función que las líneas 5 y 6, pero serán aplicados para índices impares.

Algoritmo 12: Topología Entre Similares

Entrada: Una *poblacion* de arañas; un arreglo *puntos* de individuos; *parametros* de migración
Salida: Intercambio de individuos entre islas

```
1  Generar ranking de islas (Alg. 11);
2  Obtener el index del arreglo ranking del proceso actual;
3  Empaquetar individuos para la migración (Alg. 3);
4  si index%2 == 0 entonces
5      MPI_Send puntos al proceso P_ranking[index+1];
6      MPI_Recv puntos desde el proceso P_ranking[index+1];
7  en otro caso
8      MPI_Send puntos al proceso P_ranking[index−1];
9      MPI_Recv puntos desde el proceso P_ranking[index−1];
10 Desempaquetar individuos de la inmigración (Alg. 4);
```

Algoritmo 13: Topología Entre Buenos y Malos

Entrada: Una *poblacion* de arañas; un arreglo *puntos* de individuos; *parametros* de migración

Salida: Intercambio de individuos entre islas

1 Generar *ranking* de islas (Alg. 11);
2 Obtener el *index* del arreglo *ranking* del proceso actual;
3 Empaquetar individuos para la migración (Alg. 3);
4 **si** $index \geq 0 \wedge index < 8$ **entonces**
    `// Bueno`
5 MPI_Send *puntos* al proceso $\mathtt{P}_{ranking[index+16]}$;
6 MPI_Recv *puntos* desde el proceso $\mathtt{P}_{ranking[index+16]}$;
7 **si no, si** $index \geq 16 \wedge index < size$ **entonces**
    `// Malo`
8 MPI_Send *puntos* al proceso $\mathtt{P}_{ranking[index-16]}$;
9 MPI_Recv *puntos* desde el proceso $\mathtt{P}_{ranking[index-16]}$;
10 **si no, si** $index\%2 == 0$ **entonces**
    `// Medio`
11 MPI_Send *puntos* al proceso $\mathtt{P}_{ranking[index+1]}$;
12 MPI_Recv *puntos* desde el proceso $\mathtt{P}_{ranking[index+1]}$;
13 **en otro caso**
14 MPI_Send *puntos* to $\mathtt{P}_{ranking[index-1]}$;
15 MPI_Recv *puntos* from $\mathtt{P}_{ranking[index-1]}$;
16 Desempaquetar individuos de la inmigración (Alg. 4);

Algoritmo 14: Topología Entre Aleatorios

Entrada: Una *poblacion* de arañas; un arreglo *puntos* de individuos; *parametros* de migración

Salida: Intercambio de individuos entre islas

1 Generar *ranking* de islas (Alg. 11);
2 Obtener el *index* del arreglo *ranking* del proceso actual;
3 Empaquetar individuos para la migración (Alg. 3);
4 $q1 = Rand(3)$; `// Bueno=0, Medio=1, Malo=2`
5 $q2 = (q1+1)\%3$;
6 $q3 = (q1+2)\%3$;
7 **si** $index \geq q1*8 \wedge index < (q1+1)*8$ **entonces**
8 MPI_Send *puntos* al proceso $\mathtt{P}_{ranking[index+((q2-q1)*8)]}$;
9 MPI_Recv *puntos* desde el proceso $\mathtt{P}_{ranking[index+((q2-q1)*8)]}$;
10 **si no, si** $index \geq q2*8 \wedge index < (q2+1)*8$ **entonces**
11 MPI_Send *puntos* al proceso $\mathtt{P}_{ranking[index-((q2-q1)*8)]}$;
12 MPI_Recv *puntos* desde el proceso $\mathtt{P}_{ranking[index-((q2-q1)*8)]}$;
13 **si no, si** $index\%2 == 0$ **entonces**
14 MPI_Send *puntos* al proceso $\mathtt{P}_{ranking[index+1]}$;
15 MPI_Recv *puntos* desde el proceso $\mathtt{P}_{ranking[index+1]}$;
16 **en otro caso**
17 MPI_Send *puntos* al proceso $\mathtt{P}_{ranking[index-1]}$;
18 MPI_Recv *puntos* desde el proceso $\mathtt{P}_{ranking[index-1]}$;
19 Desempaquetar individuos de la inmigración (Alg. 4);

Capítulo 5

Resultados y Discusión

5.1 Experimentos y resultados

Los algoritmos expuestos en la sección 4.1, fueron implementados usando la interfaz MPI para el lenguaje de programación C y ejecutados en un computador con dos procesadores Intel Xeon Gold 6134s (-MT-MCP-SMP-) de 8 núcleos con *hyper-threading*, con una velocidad máxima de reloj de 3.7 GHz y 128 GB de memoria RAM. Los *dataset* utilizados para estos experimentos se obtuvieron del repositorio UCI *Machine Learning Repositiry*, la característica principal de estos datos es que cumpla con una tarea predeterminada de clasificación. La descarga de estos *dataset* se realizó desde la siguiente URL: https://archive.ics.uci.edu/ml/datasets.php.

Para determinar los mejores parámetros para el algoritmo P-SSO, cada algoritmo se ejecutó 10 veces con cada combinación de los parámetros de la Tabla 5.1 sobre los *dataset* de la Tabla 5.2. A continuación se describe los *dataset* utilizados en este punto.

- crude_oil: N/A.
- caesarian: contiene resultados de las cesáreas de 80 mujeres embarazadas con las características más importantes de los problemas del parto en el ámbito médico.
- breast_tissue: contiene mediciones de impedancia eléctrica de muestras de tejido recién extirpadas de la mama.
- hayes_roth: contiene estudio en seres humanos.
- iris: contiene tipos de plantas de iris.
- tae: contienen evaluaciones del rendimiento docente.

- wine: contienen resultados de un análisis químico de vinos cultivados en la misma región de Italia.
- glass: contiene tipos de vidrio definidos en función de su contenido en óxidos.
- statlog_heart: contienen datos de enfermedades cardíacas.
- haberman: contiene casos de un estudio realizado sobre la supervivencia de pacientes operadas de cáncer de mama.
- column_2c: contiene los valores de seis características biomecánicas utilizadas para clasificar a los pacientes ortopédicos.
- column_3c: contiene los valores de seis características biomecánicas utilizadas para clasificar a los pacientes ortopédicos.
- ecoli: contiene lugares de localización de proteínas.
- ilpd: contiene registros de pacientes hepáticos y no hepáticos.
- balance_scale: contiene datos generados para modelar resultados psicológicos.
- australian_credit: contiene datos de solicitudes de tarjetas de crédito.
- breast_cancer_wisconsin: contiene datos cronológicos de cáncer de mama.
- healthy_older_people: contiene datos de movimiento secuencial de 14 personas mayores sanas de entre 66 y 86 años que utilizan un sensor portátil.
- transfusion: contiene datos de transfusión de sangre de la ciudad de Hsin-Chu.
- vehicle_silhouettes: contiene siluetas de cuatro tipos de vehículos.

Los resultados (métrica) para determinar los mejores parámetros, fueron normalizados utilizando el método *min-max*, este método transforma los datos a un rango entre $[0, 1]$, con esto se generó un *ranking* de las mejores métricas en función a los parámetros, el valor de los parámetros seleccionados se muestran en la Tabla 5.3.

Para los experimentos de los algoritmos SSO (Alg. 1) y P-SSO (Alg. 2), se tomaron en cuenta las siguientes especificaciones: el algoritmo secuencial SSO y los modelos paralelos del algoritmo P-SSO se ejecutaron 50 veces para cada *dataset*, el tamaño de la población para el algoritmo SSO fue de 2400 individuos y para el algoritmo P-SSO fue de 100 individuos por cada isla (24 islas), el número de generaciones para ambos algoritmos fue calculado

Tabla 5.1 Valores estimados para los parámetros del algoritmo P-SSO

Parámetros	**Valores Estimados**
Número de Islas	24
Topología de migración	anillo unidireccional, árbol, red-A, red-B, toroide, grafo completo, entre similares, entre buenos y malos, entre aleatorios
Tipo de individuo emigrante	mejor, peor, aleatorio
Tipo de individuo inmigrante	mejor, peor, aleatorio
Política de emigración	clonar, remover
Política de inmigración	remplazar, restaurar
Nro. de emigrantes/inmigrantes	1, 2, 3, 4, 5
Intervalo de migración	1, 2, 4, 6, 8, 10

Tabla 5.2 Información de los *dataset* para generar los mejores parámetros

Dataset	**Tamaño**	**Nro. de Clases**	**Atributos**
crude_oil	56	3	6
caesarian	80	2	6
breast_tissue	106	6	10
hayes_roth	132	3	6
iris	150	3	5
tae	151	3	6
wine	178	3	14
glass	214	7	11
statlog_heart	270	2	14
haberman	306	2	4
column_2c	310	2	7
column_3c	310	3	7
ecoli	336	8	9
ilpd	583	2	11
balance_scale	625	3	5
australian_credit	690	2	15
breast_cancer_wisconsin	699	2	11
healthy_older_people	724	4	9
transfusion	748	2	5
vehicle_silhouettes	946	4	19

de acuerdo con el número de llamadas a la función *fitness* que la versión secuencial (SSO) obtuvo al ejecutarse por 100 generaciones, con una población de 100 individuos y para un determinado *dataset*. Por ejemplo para el *dataset yeast* (Tabla 5.4) este valor fue de 9900, entonces el algoritmo SSO se ejecutó un número de generaciones equivalentes a $9900x24$ llamadas a la función *fitness* y el algoritmo P-SSO ejecuto 9900 llamadas a la función *fitness* por cada isla, de esta manera también se definió el criterio de parada para ambos algoritmos.

Los algoritmos SSO y P-SSO requieren como parámetros de entrada: el número de generaciones (equivalente al número de llamadas a la función *fitness*), los parámetros de migración (Tabla 5.3), un *dataset* (Tabla 5.4), un número de *clusters* (columna 3 de la Tabla 5.4), donde el algoritmo P-SSO se ejecutó simultáneamente con 24 procesos. Con respecto a los datos para analizar en los experimentos, se consideró como *dataset* de gran volumen a aquellos *dataset* que tengan un tamaño mayor a $10,000$ instancias y los *dataset* con un tamaño menor a $2,000$ instancias fueron considerados como *dataset* de menor volumen. El contenido de estos *dataset* se describe a continuación.

- messidor_features: contiene características extraídas de un conjunto de imágenes para predecir si una imagen contiene signos de retinopatía diabética o no.
- website_phishing: contiene diferentes características relacionadas con los sitios web legítimos y de *phishing*.
- banknote_authentication: contiene datos de imágenes tomadas para la evaluación de un procedimiento de autentificación de billetes.
- cmc: contiene un subconjunto de datos de la Encuesta Nacional de Prevalencia de Anticonceptivos de Indonesia.
- yeast: contiene datos predictivos de la localización celular de las proteínas.
- wifi_localization: contiene datos de intensidad de la señal de wifi visibles en un *smartphone*.
- electrical_grind: contiene datos del análisis de estabilidad local del sistema estrella de 4 nodos, implementando el concepto de control descentralizado de una red inteligente.
- avila_tr: contiene datos normalizados extraídos de imágenes del *dataset* "*Avila Bible*".

En cuanto a los resultados de la experimentación, se calculó el máximo, mínimo, mediana y el promedio de las 50 métricas generadas por la ejecución de los algoritmos para cada *dataset*. El promedio de estos valores se muestran en la Tabla 5.5, donde las celdas en negrita

representan a las mejores métricas. Sin embargo, estos resultados son muy similares a las métricas del algoritmo P-SSO, con una diferencia promedio del 0.15% en el peor de los casos y de $0,0000034\%$ en el mejor de los casos. Esta diferencia se calculó para cada resultado de los modelos paralelos del algoritmo P-SSO con respecto al algoritmo secuencial SSO. Para la mayoría de los casos el algoritmo SSO tiene mejores resultados y en el resto de los casos es superado por los modelos paralelos del algoritmo P-SSO.

Así mismo se calculó el máximo, mínimo, mediana y el promedio del tiempo de ejecución (en segundos) para ambos algoritmos (Tabla 5.6), a su vez con estos datos se calculó el *speedup* de cada modelo paralelo con respecto a la versión secuencial. En la Tabla 5.7 se muestra el resultado de estos análisis, donde las celdas en negrita representan a los mejores resultados. También se puede verificar que para los *dataset* de gran volumen, los modelos paralelos en promedio son 15 veces más rápido que la versión secuencial y en el caso de los *dataset* de menor volumen, en promedio son 28 veces más rápido. El código fuente y los resultados están disponibles en la siguiente URL: https://github.com/win7/parallel_social_spider_optimization.

Tabla 5.3 Configuración de los parámetros para el algoritmo P-SSO

Parámetros	$\mathcal{P}sso_{ur}$	$\mathcal{P}sso_{tr}$	$\mathcal{P}sso_{na}$	$\mathcal{P}sso_{nb}$	$\mathcal{P}sso_{to}$	$\mathcal{P}sso_{cg}$	$\mathcal{P}sso_{sa}$	$\mathcal{P}sso_{gb}$	$\mathcal{P}sso_{ra}$
Tipo de individuo emigrante	mejor	mejor	mejor	mejor	mejor	mejor	mejor	mejor	mejor
Tipo de individuo inmigrante	peor	aleatorio	aleatorio	aleatorio	aleatorio	aleatorio	aleatorio	aleatorio	peor
Política de emigración	clonar	clonar	clonar	clonar	clonar	clonar	clonar	clonar	clonar
Política de inmigración	reemplazar	reemplazar	reemplazar	reemplazar	reemplazar	reemplazar	reemplazar	reemplazar	reemplazar
Nro. de emigrantes/inmigrantes	5	5	2	3	1	1	2	4	2
Intervalo de migración	2	1	1	1	1	2	1	2	1

Tabla 5.4 Información de los *dataset* para los experimentos

Id	**Dataset**	**Tamaño**	**Nro. de Clases**	**Atributos**
d1	messidor_features	1151	2	20
d2	website_phishing	1353	3	10
d3	banknote_authentication	1372	2	5
d4	cmc	1473	3	10
d5	yeast	1484	10	9
d6	wifi_localization	2000	4	8
d7	electrical_grind	10 000	2	14
d8	avila_tr	10 430	12	11
d9	firm_teacher	10 800	4	20

Tabla 5.5 Promedio de las métricas para los algoritmos SSO y P-SSO

	Secuencial	**Paralelo: Topologías Estáticas**						**Paralelo: Topologías Dinámicas**		
Dataset	SSO	$\mathcal{P}sso_{ur}$	$\mathcal{P}sso_{tr}$	$\mathcal{P}sso_{na}$	$\mathcal{P}sso_{nb}$	$\mathcal{P}sso_{to}$	$\mathcal{P}sso_{cg}$	$\mathcal{P}sso_{sa}$	$\mathcal{P}sso_{gb}$	$\mathcal{P}sso_{ra}$
d1	62034.6724998	62037.9297887	62037.5672823	62037.7674620	**62037.5186311**	62037.6355252	62037.6218168	62037.5600500	62037.5735310	62037.8293196
d2	2584.82397500	2584.82403328	2584.82405556	2584.82403328	2584.82403328	2584.82422350	2584.82412240	2584.82403328	**2584.82401100**	**2584.82401100**
d3	7244.71093800	7244.74772896	7244.73387652	7244.74559668	**7244.72662048**	7244.74640968	7244.74311148	7244.74970788	7244.75168680	7244.74970788
d4	5541.64306600	5541.82641924	5541.72186796	5541.70283946	**5541.66884300**	5541.82047436	5541.77271190	5541.79408452	5541.79002632	5541.77961720
d5	205.48207982	205.86546078	205.30025750	**204.88076346**	204.88715628	206.69898988	206.79928670	205.03827250	205.29688400	206.10553062
d6	20377.7676562	20379.8377058	20379.4834358	20379.2734927	**20379.0142634**	20379.6755264	20379.7594524	20379.4991087	20379.5245924	20379.6908146
d7	49356.8770312	49356.9435220	49356.9351743	**49356.9222707**	49356.9258635	49356.9446961	49356.9406542	49356.9296558	49356.9412372	49356.9285692
d8	17626.4801562	17720.1284482	17608.6605286	**17597.4818665**	17598.7845244	17704.1137298	17704.6331956	17604.2260387	17615.7111016	17714.6938877
d9	20175.2939843	20219.8630144	20175.4820928	20175.3178932	20175.5632387	20196.1766053	20203.7600215	20173.7781472	**20173.5332543**	20245.9897578

Tabla 5.6 Promedio de los tiempos de ejecución para los algoritmos SSO y P-SSO

	Secuencial	**Paralelo: Topologías Estáticas**						**Paralelo: Topologías Dinámicas**		
Dataset	SSO	$\mathcal{P}sso_{ur}$	$\mathcal{P}sso_{tr}$	$\mathcal{P}sso_{na}$	$\mathcal{P}sso_{nb}$	$\mathcal{P}sso_{to}$	$\mathcal{P}sso_{cg}$	$\mathcal{P}sso_{sa}$	$\mathcal{P}sso_{gb}$	$\mathcal{P}sso_{ra}$
d1	49.21109284	1.66527098	1.66750098	1.67008292	1.6753086	1.6712932	1.66494606	1.67652414	**1.66420572**	1.68774
d2	47.48454572	**1.472071**	1.47636944	1.47613988	1.47834742	1.4768778	1.48528542	1.48440112	1.48371448	1.4926243
d3	15.54069824	**0.83009212**	0.85822522	0.83504682	0.85889024	0.85407152	0.84611528	0.83171678	0.8428601	0.83214868
d4	44.91542084	1.57555736	**1.56505032**	1.56627152	1.57160934	1.57163234	1.5781549	1.5779954	1.57296886	1.6080564
d5	87.35754414	2.6304727	2.67585478	2.67104066	2.67407552	2.64742996	**2.62962926**	2.66487998	2.66052498	2.66625
d6	51.97155728	1.94407628	**1.91892154**	1.92346662	1.92063722	1.92204442	1.91997914	1.92958554	1.93370278	1.9566194
d7	157.2578567	**9.77014712**	9.78032022	9.80268944	9.7906459	9.79355126	9.78601858	9.7948611	9.78825612	9.7814666
d8	433.07363944	**28.54155504**	28.59168958	28.59367834	28.59191192	28.57174638	28.60023074	28.60234868	28.58940822	28.5709972
d9	251.59643	16.16595492	16.0430815	15.90930872	**15.85012554**	16.19548062	16.17789182	16.1441243	16.1821063	16.3353765

Tabla 5.7 *Speedup* promedio del algoritmo P-SSO con respecto al algoritmo SSO

	Paralelo: Topologías Estáticas						**Paralelo: Topologías Dinámicas**		
Dataset	$\mathcal{P}sso_{ur}$	$\mathcal{P}sso_{tr}$	$\mathcal{P}sso_{na}$	$\mathcal{P}sso_{nb}$	$\mathcal{P}sso_{to}$	$\mathcal{P}sso_{cg}$	$\mathcal{P}sso_{sa}$	$\mathcal{P}sso_{gb}$	$\mathcal{P}sso_{ra}$
d1	29.5514	29.5119	29.4663	29.3743	29.4449	29.5572	29.3530	**29.5703**	29.1580
d2	**32.2570**	32.1631	32.1681	32.1200	32.1520	31.9700	31.9890	32.0038	31.8128
d3	**18.7217**	18.1079	18.6106	18.0939	18.1960	18.3671	18.6851	18.4381	18.6754
d4	28.5076	**28.6990**	28.6767	28.5793	28.5788	28.4607	28.4636	28.5546	27.9315
d5	33.2098	32.6466	32.7054	32.6683	32.9971	**33.2205**	32.7810	32.8347	32.7642
d6	26.7333	**27.0837**	27.0197	27.0595	27.0397	27.0688	26.9341	26.8767	26.5619
d7	**16.0958**	16.0790	16.0423	16.0621	16.0573	16.0696	16.0551	16.0660	16.0771
d8	**15.1734**	15.1468	15.1458	15.1467	15.1574	15.1423	15.1412	15.1480	15.1578
d9	15.5634	215.6826	15.8144	**15.8735**	15.5350	15.5519	15.5844	15.5478	15.4019

5.2 Discusión de resultados

Para respaldar el análisis estadístico sobre los resultados de los algoritmos paralelos de la sección anterior se utilizó las pruebas estadísticas propuestas en Demšar (2006) y aplicados en da Silveira et al. (2017 y 2018). El procedimiento, tal como se propone en Demšar (2006), es el siguiente: primero, se aplica la prueba de Friedman, segundo, la prueba de Holm. Ambas pruebas se pueden encontrar en el software `CONTROLTEST package` disponible en https://sci2s.ugr.es/sicidm.

El nivel de significancia aplicada para las pruebas de Friedman y Holm es del 5%. En la prueba de Friedman se considera como hipótesis nula la igualdad de los resultados (métrica) que generan los algoritmos y como hipótesis alternativa la variación de estos resultados por cada *dataset*. Para la prueba de Holm la hipótesis nula es que un algoritmo de control tenga el mismo desempeño con respecto a cada uno de los algoritmos restantes.

En la prueba de Friedman, la hipótesis nula fue rechazada (p-valor ≤ 0.05) en la evaluación de los *dataset d1*, *d3*, *d4*, *d5*, *d6*, *d7*, *d8*, *d9*, porque se obtuvo un p-valor de $5.83926427078052E-4$ en el peor de los casos. Sin embargo, la hipótesis nula no pudo ser refutada en la evaluación del *dataset d2*, donde se obtiene un p-valor de 0.9983393538102286. Por esta razón se realizó la prueba de Holm para evaluar los *dataset d1*, *d3*, *d4*, *d5*, *d6*, *d7*, *d8*, *d9*, donde la hipótesis nula se rechaza cuando el p-valor $\leq \alpha/i$, el resultado de este análisis se muestran en las Tablas 5.8, 5.9. Donde las celdas en negrita representan la refutación de la hipótesis nula.

El algoritmo de control con mejor diferencia estadística significativa en el 70% de los resultados (Tabla 5.8 y 5.9) es el algoritmo secuencial SSO, por esta razón es más relevante resaltar la comparación entre los modelos paralelos. El modelo paralelo con mejor diferencia significativa es $\mathcal{P}sso_{nb}$ (para 4 de 8 *dataset*), seguido del modelo paralelo $\mathcal{P}sso_{na}$ (para 3 de 8 *dataset*) y finalmente el modelo $\mathcal{P}sso_{gb}$ (para 2 de 8 *dataset*). Esto significa que estos algoritmos tienen un mejor rendimiento con respecto los modelos paralelos restantes. Por el contrario los algoritmos con peor rendimiento fueron $\mathcal{P}sso_{ur}$ (para 4 de 8 *dataset*), $\mathcal{P}sso_{cg}$ (para 3 de 8 *dataset*), $\mathcal{P}sso_{gb}$ (para 1 de 8 *dataset*) y $\mathcal{P}sso_{to}$ (para 1 de 8 *dataset*). De los análisis de rendimiento y precisión se puede inferir que el modelo de isla con topología estática red 6x4 ($\mathcal{P}sso_{nb}$ y $\mathcal{P}sso_{na}$) son mejores que el resto de los modelos paralelos y también mejor que la versión secuencial.

Con respecto a la convergencia de los algoritmos SSO y P-SSO (Figura 5.1a, 5.1b, 5.1c, 5.2a, 5.2b, 5.2c, 5.3a, 5.3b, 5.3c) se puede mencionar que los modelos paralelos convergen de manera lenta y progresiva, mejorando con esto la exploración de la población para encontrar buenas soluciones candidatas y reducir la convergencia prematura a un óptimo local. El

algoritmo SSO tiene peor taza de convergencia debido a la convergencia prematura que posee. Por otro lado los modelos paralelos con topología estática red 6x4 ($\mathcal{P}sso_{nb}$ y $\mathcal{P}sso_{na}$) tienen una taza de convergencia progresiva y homogénea que le permite explorar de mejor manera a la población y encontrar una buena solución. Estos resultados también se muestran en el trabajo desarrollado por (Lynn et al., 2018).

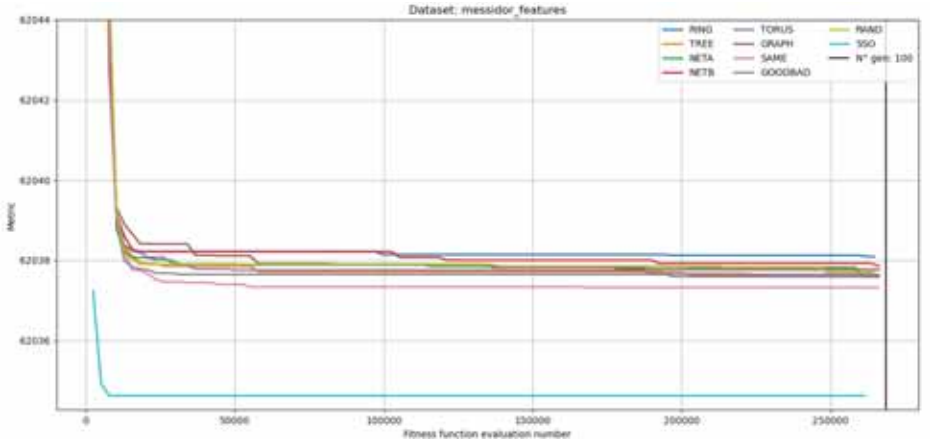
Finalmente, se efectuó una comparación estadística entre el *speedup* de los modelos paralelos y se observó que el modelo paralelo $\mathcal{P}sso_{ur}$ es ligeramente superior a los demás modelos paralelos, debido a su grado de conectividad (los 24 nodos tienen grado 1) con los nodos adyacentes.

Tabla 5.8 Prueba de Holm para *dataset* de pequeñas dimensiones

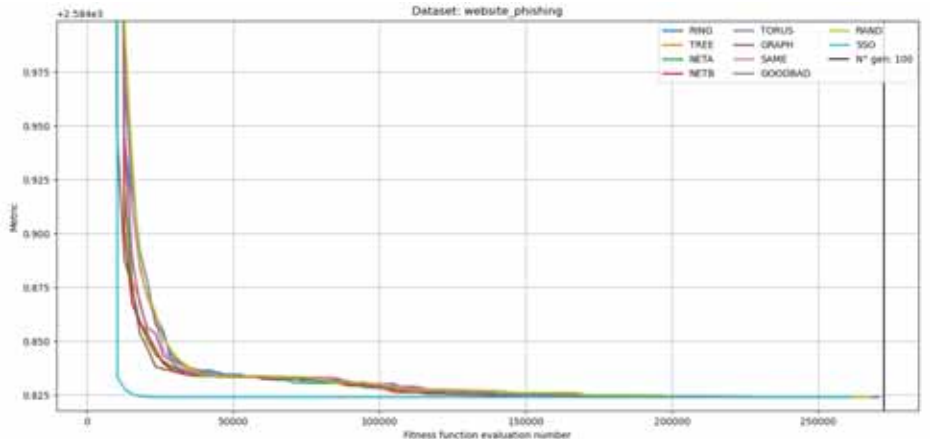
Dataset	Alg. de Control	i	Algoritmo	Rank	P-Valor	α/i
d1	$\mathcal{P}sso_{nb}$ (Rank: 4.28)	8	$\mathcal{P}sso_{ur}$	6.37	**1.36E-4**	0.00625
		7	$\mathcal{P}sso_{ra}$	5.72	0.0086	0.0071
		6	$\mathcal{P}sso_{na}$	5.60	0.0159	0.0083
		5	$\mathcal{P}sso_{to}$	4.83	0.32	0.01
		4	$\mathcal{P}sso_{tr}$	4.60	0.5591	0.0125
		3	$\mathcal{P}sso_{gb}$	4.57	0.5965	0.0167
		2	$\mathcal{P}sso_{cg}$	4.56	0.609	0.025
		1	$\mathcal{P}sso_{sa}$	4.47	0.73	0.05
d3	$\mathcal{P}sso_{nb}$ (Rank: 2.66)	8	$\mathcal{P}sso_{bg}$	6.07	**4.79E-10**	0.00625
		7	$\mathcal{P}sso_{ra}$	5.81	**8.87E-9**	0.0071
		6	$\mathcal{P}sso_{sa}$	5.80	**9.88E-9**	0.0083
		5	$\mathcal{P}sso_{ur}$	5.53	**1.61E-7**	0.01
		4	$\mathcal{P}sso_{to}$	5.36	**8.24E-7**	0.0125
		3	$\mathcal{P}sso_{na}$	5.22	**2.96E-6**	0.0167
		2	$\mathcal{P}sso_{cg}$	4.91	**3.99E-5**	0.025
		1	$\mathcal{P}sso_{tr}$	3.64	0.07	0.05
d4	$\mathcal{P}sso_{nb}$ (Rank: 2.41)	8	$\mathcal{P}sso_{ur}$	6.85	**5.22E-16**	0.00625
		7	$\mathcal{P}sso_{to}$	6.00	**5.591E-11**	0.0071
		6	$\mathcal{P}sso_{sa}$	5.93	**1.30E-10**	0.0083
		5	$\mathcal{P}sso_{gb}$	5.64	**3.70E-9**	0.01
		4	$\mathcal{P}sso_{ra}$	5.52	**1.36E-8**	0.0125
		3	$\mathcal{P}sso_{cg}$	5.30	**1.32E-7**	0.0167
		2	$\mathcal{P}sso_{tr}$	4.01	**0.003**	0.025
		1	$\mathcal{P}sso_{na}$	3.34	0.09	0.05
d5	$\mathcal{P}sso_{na}$ (Rank: 1.50)	8	$\mathcal{P}sso_{cg}$	8.22	**1.33E-34**	0.00625
		7	$\mathcal{P}sso_{to}$	7.96	**4.18E-32**	0.0071
		6	$\mathcal{P}sso_{ur}$	6.36	**7.11E-19**	0.0083
		5	$\mathcal{P}sso_{ra}$	5.70	**1.75E-14**	0.01
		4	$\mathcal{P}sso_{gb}$	5.56	**1.24E-13**	0.0125
		3	$\mathcal{P}sso_{tr}$	4.36	**1.77E-7**	0.0167
		2	$\mathcal{P}sso_{sa}$	3.68	**6.89E-5**	0.025
		1	$\mathcal{P}sso_{nb}$	1.66	0.77	0.05
d6	$\mathcal{P}sso_{nb}$ (Rank: 2.20)	8	$\mathcal{P}sso_{ur}$	6.62	**7.04E-16**	0.00625
		7	$\mathcal{P}sso_{cg}$	6.28	**9.40E-14**	0.0071
		6	$\mathcal{P}sso_{ra}$	5.79	**5.59E-11**	0.0083
		5	$\mathcal{P}sso_{to}$	5.77	**7.13E-11**	0.01
		4	$\mathcal{P}sso_{bg}$	5.05	**1.96E-7**	0.0125
		3	$\mathcal{P}sso_{tr}$	4.86	**1.19E-6**	0.0167
		2	$\mathcal{P}sso_{sa}$	4.85	**1.31E-6**	0.025
		1	$\mathcal{P}sso_{na}$	3.58	**0.01**	0.05

Tabla 5.9 Prueba de Holm para *dataset* de grandes dimensiones

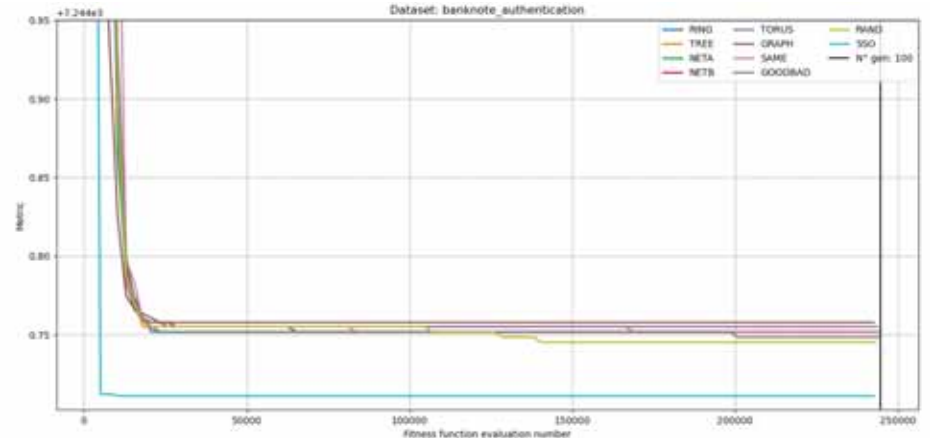
Dataset	**Alg. de Control**	**i**	**Algoritmo**	**Rank**	**P-Valor**	α/**i**
d7	$\mathcal{P}sso_{na}$ (Rank: 2.55)	8	$\mathcal{P}sso_{to}$	7.14	**5.29E-17**	0.00625
		7	$\mathcal{P}sso_{ur}$	6.81	**7.39E-15**	0.0071
		6	$\mathcal{P}sso_{cg}$	6.24	**1.62E-11**	0.0083
		5	$\mathcal{P}sso_{gb}$	6.19	**3.02E-11**	0.01
		4	$\mathcal{P}sso_{tr}$	4.97	**9.95E-6**	0.0125
		3	$\mathcal{P}sso_{ra}$	4.02	**0.0073**	0.0167
		2	$\mathcal{P}sso_{sa}$	3.98	**0.009**	0.025
		1	$\mathcal{P}sso_{nb}$	3.10	0.32	0.05
d8	$\mathcal{P}sso_{na}$ (Rank: 1.48)	8	$\mathcal{P}sso_{ur}$	8.34	**5.48E-36**	0.00625
		7	$\mathcal{P}sso_{ra}$	7.80	**8.42E-31**	0.0071
		6	$\mathcal{P}sso_{cg}$	6.96	**1.45E-23**	0.0083
		5	$\mathcal{P}sso_{to}$	6.90	**4.35E-23**	0.01
		4	$\mathcal{P}sso_{gb}$	4.78	**1.69E-9**	0.0125
		3	$\mathcal{P}sso_{tr}$	3.84	**1.64E-5**	0.0167
		2	$\mathcal{P}sso_{sa}$	3.08	**0.003**	0.025
		1	$\mathcal{P}sso_{nb}$	1.82	0.53	0.05
d9	$\mathcal{P}sso_{gb}$ (Rank: 2.67)	8	$\mathcal{P}sso_{cg}$	7.66	**8.20E-20**	0.00625
		7	$\mathcal{P}sso_{to}$	7.32	**2.07E-17**	0.0071
		6	$\mathcal{P}sso_{ra}$	6.24	**7.13E-11**	0.0083
		5	$\mathcal{P}sso_{ur}$	5.20	**3.85E-6**	0.01
		4	$\mathcal{P}sso_{tr}$	4.54	**6.40E-4**	0.0125
		3	$\mathcal{P}sso_{nb}$	4.41	**0.0015**	0.0167
		2	$\mathcal{P}sso_{na}$	4.05	**0.012**	0.025
		1	$\mathcal{P}sso_{sa}$	2.91	0.66	0.05



(a) Convergencia de los algoritmos SSO y P-SSO para *dataset* d1

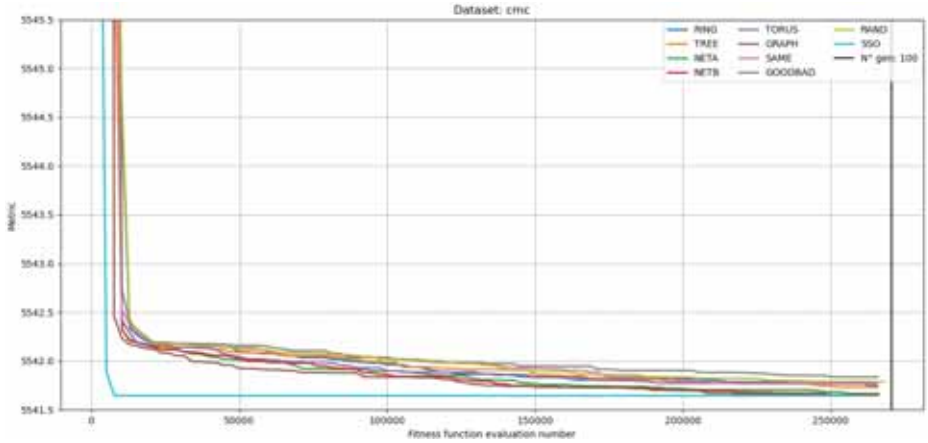


(b) Convergencia de los algoritmos SSO y P-SSO para *dataset* d2

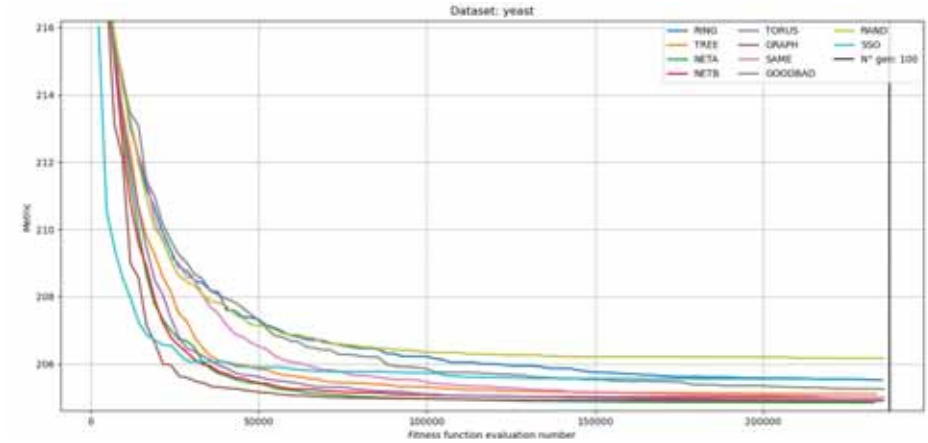


(c) Convergencia de los algoritmos SSO y P-SSO para *dataset* d3

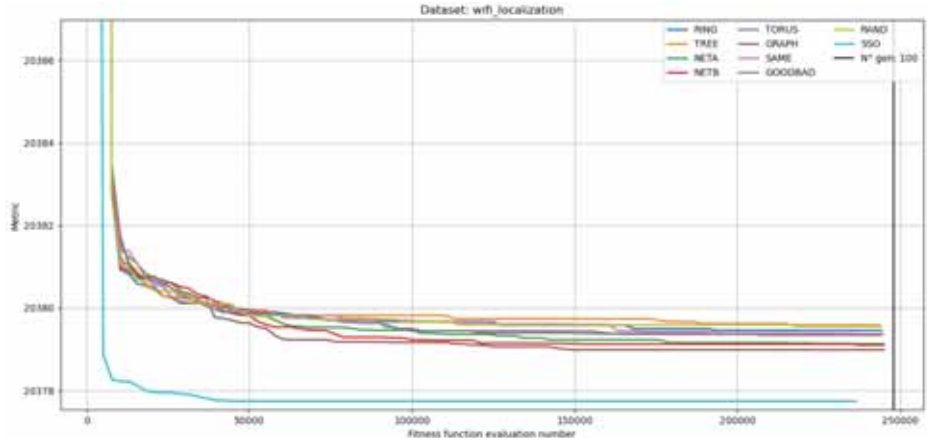
Fig. 5.1 Convergencia de los algoritmos SSO y P-SSO para *dataset* de pequeñas dimensiones (1)



(a) Convergencia de los Algoritmos SSO y P-SSO para *dataset* d4

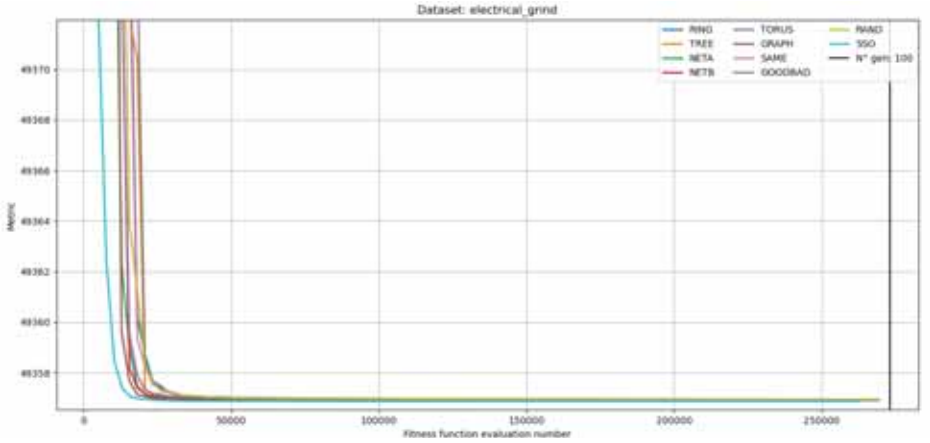


(b) Convergencia de los algoritmos SSO y P-SSO para *dataset* d5

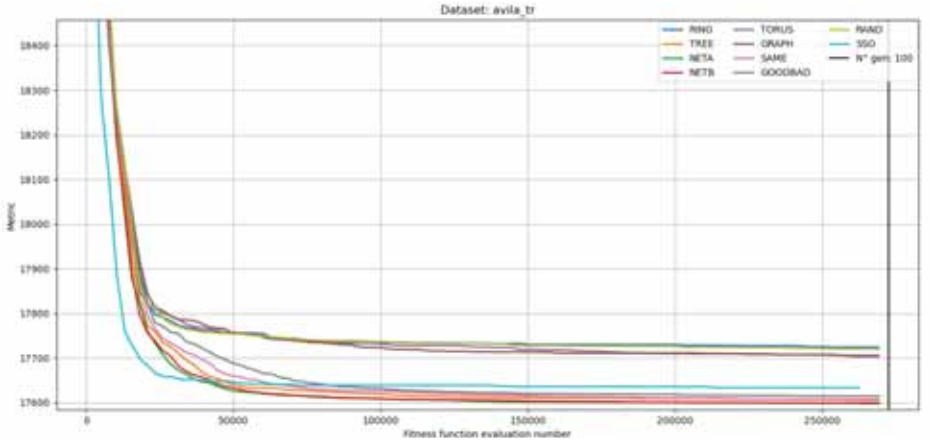


(c) Convergencia de los algoritmos SSO y P-SSO para *dataset* d6

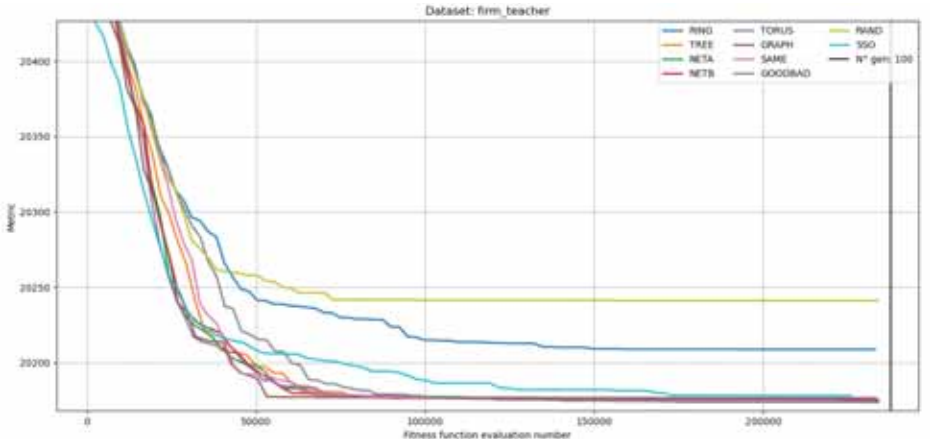
Fig. 5.2 Convergencia de los algoritmos SSO y P-SSO para *dataset* de pequeñas dimensiones (2)



(a) Convergencia de los algoritmos SSO y P-SSO para *dataset* d7



(b) Convergencia de los algoritmos SSO y P-SSO para *dataset* d8



(c) Convergencia de los algoritmos SSO y P-SSO para *dataset* d9

Fig. 5.3 Convergencia de los algoritmos SSO y P-SSO para *dataset* de grandes dimensiones

Conclusión

Los modelos de isla son una buena alternativa para paralelizar algoritmos genéticos y algoritmos bioinspirados, los parámetros de migración ayudan a tener un intercambio ordenado de individuos entre las islas que mantienen una comunicación. En consecuencia se implementó la versión paralela del algoritmo SSO, utilizando la interfaz MPI, para lo cual se aplicó topologías de comunicación estáticas (anillo unidireccional, árbol, red-A, red-B, toroide, grafo completo) y dinámicas (comunicación, entre similares, entre buenos y malos, entre aleatorios). Luego de llevar a cabo los experimentos se aplicaron pruebas estadísticas para validar los resultados obtenidos, tanto para la métrica y para el tiempo de ejecución.

- En este trabajo de investigación se determinó que el algoritmo SSO muestra mejores métricas en 70% de los resultados. Sin embargo, estos resultados son muy similares a las métricas que genera el algoritmo P-SSO, con una diferencia promedio del 0.15% en el peor de los casos y de 0,0000034% en el mejor de los casos.

- En este trabajo de investigación se determinó que los modelos paralelos del algoritmo P-SSO en promedio son 15 veces más rápidas que el algoritmo SSO, para el análisis de *dataset* de grandes dimensiones.

- En este trabajo de investigación se determinó que los modelos paralelos del algoritmo P-SSO en promedio son 28 veces más rápidas que el algoritmo SSO, para el análisis de *dataset* de pequeñas dimensiones.

- En este trabajo de investigación se determinó que los modelos paralelos red-A ($\mathcal{P}sso_{na}$) y red-B ($\mathcal{P}sso_{nb}$), ambos con topologías estáticas de red 6×4, generan mejores métricas que los otros modelos paralelos estudiados en este trabajo.

- Finalmente, en este trabajo de investigación se determinó que el modelo paralelo con topología estática anillo unidireccional ($\mathcal{P}sso_{ur}$), tiene mejor tiempo de ejecución que los demás modelos paralelos estudiados en este trabajo.

Recomendación

Para seguir nutriendo y aportando a la línea de investigación de este trabajo, se recomienda seguir investigando a cerca de los mecanismos de modelos de islas para paralelizar algoritmos bioinspirados. También, se sugiere utilizar sistemas de memoria compartida y arquitecturas como GPU o GPGPU para paralelizar estos algoritmos. Finalmente, se recomienda desarrollar los siguientes trabajos futuros.

- Diseñar e implementar el algoritmo P-SSO utilizando el modelo de islas con las topologías estáticas: hipercubo, mariposa y piramidal para 24 nodos y realizar comparaciones de la métrica y el tiempo de ejecución con los modelos paralelos estudiados en este trabajo.
- Diseñar e implementar una versión híbrida compuesta por los algoritmos SSO y P-SSO, donde el algoritmo SSO se utilice para generar la población inicial.
- Modificar los modelos paralelos del algoritmo P-SSO estudiados en este trabajo, para que se ejecuten con un número mayor de procesadores.

Referencias

[1] Andalon-Garcia, I. R. and Chavoya, A. (2012). Performance comparison of three topologies of the island model of a parallel genetic algorithm implementation on a cluster platform. In *CONIELECOMP 2012, 22nd International Conference on Electrical Communications and Computers*, pages 1–6. IEEE.

[2] Arias Gonzales, J. and Covinos, M. (2021). *DISEÑO Y METODOLOGÍA DE LA INVESTIGACIÓN*. ResearchGate.

[3] Cantú-Paz, E. (1998). A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171.

[4] Chandran, T. R., Reddy, A., and Janet, B. (2016). A social spider optimization approach for clustering text documents. In *2016 2nd International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB)*, pages 22–26. IEEE.

[5] Cuevas, E., Cienfuegos, M., ZaldíVar, D., and Pérez-Cisneros, M. (2013). A swarm optimization algorithm inspired in the behavior of the social-spider. *Expert Systems with Applications*, 40(16):6374–6384.

[6] da Silveira, L. Â., Soncco-Álvarez, J. L., and Ayala-Rincón, M. (2017). Parallel genetic algorithms with sharing of individuals for sorting unsigned genomes by reversals. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 741–748. IEEE.

[7] da Silveira, L. A., Soncco-Alvarez, J. L., de Barros, J. B., Llanos, C. H., and Ayala-Rincón, M. (2019a). On the behavior of parallel island models. *Univ. de Brasılia, Tech. Rep*.

[8] da Silveira, L. A., Soncco-Alvarez, J. L., de Lima, T. A., and Ayala-Rincon, M. (2018). Parallel multi-island genetic algotirth for sorting unsigned genomes by reversals. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE.

[9] da Silveira, L. A., Soncco-Alvarez, J. L., de Lima, T. A., and Ayala-Rincón, M. (2019b). Behavior of bioinspired algorithms in parallel island models. -.

[10] da Silveira, L. A., Soncco-Álvarez, J. L., de Lima, T. A., and Ayala-Rincón, M. (2019c). Parallel island model genetic algorithms applied in np-hard problems. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 3262–3269. IEEE.

[11] Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research*, 7:1–30.

[12] Dua, D. and Graff, C. (2017). UCI machine learning repository.

[13] Duarte, G., Lemonge, A., and Goliatt, L. (2017). A dynamic migration policy to the island model. In *2017 IEEE Congress on evolutionary computation (CEC)*, pages 1135–1142. IEEE.

[14] ElSoud, M. A. and Anter, A. M. (2016). Computational intelligence optimization algorithm based on meta-heuristic social-spider: case study on ct liver tumor diagnosis. *Comput Intell*, 7(4):466–475.

[15] Flynn, M. J. (1966). Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909.

[16] Gebali, F. (2011). *Algorithms and parallel computing*, volume 84. John Wiley & Sons.

[17] Grama, A., Kumar, V., Gupta, A., and Karypis, G. (2003). *Introduction to parallel computing*. Pearson Education.

[18] Han, J., Pei, J., and Kamber, M. (2011). *Data mining: concepts and techniques*. Elsevier.

[19] Jain, A. K., Murty, M. N., and Flynn, P. J. (1999). Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323.

[20] León, J., Chullo-Llave, B., Enciso-Rodas, L., and Soncco-Álvarez, J. L. (2020). A multi-objective optimization algorithm for center-based clustering. *Electronic Notes in Theoretical Computer Science*, 349:49 – 67. Proceedings of CLEI 19, the XLV Latin American Computing Conference.

[21] Lynn, N., Ali, M. Z., and Suganthan, P. N. (2018). Population topologies for particle swarm optimization and differential evolution. *Swarm and evolutionary computation*, 39:24–35.

[22] Maulik, U. and Bandyopadhyay, S. (2000). Genetic algorithm-based clustering technique. *Pattern Recognition*, 33(9):1455 – 1465.

[23] Message Passing Interface Forum, V. . (2012). *MPI: A Message-Passing Interface Standard*. High-Performance Computing Center Stuttgart University of Stuttgart Nobelstr. 19 D-70550 Stuttgart Germany, 3 edition.

[24] Nielsen, F. (2016). Introduction to mpi: The message passing interface. In *Introduction to HPC with MPI for Data Science*, pages 21–62. Springer.

[25] Pacheco, P. S. (2011). *An introduction to parallel programming*. ELSEVIER, 1 edition.

[26] Rauber, T. and Rünger, G. (2013). Parallel programming: For multicore and cluster systems.[sl]: Springer science & business media, 2013. *Citado na*, page 30.

[27] Shukla, U. P. and Nanda, S. J. (2016). Parallel social spider clustering algorithm for high dimensional datasets. *Engineering Applications of Artificial Intelligence*, 56:75–90.

[28] Sudholt, D. (2015). Parallel evolutionary algorithms. In *Springer Handbook of Computational Intelligence*, pages 929–959. Springer.

[29] Thalamala, R. C., Reddy, A. V. S., and Janet, B. (2018). A novel bio-inspired algorithm based on social spiders for improving performance and efficiency of data clustering. *Journal of Intelligent Systems*, 29(1):311–326.

[30] Trobec, R., Slivnik, B., Bulić, P., and Robič, B. (2018). *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*. Springer.

[31] Vera-Olivera, H., Soncco-Alvarez, J. L., and Enciso-Rodas, L. (2016). Social spider algorithm approach for clustering. In *Proceedings of the 3rd Annual International Symposium on Information Management and Big Data-SIMBig 2016, Cusco, Peru*, pages 114–121.

[32] Xu, R. and Wunsch, D. (2008). *Clustering*, volume 10. John Wiley & Sons.

[33] Zhou, Y., Zhou, Y., Luo, Q., and Abdel-Basset, M. (2017). A simplex method-based social spider optimization algorithm for clustering analysis. *Engineering Applications of Artificial Intelligence*, 64:67–82.

Anexo A

Matriz de consistencia

En la Tabla A.1 se presenta una visión panorámica de los elementos básicos del proyecto de investigación. Estos elementos describen de manera resumida, al problema, objetivos (general y específicos), hipótesis, variables (dependiente e independiente) y la metodología aplicada para abordar el proyecto.

Tabla A.1 Matriz de consistencia

Título: Paralelización del Algoritmo Basado en el Comportamiento Social de las Arañas para *Clustering*.				
Problema	**Objetivos**	**Hipótesis**	**Variables e Indicadores**	**Metodología**
Problema general: ¿En que porcentaje la implementación del algoritmo SSO paralelo va a mejorar la precisión de la métrica y el tiempo de ejecución del algoritmo SSO secuencial, para resolver problemas de *clustering*? **Problemas específicos:** ¿Cuál es el valor de la métrica que genera el algoritmo SSO en su versión secuencial? ¿Cuál es el tiempo de ejecución del algoritmo SSO en su versión secuencial? ¿Cuáles son los mejores parámetros para ejecutar el algoritmo SSO en su versión paralela? ¿Cuál es el valor de la métrica que genera el algoritmo SSO en su versión paralela? ¿Cuál es el tiempo de ejecución del algoritmo SSO en su versión paralela? ¿Cuál es la diferencia entre las métricas que generan los algoritmos SSO secuencial y paralelo? ¿Cuál es el *speedup* entre los algoritmos SSO secuencial y paralelo?	**Objetivo general:** Determinar en que porcentaje el algoritmo SSO paralelo para resolver problemas de *clustering*, va a mejorar la precisión de la métrica y el tiempo de ejecución con respecto a la versión secuencial del algoritmo SSO. **Objetivos específicos:** –Medir la métrica que genera el algoritmo SSO. –Medir el tiempo de ejecución del algoritmo SSO. –Determinar los parámetros mas adecuados para el algoritmo P-SSO. –Medir la métrica que genera el algoritmo P-SSO. –Medir el tiempo de ejecución del algoritmo P-SSO. –Comparar y validar los resultados de la métrica entre los algoritmos SSO y P-SSO, utilizando métodos estadísticos. –Evaluar el *speedup* resultante entre los algoritmos SSO y P-SSO, utilizando métodos estadísticos.	**Hipótesis general:** El algoritmo *Social Spider Optimizatión* paralelo (P-SSO) para resolver problemas de *clustering*, mejora la precisión de la métrica y el tiempo de ejecución. **Hipótesis específicos:** –Los modelos paralelos del algoritmo P-SSO proporcionan mejor precisión de la métrica y tiempo de ejecución que el algoritmo SSO, para resolver problemas de *clustering* en *datasets* de grandes dimensiones. –Los modelos paralelos del algoritmo P-SSO proporcionan mejor precisión de la métrica y tiempo de ejecución que el algoritmo SSO, para resolver problemas de *clustering* en *datasets* de dimensiones pequeñas. –En promedio los modelos paralelos del algoritmo P-SSO se ejecuta 2 veces más rápido que el algoritmo SSO, para un mismo conjunto de datos. –El modelo paralelo con topología estática grafo completo, genera mejores resultados en cuanto a la precisión de la métrica.	**Variables:** –Var. independiente: Tamaño del *dataset*. –Var. dependiente: Valor de la métrica. –Var. dependiente: Tiempo de ejecución.	**Tipo de investigación:** Según su finalidad es una investigación aplicada. **Nivel de la investigación:** Según su alcance es una investigación explicativa. **Diseño de experimento:** Pre experimento.

Anexo B

Código fuente

El código fuente fue escrito en el lenguaje de programación C. Esta sección esta dividido en los siguientes módulos.

- Módulo: Algoritmo secuencial *Social Spider Optimization* (SSO)
- Módulos: Algoritmo paralelo *Social Spider Optimization* (P-SSO)

B.1 Algoritmo secuencial *Social Spider Optimization*

- main_sso_exp.c: Este módulo contiene el algoritmo de optimización basado en el comportamiento social de arañas.
- utils.c: Este módulo contiene las operaciones básicas para el manejo de memoria, archivos.
- clusterCenter.c: Este módulo contiene la definición de un centro de un determinado *cluster*.
- spider.c: Este módulo contiene la definición de un cromosoma.
- population.c: Este módulo contiene métodos que modifican la población del algoritmo SSO, tales como el operador de apareamiento.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "clusterCenter.h"
#include "spider.h"
#include "population.h"
#include "utils.h"

int main(int argc, char *argv[]){
    Population population;
    AuxPopulation auxPopulation;
    AuxSpider auxSpider;
    int *columns = NULL;
    int column = 0;
    double **dataset = NULL;
    double **population_generated = NULL;

    char *fileNameColumns = argv[1];
    int numberClusters = atoi(argv[2]);
    int row = atoi(argv[3]);
    char *fileNameRows = argv[4];
    char *fileNamePopulation = argv[5];
    int limitCallFitness = atoi(argv[6]);
    int seed = atoi(argv[7]);
    char *fileNameOutput = argv[8];

    int numberGenerations = 100 * 1000000;
    int populationSize = 100 * 24;

    clock_t t1 = clock();

    allocateMemoryReadColumn(&columns, &column, fileNameColumns);
    allocateMemoryReadDataset(&dataset, row, column, columns, fileNameRows);
    allocateMemoryReadPopulation(&population_generated, populationSize,
```

```
36      column * numberClusters, fileNamePopulation);
37
38      generateSeed(seed);
39      readInitialPopulation(&population, populationSize, numberClusters,
40                            dataset, row, column, 0, &auxPopulation,
41                            &auxSpider,
42      population_generated, 0, limitCallFitness);
43      calculateWeightPopulation(&population, dataset, row, column, &auxSpider);
44      // showPopulation(&population);
45
46      char fileName[256];
47      sprintf(fileName, "%s", fileNameOutput);
48      FILE *fp = fopen(fileName, "a");
49
50      float best_metric = 0;
51      int count_fitness = 0;
52
53      for(int k = 2; k <= numberGenerations; k++) {
54          // get metric and eval fitness value for generation (convergence)
55          /* best_metric = population.spiders[population.indexBest]->fitness;
56          count_fitness = population.countFitness;
57          printf("[%d,%f,%d]\n", k - 1, best_metric, count_fitness);
58          fprintf(fp, "%d,%f,%d\n", k - 1, best_metric, count_fitness); */
59
60          femaleCooperativeOperator(&population, &auxPopulation);
61          maleCooperativeOperator(&population, dataset, row, column,
62                                  &auxPopulation);
63          matingOperator(&population, dataset, row, column, &auxPopulation,
64                         &auxSpider);
65          replacement(&population, &auxPopulation);
66          calculateWeightPopulation(&population, dataset, row, column,
67                                    &auxSpider);
68
69          // showPopulation(&population);
70          // showBestFitness(&population, k);
71          // showBestSpider(&population);
```

```
72
73          count_fitness = population.countFitness;
74          if (count_fitness >= limitCallFitness) {
75              // printf("End core %d: %d\n", 0, count_fitness);
76              break;
77          }
78      }
79
80      // showAllResults(&population, numberGenerations, dataset, row, column);
81      // showMetricBestSpider(&population);
82
83      best_metric = population.spiders[population.indexBest]->fitness;
84      clock_t t2 = clock();
85
86      printf("[%f,%f]\n", best_metric, ((double)(t2 - t1)) / CLOCKS_PER_SEC);
87      fprintf(fp, "%f,%f\n", best_metric, ((double)(t2 - t1)) / CLOCKS_PER_SEC);
88
89      /* char fileName[256];
90      sprintf(fileName, "%s", "output/aux_sso.out");
91      FILE *fp = fopen(fileName, "a");
92      fprintf(fp, "%f,%f\n", best_metric, ((double)(t2 - t1)) / CLOCKS_PER_SEC);
93      fclose(fp); */
94      fclose(fp);
95
96      freeMemoryReadColumn(columns);
97      freeMemoryReadDataset(dataset, row);
98      freeMemoryReadDataset(population_generated, populationSize);
99      freeMemoryPopulation(&population, &auxPopulation, &auxSpider);
100
101     return 0;
102 }
```

```
1  #include "utils.h"
2
3  void generateSeed(int seed){
4      // time_t t;
```

```
5       // unsigned int seed = time(&t);
6       // unsigned int seed = 1536705665 + world_rank;
7       srand(seed);
8       // printf("Seed: %d\n", seed);
9   }
10
11  int randomInt(int n){
12      return rand() % n;
13  }
14
15  double random_(){
16      return rand() / ((double) RAND_MAX);
17  }
18
19  void allocateMemoryArrayInt(int **array, int length){
20      *array = (int*)malloc(sizeof(int) * length);
21      if (*array == NULL){
22          printf("No have memory.\n");
23          exit(1);
24      }
25      memset(*array, 0, sizeof(int) * length);
26      /* for (int i = 0; i < length; i++){
27          (*array)[i] = 0;
28      } */
29  }
30
31  void freeMemoryArrayInt(int *array){
32      free(array);
33  }
34
35  void allocateMemoryArrayDouble(double **array, int length){
36      *array = (double*)malloc(sizeof(double) * length);
37      if (*array == NULL){
38          printf("No have memory.\n");
39          exit(1);
40      }
```

```
41      memset(*array, 0, sizeof(double) * length);
42      /* for (int i = 0; i < length; i++){
43          (*array)[i] = 0;
44      } */
45  }
46
47  void freeMemoryArrayDouble(double *array){
48      if (array != NULL) {
49          free(array);
50      }
51  }
52
53  void allocateMemoryMatrixDouble(double ***matrix, int row, int column){
54      *matrix = (double**)malloc(sizeof(double*) * row);
55      if (*matrix == NULL){
56          printf("No have memory.\n");
57          exit(1);
58      }
59      for (int i = 0; i < row; i++){
60          allocateMemoryArrayDouble(&(*matrix)[i], column);
61      }
62  }
63
64  void freeMemoryMatrixDouble(double **matrix, int row){
65      for (int i = 0; i < row; i++){
66          freeMemoryArrayDouble(matrix[i]);
67      }
68      free(matrix);
69  }
70
71  void allocateMemoryReadColumn(int **array, int *column, char *fileName){
72      FILE *cfPtr;
73      int data;
74      int k = 0;
75      *array = (int*)malloc(sizeof(int) * 255);
76
```

```
77      if ((cfPtr = fopen(fileName, "r")) == NULL) {
78          printf("File could not be opened\n");
79          exit(1);
80      } else {
81          while(fscanf(cfPtr, "%d", &data) != EOF ) {
82              (*array)[k] = data;
83              k++;
84          }
85          /* while (!feof(cfPtr)) {
86              fscanf(cfPtr, "%d", &data);
87              (*array)[k] = data;
88              k++;
89          } */
90          fclose( cfPtr );
91      }
92      *column = k;
93  }
94
95  void freeMemoryReadColumn(int *array){
96      free(array);
97  }
98
99  void allocateMemoryReadDataset_(double ***matrix, int row, int column,
100                                  int *columns){
101     char item_read[255];
102     char *item;
103     int j = 0;
104     int k = 0;
105     *matrix = (double**)malloc(sizeof(double*) * row);
106     if (*matrix == NULL){
107         printf("No have memory.\n");
108         exit(1);
109     }
110     for (int i = 0; i < row; i++){
111         (*matrix)[i] = (double*)malloc(sizeof(double) * column);
112         scanf("%s", item_read);
```

```
113         item = strtok(item_read, ",");
114         j = 0;
115         k = 0;
116         while (item != NULL) {
117             if(k == columns[j]){
118                 (*matrix)[i][j] = atof(item);
119                 j++;
120             }
121             k++;
122             if(j == column){
123                 break;
124             }
125             item = strtok(NULL, ",");
126         }
127     }
128 }
129
130 void allocateMemoryReadDataset(double ***matrix, int row, int column,
131                                int *columns, char *fileName){
132     FILE *cfPtr;
133     char item_read[1024];
134     char *item;
135     int j = 0;
136     int k = 0;
137     // printf("Read dataset\n");
138     *matrix = (double**)malloc(sizeof(double*) * row);
139     if ((cfPtr = fopen(fileName, "r")) == NULL) {
140         printf("File could not be opened\n");
141         exit(1);
142     } else {
143         if (*matrix == NULL){
144             printf("No have memory.\n");
145             exit(1);
146         }
147         for (int i = 0; i < row; i++){
148             (*matrix)[i] = (double*)malloc(sizeof(double) * column);
```

```
149             fscanf(cfPtr, "%s", item_read);
150             item = strtok(item_read, ",");
151             j = 0;
152             k = 0;
153             while (item != NULL) {
154                 if(k == columns[j]){
155                     (*matrix)[i][j] = atof(item);
156                     // printf("%f ", (*matrix)[i][j]);
157                     j++;
158                 }
159                 k++;
160                 if(j == column){
161                     break;
162                 }
163                 item = strtok(NULL, ",");
164             }
165             // printf("\n");
166         }
167         fclose(cfPtr);
168     }
169 }
170 
171 void freeMemoryReadDataset(double **matrix, int row){
172     for (int i = 0; i < row; i++){
173         freeMemoryArrayDouble(matrix[i]);
174     }
175     free(matrix);
176 }
177 
178 void allocateMemoryReadPopulation(double ***matrix, int row, int column,
179                                   char *fileName){
180     FILE *cfPtr;
181     char item_read[2048];
182     char *item;
183     int j;
184     // printf("Read population\n");
```

```
185     *matrix = (double**)malloc(sizeof(double*) * row);
186     if ((cfPtr = fopen(fileName, "r")) == NULL) {
187         printf("File could not be opened\n");
188         exit(1);
189     } else {
190         if (*matrix == NULL){
191             printf("No have memory.\n");
192             exit(1);
193         }
194         for (int i = 0; i < row; i++){
195             (*matrix)[i] = (double*)malloc(sizeof(double) * column);
196             fscanf(cfPtr, "%s", item_read);
197             item = strtok(item_read, ",");
198             j = 0;
199             while (item != NULL) {
200                 (*matrix)[i][j] = atof(item);
201                 // printf("%f ", (*matrix)[i][j]);
202                 item = strtok(NULL, ",");
203                 j++;
204 
205             }
206             // printf("\n");
207         }
208         fclose(cfPtr);
209     }
210 }
211 
212 void freeMemoryReadPopulation(double **matrix, int row){
213     for (int i = 0; i < row; i++){
214         freeMemoryArrayDouble(matrix[i]);
215     }
216     free(matrix);
217 }
218 
219 void allocateMemoryDictionary(DictionaryPtr *dictionaryPtr, int key){
220     *dictionaryPtr =  (Dictionary*)malloc(sizeof(Dictionary));
```

```
221     if (*dictionaryPtr == NULL){
222         printf("No have memory.\n");
223         exit(1);
224     }
225 
226     (*dictionaryPtr)->key = key;
227     (*dictionaryPtr)->value = 0;
228 }
229 
230 void freeMemoryDictionary(DictionaryPtr dictionaryPtr){
231     free(dictionaryPtr);
232 }
233 
234 void allocateMemoryArrayDictionary(DictionaryPtr **dictionaryPtr,
235                                    int lengthDictionary){
236     *dictionaryPtr = (DictionaryPtr*)malloc(sizeof(Dictionary) *
237                         lengthDictionary);
238     if (*dictionaryPtr == NULL){
239         printf("No have memory.\n");
240         exit(1);
241     }
242     for (int i = 0; i < lengthDictionary; i++){
243         allocateMemoryDictionary(&(*dictionaryPtr)[i], i);
244     }
245 }
246 
247 void freeMemoryArrayDictionary(DictionaryPtr *dictionaryPtr,
248                                int lengthDictionary){
249     for (int i = 0; i < lengthDictionary; i++){
250         freeMemoryDictionary(dictionaryPtr[i]);
251     }
252     free(dictionaryPtr);
253 }
254 
255 void readPolicy(PolicyPtr policyPtr, char *fileName){
256     FILE *cfPtr;
```

```
257     char item_read[255];
258     char *item;
259
260     if ((cfPtr = fopen(fileName, "r")) == NULL) {
261         printf("File could not be opened\n");
262         exit(1);
263     } else {
264         fscanf(cfPtr, "%s", item_read);
265         item = strtok(item_read, ":");
266         item = strtok(NULL, ":");
267         if(strcmp(item, "RING") == 0){
268             policyPtr->topology = RING;
269         }else{
270             if(strcmp(item, "TREE") == 0){
271                 policyPtr->topology = TREE;
272             }else{
273                 if(strcmp(item, "NETA") == 0){
274                     policyPtr->topology = NETA;
275                 } else{
276                     if(strcmp(item, "NETB") == 0){
277                         policyPtr->topology = NETB;
278                     } else{
279                         if(strcmp(item, "TORUS") == 0){
280                             policyPtr->topology = TORUS;
281                         } else {
282                             if(strcmp(item, "GRAPH") == 0){
283                                 policyPtr->topology = GRAPH;
284                             } else {
285                                 if(strcmp(item, "SAME") == 0){
286                                     policyPtr->topology = SAME;
287                                 } else {
288                                     if(strcmp(item, "GOODBAD") == 0){
289                                         policyPtr->topology = GOODBAD;
290                                     } else {
291                                         if(strcmp(item, "RAND") == 0){
292                                             policyPtr->topology = RAND;
```

```
293                                     } else {
294                                         printf("Topology could not found\n");
295                                         exit(1);
296                                     }
297                                 }
298                             }
299                         }
300                     }
301                 }
302             }
303         }
304     }
305
306     fscanf(cfPtr, "%s", item_read);
307     item = strtok(item_read, ":");
308     item = strtok(NULL, ":");
309     if(strcmp(item, "REMOVE") == 0){
310         policyPtr->emigration = REMOVE;
311     }else{
312         policyPtr->emigration = CLONE;
313     }
314
315     fscanf(cfPtr, "%s", item_read);
316     item = strtok(item_read, ":");
317     item = strtok(NULL, ":");
318     if(strcmp(item, "BEST") == 0){
319         policyPtr->choiceEmi = BEST;
320     }else{
321         if(strcmp(item, "WORST") == 0){
322             policyPtr->choiceEmi = WORST;
323         }else{
324             policyPtr->choiceEmi = RANDOM;
325         }
326     }
327
328     fscanf(cfPtr, "%s", item_read);
```

```
329         item = strtok(item_read, ":");
330         item = strtok(NULL, ":");
331         if(strcmp(item, "BEST") == 0){
332             policyPtr->choiceImm = BEST;
333         }else{
334             if(strcmp(item, "WORST") == 0){
335                 policyPtr->choiceImm = WORST;
336             }else{
337                 policyPtr->choiceImm = RANDOM;
338             }
339         }
340 
341         fscanf(cfPtr, "%s", item_read);
342         item = strtok(item_read, ":");
343         item = strtok(NULL, ":");
344         policyPtr->numberEmiImm = atoi(item);
345 
346         fscanf(cfPtr, "%s", item_read);
347         item = strtok(item_read, ":");
348         item = strtok(NULL, ":");
349         policyPtr->intervalEmiImm = atoi(item);
350 
351         fclose(cfPtr);
352     }
353 }
354 
355 void showArrayDoubleSend(PopulationPtr populationPtr, PolicyPtr policyPtr,
356                          double *array, int source) {
357     printf("Process %d send:\n", source);
358     for (int k = 0; k < policyPtr->numberEmiImm; k++){
359         printf("%d: [", k);
360         for(int i = 0; i < populationPtr->numClusters; i++){
361             printf("(");
362             for(int j = 0; j < populationPtr->pointDimension; j++){
363                 printf("%f, ", array[(k * populationPtr->numClusters + i) *
364                         populationPtr->pointDimension + j]);
```

```
365             }
366             printf("), ");
367         }
368         printf("]\n");
369     }
370 }
371
372 void showArrayDoubleReceived(PopulationPtr populationPtr, PolicyPtr policyPtr,
373                              double *array, int destination, int source,
374                              int orderNode) {
375     printf("Process %d received from process %d:\n", destination, source);
376     for (int k = 0; k < policyPtr->numberEmiImm; k++) {
377         printf("%d: [", (orderNode - 1) * policyPtr->numberEmiImm + k);
378         for(int i = 0; i < populationPtr->numClusters; i++){
379             printf("(");
380             for(int j = 0; j < populationPtr->pointDimension; j++){
381                 printf("%f, ", array[(k * populationPtr->numClusters + i) *
382                         populationPtr->pointDimension + j]);
383             }
384             printf("), ");
385         }
386         printf("]\n");
387     }
388 }
389
390 void showPolicyMigration(PolicyPtr policyPtr) {
391     char topology[7][10] = {"RING", "TREE", "NETA", "NETB", "TORUS",
392                             "GRAPH", "DINAMIC"};
393     char emigration[2][10] = {"CLONE", "REMOVE"};
394     char choiceEmiImm[3][10] = {"BEST", "WORST", "RANDOM"};
395     printf("Topology:       \t%s\n", topology[policyPtr->topology]);
396     printf("Emigration:     \t%s\n", emigration[policyPtr->emigration]);
397     printf("Choice Em:      \t%s\n", choiceEmiImm[policyPtr->choiceEmi]);
398     printf("Choice Im:      \t%s\n", choiceEmiImm[policyPtr->choiceImm]);
399     printf("Number Em/Im:   \t%d\n", policyPtr->numberEmiImm);
400     printf("Interval Em/Im: \t%d\n\n", policyPtr->intervalEmiImm);
```

```
401 }
402
403 void insertSort(DictionaryPtr *dictionaryPtr, int lengthDictionary) {
404     int index;
405     int key;
406     double value;
407     for (int i = 1; i < lengthDictionary; i++) {
408         key = dictionaryPtr[i]->key;
409         value = dictionaryPtr[i]->value;
410         index = i - 1;
411         while (index >= 0 && dictionaryPtr[index]->value > value) {
412             dictionaryPtr[index + 1]->key = dictionaryPtr[index]->key;
413             dictionaryPtr[index + 1]->value = dictionaryPtr[index]->value;
414             index--;
415         }
416         dictionaryPtr[index + 1]->key = key;
417         dictionaryPtr[index + 1]->value = value;
418     }
419 }
420
421 void bubbleSort(PopulationPtr populationPtr) {
422     DictionaryPtr temp;
423     for (int i = 1; i < populationPtr->lengthSpider; i++){
424         for (int j = 0; j < populationPtr->lengthSpider - i; j++) {
425             if (populationPtr->fitness[j]->value >
426                 populationPtr->fitness[j + 1]->value) {
427                 temp = populationPtr->fitness[j];
428                 populationPtr->fitness[j] = populationPtr->fitness[j + 1];
429                 populationPtr->fitness[j + 1] = temp;
430             }
431         }
432     }
433 }
434
435 double average(PopulationPtr populationPtr) {
436     double sum = 0;
```

```
437     double value = 0;
438     for (int i = 0; i < populationPtr->lengthSpider; i++) {
439         value = populationPtr->fitness[i]->value;
440         sum += value;
441     }
442     return sum / populationPtr->lengthSpider;
443 }
444 
445 double standardDeviation(PopulationPtr populationPtr) {
446     double sum = 0;
447     double value = 0;
448     double average_ = average(populationPtr);
449     for (int i = 0; i < populationPtr->lengthSpider; i++) {
450         value = populationPtr->fitness[i]->value;
451         sum += pow(value - average_, 2);
452     }
453     return sqrt(sum / populationPtr->lengthSpider);
454 }
```

```
1  #include "clusterCenter.h"
2  
3  void allocateMemoryClusterCenter(ClusterCenterPtr *clusterCenterPtr,
4                                   int lengthPoint){
5      *clusterCenterPtr =  (ClusterCenter*)malloc(sizeof(ClusterCenter));
6      if (*clusterCenterPtr == NULL){
7          printf("No have memory.\n");
8          exit(1);
9      }
10     allocateMemoryArrayDouble(&(*clusterCenterPtr)->point, lengthPoint);
11     (*clusterCenterPtr)->length = lengthPoint;
12 }
13 
14 void freeMemoryClusterCenter(ClusterCenterPtr clusterCenterPtr){
15     freeMemoryArrayDouble(clusterCenterPtr->point);
16     free(clusterCenterPtr);
17 }
```

```
18
19 void allocateMemoryArrayClusterCenter(ClusterCenterPtr **clusterCenterPtr,
20                                         int lengthCluster, int lengthPoint){
21     *clusterCenterPtr = (ClusterCenterPtr*)malloc(sizeof(ClusterCenter) *
22                         lengthCluster);
23     if (*clusterCenterPtr == NULL){
24         printf("No have memory.\n");
25         exit(1);
26     }
27     for (int i = 0; i < lengthCluster; i++){
28         allocateMemoryClusterCenter(&(*clusterCenterPtr)[i], lengthPoint);
29     }
30 }
31
32 void freeMemoryArrayClusterCenter(ClusterCenterPtr *clusterCenterPtr,
33                                   int lengthCluster){
34     for (int i = 0; i < lengthCluster; i++){
35         freeMemoryClusterCenter(clusterCenterPtr[i]);
36     }
37     free(clusterCenterPtr);
38 }
39
40 /* Methods */
41 double calculateDistanceCC(ClusterCenterPtr clusterCenterPtr, double *point,
42                            int length){
43     int pointDimension = clusterCenterPtr->length;
44     double totalSum = 0;
45     for(int i = 0; i < pointDimension; i++){
46         totalSum += (clusterCenterPtr->point[i] - point[i]) *
47                     (clusterCenterPtr->point[i] - point[i]);
48     }
49     return sqrt(totalSum);
50 }
51
52 double calculateDistanceCC_(int i, int j, double **dataset, int row,
53                            int column){
```

```
54      int pointDimension = column;
55      double totalSum = 0.0;
56      for(int k = 0; k < pointDimension; k++){
57          totalSum += ((dataset[i][k] - dataset[j][k]) *
58                       (dataset[i][k] - dataset[j][k]));
59      }
60      return sqrt(totalSum);
61  }
```

```
1   #include "spider.h"
2
3   void allocateMemorySpider(SpiderPtr *spiderPtr, int type,
4                             int lengthCluster, int lengthPoint,
5                             double **dataset, int row, int column){
6       *spiderPtr =  (Spider*)malloc(sizeof(Spider));
7       if (*spiderPtr == NULL){
8           printf("No have memory.\n");
9           exit(1);
10      }
11
12      (*spiderPtr)->type = type;
13      allocateMemoryArrayClusterCenter(&(*spiderPtr)->centers,
14      lengthCluster, lengthPoint);
15      (*spiderPtr)->lengthCenters = lengthCluster;
16      allocateMemoryArrayInt(&(*spiderPtr)->datasetClusters, row);
17      (*spiderPtr)->lengthDatasetClusters = row;
18      (*spiderPtr)->fitness = INT_MAX;
19      (*spiderPtr)->weight = 0;
20  }
21
22  void freeMemorySpider(SpiderPtr spiderPtr){
23      freeMemoryArrayClusterCenter(spiderPtr->centers,
24      spiderPtr->lengthCenters);
25      freeMemoryArrayInt(spiderPtr->datasetClusters);
26      free(spiderPtr);
27  }
```

```
28
29 void allocateMemoryArraySpider(SpiderPtr **spiderPtr, int lengthSpider,
30                                 int type, int lengthCluster, int lengthPoint,
31                                 double **dataset, int row, int column){
32     *spiderPtr = (SpiderPtr*)malloc(sizeof(Spider) * lengthSpider);
33     if (*spiderPtr == NULL){
34         printf("No have memory.\n");
35         exit(1);
36     }
37     for (int i = 0; i < lengthSpider; i++){
38         allocateMemorySpider(&(*spiderPtr)[i], type,
39         lengthCluster, lengthPoint, dataset, row, column);
40     }
41 }
42
43 void freeMemoryArraySpider(SpiderPtr *spiderPtr, int lengthSpider){
44     for (int i = 0; i < lengthSpider; i++){
45         freeMemorySpider(spiderPtr[i]);
46     }
47     free(spiderPtr);
48 }
49
50 /* Methods */
51 double calculateMetric(SpiderPtr spiderPtr, double **dataset,
52                        int row, int column){
53     double distance = 0;
54     int index = 0;
55     for(int i = 0; i < row; i++){
56         index = spiderPtr->datasetClusters[i];
57         distance += calculateDistanceCC(spiderPtr->centers[index],
58                     dataset[i], column);
59     }
60     return distance;
61 }
62
63 /** Calculate the dist. between "current" spider and spider "spi" **/
```

```
64 double calculateDistanceS(SpiderPtr spiderPtr, SpiderPtr spi){
65     double totalDistance = 0;
66     for(int i = 0; i < spiderPtr->lengthCenters; i++){
67         totalDistance += calculateDistanceCC(spiderPtr->centers[i],
68                                 spi->centers[i]->point,
69                                 spi->centers[i]->length);
70     }
71     return totalDistance;
72 }
73
74 void buildClusters(SpiderPtr spiderPtr, double **dataset, int row,
75                     int column, AuxSpiderPtr auxSpiderPtr){
76     int numClusters = spiderPtr->lengthCenters;
77     double min = 0;
78     int clusterIndex = 0;
79     // build the dataset clusters
80     for(int i = 0; i < row; i++){
81         for(int j = 0; j < numClusters; j++){
82             // euclidean distance from a dataset point "i" to
83             // each cluster center
84             auxSpiderPtr->distances[j] = calculateDistanceCC(
85                                             spiderPtr->centers[j],
86                                             dataset[i], column);
87         }
88         // determine minimum distance
89         min = auxSpiderPtr->distances[0];
90         clusterIndex = 0;
91         for(int j = 1; j < numClusters; j++){
92             if (auxSpiderPtr->distances[j] < min){
93                 min = auxSpiderPtr->distances[j];
94                 clusterIndex = j;
95             }
96         }
97         // assign cluster index of the minimum distance
98         // to the dataset point "i"
99         spiderPtr->datasetClusters[i] = clusterIndex;
```

```
100     }
101 }
102
103 void computeNewClusterCenters(SpiderPtr spiderPtr, double **dataset, int row,
104                                 int column, AuxSpiderPtr auxSpiderPtr){
105     // calculate total sum for each cluster
106     int numClusters = spiderPtr->lengthCenters;
107     int pointDimension = column;
108     int pos = 0;
109     int index = 0;
110     double newValue = 0;
111
112     for (int i = 0; i < numClusters; i++){
113         auxSpiderPtr->numPointsForCluster[i] = 0;
114     }
115
116     for (int i = 0; i < numClusters; i++){
117         for (int j = 0; j < pointDimension; j++){
118             auxSpiderPtr->sum[i][j] = 0;
119         }
120     }
121
122     for(int i = 0; i < row; i++){
123         index = spiderPtr->datasetClusters[i];
124         for(int j = 0; j < pointDimension; j++){
125             auxSpiderPtr->sum[index][j] += dataset[i][j];
126         }
127         auxSpiderPtr->numPointsForCluster[index]++;
128     }
129     // replace mean points as new cluster centers
130     for(int i = 0; i < numClusters; i++){
131         for(int j = 0; j < pointDimension; j++){
132             if (auxSpiderPtr->numPointsForCluster[i] == 0){
133                 // when all points of cluster have the same value
134                 // choose new value from random_ point at dataset
135                 pos = randomInt(row);
```

```
136                 newValue = dataset[pos][j];
137             }
138             else{
139                 // calculate mean point
140                 newValue = auxSpiderPtr->sum[i][j] /
141                             auxSpiderPtr->numPointsForCluster[i];
142             }
143             spiderPtr->centers[i]->point[j] = newValue;
144         }
145     }
146 }
147
148 void evaluateFitness(PopulationPtr populationPtr, SpiderPtr spiderPtr,
149                         double **dataset, int row, int column,
150                         AuxSpiderPtr auxSpiderPtr){
151     // build clusters according to its centers
152     buildClusters(spiderPtr, dataset, row, column, auxSpiderPtr);
153     // compute new cluster centers
154     computeNewClusterCenters(spiderPtr, dataset, row, column, auxSpiderPtr);
155     // calculate clustering metric
156     spiderPtr->fitness = calculateMetric(spiderPtr, dataset, row, column);
157
158     populationPtr->countFitness += 1;
159 }
160
161 /** Difference of the cluster centers of two spiders **/
162 void diffSpiders(SpiderPtr spi1, SpiderPtr spi2, ClusterCenterPtr *c,
163                     int length){
164     for (int i = 0; i < spi1->lengthCenters; i++){
165         for (int j = 0; j < spi1->centers[0]->length; j++){
166             c[i]->point[j] = spi1->centers[i]->point[j] -
167                                 spi2->centers[i]->point[j];
168         }
169     }
170 }
171
```

```
172 /** Multiply cluster centers by a constant **/
173 void sumSpider(SpiderPtr spiderPtr, ClusterCenterPtr *c, int length,
174                double signal){
175     for (int i = 0; i < spiderPtr->lengthCenters; i++){
176         for (int j = 0; j < spiderPtr->centers[0]->length; j++){
177             spiderPtr->centers[i]->point[j] += signal * c[i]->point[j];
178         }
179     }
180 }
181
182 /** Multiply cluster centers by a constant **/
183 void mulClusterCentersByConstant(ClusterCenterPtr *c, int length,
184                                  double cons){
185     for(int i = 0; i < length; i++){
186         for(int j = 0; j < c[0]->length; j++){
187             c[i]->point[j] *= cons;
188         }
189     }
190 }
191
192 void showSpider(SpiderPtr spiderPtr){
193     printf("%s", "[");
194     for(int i = 0; i < spiderPtr->lengthCenters; i++){
195         printf("%s", "(");
196         for(int j = 0; j < spiderPtr->centers[0]->length; j++){
197             printf("%f; ", spiderPtr->centers[i]->point[j]);
198         }
199         printf("%s", "),");
200     }
201     printf("] Fitness: %f \n", spiderPtr->fitness);
202 }
```

```
1 #include "population.h"
2
3 void freeMemoryPopulation(PopulationPtr populationPtr,
4                           AuxPopulationPtr auxPopulationPtr,
```

```
                                AuxSpiderPtr auxSpiderPtr){
    freeMemoryArraySpider(populationPtr->spiders,
    populationPtr->lengthSpider);
    freeMemoryArraySpider(populationPtr->offspring,
    populationPtr->numberMales);
    freeMemoryArrayDictionary(populationPtr->fitness,
    populationPtr->lengthSpider);
    freeMemoryArrayInt(populationPtr->indexMigration);
    freeMemoryAuxilarPopulation(populationPtr, auxPopulationPtr);
    freeMemoryAuxilarSpider(populationPtr, auxSpiderPtr);
}

/* Methods */
double maximumDistance(double **dataset, int row, int column){
    double maximumDistance = 0;
    double distance = 0;
    for(int i = 0; i < row - 1; i++){
        for(int j = i + 1; j < row; j++){
            distance = calculateDistanceCC_(i, j, dataset, row, column);
            if (distance > maximumDistance){
                maximumDistance = distance;
            }
        }
    }
    return maximumDistance;
}

/** Generate initial population: males and females, and calculate radius **/
void generateInitialPopulation(PopulationPtr populationPtr,
                               int populationSize, int numberClusters,
                               double **dataset, int row, int column,
                               int numberMigration,
                               AuxPopulationPtr auxPopulationPtr,
                               AuxSpiderPtr auxSpiderPtr){
    populationPtr->numberFemales = floor((0.9 - random_() * 0.25) *
                                    populationSize);
```

```
41      populationPtr->numberMales = populationSize -
42                                      populationPtr->numberFemales;
43      populationPtr->numClusters = numberClusters;
44      populationPtr->pointDimension = column;
45      populationPtr->radiusMating = 0;
46      populationPtr->lengthSpider = populationSize;
47      populationPtr->lengthOffspring = populationPtr->numberMales;    // 0;
48      populationPtr->medianWeight = 0;
49      populationPtr->indexBest = 0;
50      populationPtr->indexWorst = 0;
51      populationPtr->countFitness = 0;
52      int pos = -1;   // invalid position
53      double d = 0;
54
55      // allocate memory for dictionary fitness
56      allocateMemoryArrayDictionary(&populationPtr->fitness, populationSize);
57
58      // allocate memory for indexMigration
59      allocateMemoryArrayInt(&populationPtr->indexMigration, numberMigration);
60
61      // allocate memory for offspring
62      allocateMemoryArraySpider(&populationPtr->offspring,
63      populationPtr->numberMales, 0, populationPtr->numClusters,
64      populationPtr->pointDimension, dataset, row, column);
65
66      // allocate memory for spider
67      allocateMemoryArraySpider(&populationPtr->spiders,
68      populationPtr->lengthSpider, 0, populationPtr->numClusters,
69      populationPtr->pointDimension, dataset, row, column);
70
71      // generate FEMALES by choosing points randomly from dataset
72      for(int i = 0; i < populationPtr->numberFemales; i++){
73          populationPtr->spiders[i]->type = 0;
74          for(int j = 0; j < populationPtr->numClusters; j++){
75              // random_ integer in range [0 , dataset.length - 1]
76              pos = randomInt(row);
```

```
77              for(int k = 0; k < populationPtr->pointDimension; k++){
78                  populationPtr->spiders[i]->
79                  centers[j]->point[k] = dataset[pos][k];
80                  // printf("%f ", populationPtr->spiders[i]->
81                              centers[j]->point[k]);
82              }
83          }
84          // printf("\n");
85      }
86
87      // generate MALES by choosing points randomly from dataset
88      for(int i = populationPtr->numberFemales; i <
89          populationPtr->lengthSpider; i++){
90          populationPtr->spiders[i]->type = 1;
91          for(int j = 0; j < populationPtr->numClusters; j++){
92              // random_ integer in range [0 , dataset.length - 1]
93              pos = randomInt(row);
94              for(int k = 0; k < populationPtr->pointDimension; k++){
95                  populationPtr->spiders[i]->
96                  centers[j]->point[k] = dataset[pos][k];
97                  // printf("%f ", populationPtr->spiders[i]->
98                              centers[j]->point[k]);
99              }
100         }
101         // printf("\n");
102     }
103
104     // calculate the radius of mating
105     d = maximumDistance(dataset, row, column);
106     populationPtr->radiusMating = d / 2;
107
108     // allocate memory for auxSpiderPtr var
109     allocateMemoryArrayDouble(&auxSpiderPtr->distances,
110                               populationPtr->numClusters);
111     allocateMemoryArrayInt(&auxSpiderPtr->numPointsForCluster,
112                             populationPtr->numClusters);
```

```
113     allocateMemoryMatrixDouble(&auxSpiderPtr->sum,
114                                populationPtr->numClusters,
115                                populationPtr->pointDimension);
116
117     // allocate memory for auxPopulationPtr var
118     allocateMemoryArrayDouble(&auxPopulationPtr->males,
119                                   populationPtr->numberMales);
120     allocateMemoryArrayClusterCenter(&auxPopulationPtr->spidersWeightsPtr,
121                                         populationPtr->numClusters,
122                                         populationPtr->pointDimension);
123     // before numberMales
124     allocateMemoryArrayInt(&auxPopulationPtr->matingGroup,
125                                populationPtr->numberFemales + 1);
126     // before numberMales
127     allocateMemoryArrayDouble(&auxPopulationPtr->matingRoulette,
128                                   populationPtr->numberFemales + 1);
129     allocateMemorySpider(&auxPopulationPtr->spiderWeightedMeanPtr, 1,
130                          populationPtr->numClusters,
131                          populationPtr->pointDimension, dataset,
132                          row, column);
133     allocateMemoryArrayClusterCenter(&auxPopulationPtr->clusCentersPtr,
134                                         populationPtr->numClusters,
135                                         populationPtr->pointDimension);
136     allocateMemoryArrayDouble(&auxPopulationPtr->replacementRoulette,
137                                   populationPtr->lengthSpider);
138 }
139
140 void readInitialPopulation(PopulationPtr populationPtr, int populationSize,
141                            int numberClusters, double **dataset,
142                            int row, int column, int numberMigration,
143                            AuxPopulationPtr auxPopulationPtr,
144                            AuxSpiderPtr auxSpiderPtr, double **population,
145                            int initialIndexPopulation,
146                            int limitCountFitness){
147     populationPtr->numberFemales = floor((0.9 - random_() * 0.25) *
148                                      populationSize);
```

```
149     populationPtr->numberMales = populationSize - populationPtr->
150                                  numberFemales;
151     populationPtr->numClusters = numberClusters;
152     populationPtr->pointDimension = column;
153     populationPtr->radiusMating = 0;
154     populationPtr->lengthSpider = populationSize;
155     populationPtr->lengthOffspring = populationPtr->numberMales;    // 0;
156     populationPtr->medianWeight = 0;
157     populationPtr->indexBest = 0;
158     populationPtr->indexWorst = 0;
159     populationPtr->countFitness = 0;
160     populationPtr->limitCountFitness = limitCountFitness;
161     /// int pos = -1;   // invalid position
162     double d = 0;
163
164     // allocate memory for dictionary fitness
165     allocateMemoryArrayDictionary(&populationPtr->fitness, populationSize);
166
167     // allocate memory for indexMigration
168     allocateMemoryArrayInt(&populationPtr->indexMigration, numberMigration);
169
170     // allocate memory for offspring
171     allocateMemoryArraySpider(&populationPtr->offspring,
172                               populationPtr->numberMales, 0,
173                               populationPtr->numClusters,
174                               populationPtr->pointDimension,
175                               dataset, row, column);
176
177     // allocate memory for spider
178     allocateMemoryArraySpider(&populationPtr->spiders,
179                               populationPtr->lengthSpider,
180                               0, populationPtr->numClusters,
181                               populationPtr->pointDimension,
182                               dataset, row, column);
183
184     // generate FEMALES by choosing points randomly from dataset
```

```
185     for(int i = 0; i < populationPtr->numberFemales; i++){
186         populationPtr->spiders[i]->type = 0;
187         for(int j = 0; j < populationPtr->numClusters; j++){
188             // random_ integer in range [0 , dataset.length - 1]
189             /// pos = randomInt(row);
190             for(int k = 0; k < populationPtr->pointDimension; k++){
191                 /// populationPtr->spiders[i]->centers[j]->
192                 point[k] = dataset[pos][k];
193                 populationPtr->spiders[i]->centers[j]->
194                 point[k] = population[i + initialIndexPopulation][j *
195                             populationPtr->pointDimension + k];
196                 // printf("%f ", populationPtr->spiders[i]->
197                             centers[j]->point[k]);
198             }
199         }
200         // printf("\n");
201     }
202
203     // generate MALES by choosing points randomly from dataset
204     for(int i = populationPtr->numberFemales; i < populationPtr->
205         lengthSpider; i++){
206         populationPtr->spiders[i]->type = 1;
207         for(int j = 0; j < populationPtr->numClusters; j++){
208             // random_ integer in range [0 , dataset.length - 1]
209             /// pos = randomInt(row);
210             for(int k = 0; k < populationPtr->pointDimension; k++){
211                 /// populationPtr->spiders[i]->centers[j]->
212                 point[k] = dataset[pos][k];
213                 populationPtr->spiders[i]->centers[j]->
214                 point[k] = population[i + initialIndexPopulation][j *
215                             populationPtr->pointDimension + k];
216                 // printf("%f ", populationPtr->spiders[i]->
217                             centers[j]->point[k]);
218             }
219         }
220         // printf("\n");
```

```
221     }
222
223     // calculate the radius of mating
224     d = maximumDistance(dataset, row, column);
225     populationPtr->radiusMating = d / 2;
226
227     // allocate memory for auxSpiderPtr var
228     allocateMemoryArrayDouble(&auxSpiderPtr->distances,
229                               populationPtr->numClusters);
230     allocateMemoryArrayInt(&auxSpiderPtr->numPointsForCluster,
231                            populationPtr->numClusters);
232     allocateMemoryMatrixDouble(&auxSpiderPtr->sum,
233                               populationPtr->numClusters,
234                               populationPtr->pointDimension);
235
236     // allocate memory for auxPopulationPtr var
237     allocateMemoryArrayDouble(&auxPopulationPtr->males,
238                               populationPtr->numberMales);
239     allocateMemoryArrayClusterCenter(&auxPopulationPtr->spidersWeightsPtr,
240                                      populationPtr->numClusters,
241                                      populationPtr->pointDimension);
242     // before numberMales
243     allocateMemoryArrayInt(&auxPopulationPtr->matingGroup,
244                            populationPtr->numberFemales + 1);
245     // before numberMales
246     allocateMemoryArrayDouble(&auxPopulationPtr->matingRoulette,
247                               populationPtr->numberFemales + 1)
248     allocateMemorySpider(&auxPopulationPtr->spiderWeightedMeanPtr, 1,
249                            populationPtr->numClusters,
250                            populationPtr->pointDimension, dataset,
251                            row, column);
252     allocateMemoryArrayClusterCenter(&auxPopulationPtr->clusCentersPtr,
253                                      populationPtr->numClusters,
254                                      populationPtr->pointDimension);
255     allocateMemoryArrayDouble(&auxPopulationPtr->replacementRoulette,
256                               populationPtr->lengthSpider);
```

```
257 }
258
259 void generatePopulation(int populationSize, int numberClusters,
260                         double **dataset, int row, int column){
261     // int numberFemales;
262     // int numberMales = populationSize - numberFemales;
263     int numClusters = numberClusters;
264     int pointDimension = column;
265     int lengthSpider = populationSize;
266
267     int pos = 0;
268
269     /* char fileName[256];
270     sprintf(fileName, "%s", fileNameOut);
271     FILE *fp = fopen(fileName, "w"); */
272     for(int i = 0; i < lengthSpider; i++){
273         for(int j = 0; j < numClusters; j++){
274             // random_ integer in range [0 , dataset.length - 1]
275             pos = randomInt(row);
276             //printf("%d %d\n", row, pos);
277             for(int k = 0; k < pointDimension; k++){
278                 if (j * k == (numClusters - 1) * (pointDimension - 1)) {
279                     // fprintf(fp, "%f", dataset[pos][k]);
280                     printf("%f", dataset[pos][k]);
281                 } else {
282                     // fprintf(fp, "%f,", dataset[pos][k]);
283                     printf("%f,", dataset[pos][k]);
284                 }
285             }
286         }
287         // fprintf(fp, "%s", "\n");
288         printf("\n");
289     }
290 }
291
292 void evaluateFitnessPopulation(PopulationPtr populationPtr,
```

```
                                double **dataset, int row, int column,
                                AuxSpiderPtr auxSpiderPtr){
    for(int i = 0; i < populationPtr->lengthSpider; i++){
        // keeping the best spider the same
        if (i == populationPtr->indexBest){
            continue;
        }
        evaluateFitness(populationPtr, populationPtr->spiders[i],
                        dataset, row, column, auxSpiderPtr);
    }
}

void calculateBestWorstPopulation(PopulationPtr populationPtr){
    double minFitness = populationPtr->spiders[0]->fitness;
    double maxFitness = populationPtr->spiders[0]->fitness;

    for(int i = 0; i < populationPtr->lengthSpider; i++){
        populationPtr->fitness[i]->key = i;
        populationPtr->fitness[i]->value = populationPtr->
                                           spiders[i]->fitness;

        if (populationPtr->spiders[i]->fitness < minFitness){
            minFitness = populationPtr->spiders[i]->fitness;
            populationPtr->indexBest = i;
        }

        if (populationPtr->spiders[i]->fitness > maxFitness){
            maxFitness = populationPtr->spiders[i]->fitness;
            populationPtr->indexWorst = i;
        }
    }

    // Insert sort by fitness
    insertSort(populationPtr->fitness, populationPtr->lengthSpider);
}

```

```
329 /** Calculate the weight of every spider in the population **/
330 void calculateWeightPopulation(PopulationPtr populationPtr,
331                                double **dataset, int row, int column,
332                                AuxSpiderPtr auxSpiderPtr){
333     // Evaluate the fitness of the population
334     evaluateFitnessPopulation(populationPtr, dataset, row, column,
335                                auxSpiderPtr);
336     // Calculate the best and the worst individual of pop.
337     calculateBestWorstPopulation(populationPtr);
338     // Calculate weight of population
339     double w = 0;
340     for(int i = 0; i < populationPtr->lengthSpider; i++){
341         w = (populationPtr->spiders[populationPtr->indexWorst]->fitness -
342             populationPtr->spiders[i]->fitness) / (populationPtr->
343             spiders[populationPtr->indexWorst]->fitness -
344             populationPtr->spiders[populationPtr->indexBest]->fitness);
345 
346         populationPtr->spiders[i]->weight = w;
347     }
348 }
349 
350 /** Return the index of the nearest individual with higher weight
351 compared to individual with index i **/
352 int nearestWithHigherWeightTo(PopulationPtr populationPtr, int i){
353     int index = 0;
354     double nearestDistance = calculateDistanceS(populationPtr->spiders[index],
355                                                 populationPtr->spiders[i]);
356 
357     for(int k = 0; k < populationPtr->lengthSpider; k++){
358         if (populationPtr->spiders[k]->weight > populationPtr->
359             spiders[i]->weight){
360             double newDistance = calculateDistanceS(populationPtr->
361                                     spiders[k], populationPtr->spiders[i]);
362             if (newDistance < nearestDistance){
363                 nearestDistance = newDistance;
364                 index = k;
```

```
365             }
366         }
367     }
368     return index;
369 }
370 
371 /** Apply the female cooperative operator **/
372 void femaleCooperativeOperator(PopulationPtr populationPtr,
373                                AuxPopulationPtr auxPopulationPtr){
374     double thresholdPF = 0.5;   // Test with other values
375     double w = 0, d = 0, d2 = 0;
376     double vibci = 0;
377     double vibbi = 0;
378     double rm = 0, alpha = 0, beta = 0, delta = 0, rand = 0;
379     int index = 0;
380     double cons = 0, signal = 0;
381 
382     for(int i = 0; i < populationPtr->numberFemales; i++){
383         // keeping the best spider the same
384         if (i == populationPtr->indexBest){
385             continue;
386         }
387         // Calculate Vibci: Vib. of ind. nearest with higher weight compared to i
388         index = nearestWithHigherWeightTo(populationPtr, i);
389         w = populationPtr->spiders[index]->weight;
390         d = calculateDistanceS(populationPtr->spiders[i],
391                                populationPtr->spiders[index]);
392         d2 = pow(d, 2);
393         vibci = w / pow(M_E, d2);
394 
395         // Calculate Vibbi: vibration of individual with best fitness
396         w = populationPtr->spiders[populationPtr->indexBest]->weight;
397         d = calculateDistanceS(populationPtr->spiders[i], populationPtr->
398                                spiders[populationPtr->indexBest]);
399         d2 = pow(d, 2);
400         vibbi = w / pow(M_E, d2);
```

```
401
402         rm = random_();
403         alpha = random_();
404         beta = random_();
405         delta = random_();
406         rand = random_();
407
408         // Define if the movement is: attraction or repulsion
409         if (rm < thresholdPF){
410             signal = 1;     // --> 1=sum
411         }
412         else{
413             signal = -1;    // --> -1=subtraction
414         }
415
416         // Sum expression with alpha
417         diffSpiders(populationPtr->spiders[index], populationPtr->
418                     spiders[i], auxPopulationPtr->clusCentersPtr,
419                     populationPtr->numClusters);
420         cons = alpha * vibci;
421         mulClusterCentersByConstant(auxPopulationPtr->clusCentersPtr,
422                                     populationPtr->numClusters, cons);
423         sumSpider(populationPtr->spiders[i], auxPopulationPtr->clusCentersPtr,
424                     populationPtr->numClusters, signal);
425
426         // Sum expression with beta
427         diffSpiders(populationPtr->spiders[populationPtr->indexBest],
428                     populationPtr->spiders[i], auxPopulationPtr->
429                     clusCentersPtr, populationPtr->numClusters);
430         cons = beta * vibbi;
431         mulClusterCentersByConstant(auxPopulationPtr->clusCentersPtr,
432                                     populationPtr->numClusters, cons);
433         sumSpider(populationPtr->spiders[i], auxPopulationPtr->clusCentersPtr,
434                     populationPtr->numClusters, signal);
435
436         // Sum expression with delta
```

```
437         for(int j = 0; j < populationPtr->numClusters; j++){
438             for(int k = 0; k < populationPtr->pointDimension; k++){
439                 auxPopulationPtr->clusCentersPtr[j]->point[k] = delta *
440                                                                 (rand - 0.5);
441             }
442         }
443         sumSpider(populationPtr->spiders[i], auxPopulationPtr->
444                     clusCentersPtr, populationPtr->numClusters, signal);
445     }
446 }
447
448 /* calculate median weight of males using variant of quicksort */
449 void calculateMedianWeightOfMales(PopulationPtr populationPtr,
450                                   AuxPopulationPtr auxPopulationPtr){
451     int k = 0;
452     int medianPos = 0;
453     int begin, end, p, r, i = 0;
454     double pivot = 0;
455     double tmp = 0;
456
457     // Copy the weight of males into males array
458     for(int i = populationPtr->numberFemales; i < populationPtr->
459         lengthSpider; i++){
460         auxPopulationPtr->males[k] = populationPtr->spiders[i]->weight;
461         k++;
462     }
463     // Calculate the median position
464     medianPos = (0 + populationPtr->numberMales - 1) / 2;
465     // Calculate the median value
466     begin = 0;
467     end = populationPtr->numberMales - 1;
468
469     while(TRUE){
470         p = begin;
471         r = end;
472         pivot = auxPopulationPtr->males[r];
```

```
473             i = p - 1;
474             for(int j = p; j < r; j++){
475                 if (auxPopulationPtr->males[j] < pivot){
476                     i++;
477                     // swap males[i] and males[j]
478                     tmp = auxPopulationPtr->males[i];
479                     auxPopulationPtr->males[i] = auxPopulationPtr->males[j];
480                     auxPopulationPtr->males[j] = tmp;
481                 }
482             }
483             // swap males[i+1] and males[r]
484             tmp = auxPopulationPtr->males[i + 1];
485             auxPopulationPtr->males[i + 1] = auxPopulationPtr->males[r];
486             auxPopulationPtr->males[r] = tmp;
487 
488             if (i + 1 == medianPos){
489                 break;
490             }
491             else{
492                 if (medianPos > i + 1){
493                     // search right
494                     begin = i + 2;
495                 }
496                 else{
497                     // search left
498                     end = i;
499                 }
500             }
501         }
502         populationPtr->medianWeight = auxPopulationPtr->males[medianPos];
503     }
504 
505     int nearestFemaleTo(PopulationPtr populationPtr, int i){
506         int index = 0;
507         double nearestDistance = calculateDistanceS(populationPtr->
508                                     spiders[index], populationPtr->spiders[i]);
```

```
509     double newDistance = 0;
510     for(int k = 0; k < populationPtr->numberFemales; k++){  // females
511         newDistance = calculateDistanceS(populationPtr->spiders[k],
512                         populationPtr->spiders[i]);
513         if (newDistance < nearestDistance){
514             nearestDistance = newDistance;
515             index = k;
516         }
517     }
518     return index;
519 }
520
521 void calculateMaleSpiderWeightedMean(PopulationPtr populationPtr,
522                                      SpiderPtr spi,
523                                      AuxPopulationPtr auxPopulationPtr){
524     // calculate total weight of males
525     double totalWeight = 0;
526     for(int i = populationPtr->numberFemales; i < populationPtr->
527         lengthSpider; i++){
528         totalWeight += populationPtr->spiders[i]->weight;
529     }
530
531     // calculate spiders multiplied by their weights
532     for(int i = 0; i < populationPtr->numClusters; i++){
533         for(int j = 0; j < populationPtr->pointDimension; j++){
534             auxPopulationPtr->spidersWeightsPtr[i]->point[j] = 0;
535         }
536     }
537
538     for(int i = populationPtr->numberFemales; i < populationPtr->
539         lengthSpider; i++){
540         for(int j = 0; j < populationPtr->numClusters; j++){
541             for(int k = 0; k < populationPtr->pointDimension; k++){
542                 auxPopulationPtr->spidersWeightsPtr[j]->point[k] +=
543                 populationPtr->spiders[i]->centers[j]->point[k] *
544                 populationPtr->spiders[i]->weight;
```

```
545             }
546         }
547     }
548
549     // calculate the weighted mean
550     for(int j = 0; j < populationPtr->numClusters; j++){
551         for(int k = 0; k < populationPtr->pointDimension; k++){
552             spi->centers[j]->point[k] = auxPopulationPtr->
553             spidersWeightsPtr[j]->point[k] / totalWeight;
554         }
555     }
556 }
557
558 /** Create Mating Roulette for a male spider **/
559 void createMatingRoulette(PopulationPtr populationPtr,
560                           double *matingRoulette, int lengthMatingRoulette,
561                           int  *matingGroup, int lengthMatingGroup){
562     // Sum fitness of mating spiders
563     double total =  0;
564     for(int i = 0; i < lengthMatingGroup; i++){
565         total += populationPtr->spiders[matingGroup[i]]->fitness;
566     }
567     // Calculate values of the roulette
568     matingRoulette[0] = populationPtr->spiders[matingGroup[0]]->
569                         fitness / total;
570     for(int i = 1; i < lengthMatingGroup; i++){
571         matingRoulette[i] = matingRoulette[i - 1] + populationPtr->
572                           spiders[matingGroup[i]]->fitness / total;
573     }
574 }
575
576 /** Apply the mating operator **/
577 void matingOperator(PopulationPtr populationPtr, double **dataset,
578                     int row, int column,
579                     AuxPopulationPtr auxPopulationPtr,
580                     AuxSpiderPtr auxSpiderPtr){
```

```
581     int indexOffspring = 0;
582     int indexGroup = 0;
583     double rand = -1;
584     double distance = 0;
585 
586     // Begin mating
587     for(int i = populationPtr->numberFemales; i < populationPtr->
588         lengthSpider; i++){    // males
589         if (populationPtr->spiders[i]->weight > populationPtr->
590             medianWeight){   // male is dominant
591             indexGroup = 0;
592             // Calculate females in the radius of male "i"
593             auxPopulationPtr->matingGroup[indexGroup] = i;
594             // add male index as first element
595             indexGroup++;
596             for(int j = 0; j < populationPtr->numberFemales; j++){
597                 // females
598                 distance = calculateDistanceS(populationPtr->spiders[i],
599                             populationPtr->spiders[j]);
600                 if (distance < populationPtr->radiusMating){
601                     auxPopulationPtr->matingGroup[indexGroup] = j;
602                     // add female index
603                     indexGroup++;
604                 }
605             }
606             if (indexGroup > 1){    // do mating
607                 // Create mating roulette
608                 createMatingRoulette(populationPtr, auxPopulationPtr->
609                                      matingRoulette, indexGroup,
610                                      auxPopulationPtr->matingGroup,
611                                      indexGroup);
612                 // Create the new spider using mating roulette
613                 for(int j = 0; j < populationPtr->numClusters; j++){
614                     rand = random_();   // 0.0 <= rand < 1.0
615                     // Go through the mating roulette
616                     for(int k = 0; k < indexGroup; k++){
```

```
617                         if (rand < auxPopulationPtr->matingRoulette[k]){
618                             // Copy cluster "j"
619                             for(int h = 0; h < populationPtr->
620                                 pointDimension; h++){
621                                 populationPtr->offspring[indexOffspring]->
622                                 centers[j]->point[h] = populationPtr->
623                                 spiders[auxPopulationPtr->
624                             }
625                             break;
626                         }
627                     }
628                 }
629                 // Calculate fitness of new spider and put into offspring
630                 evaluateFitness(populationPtr, populationPtr->
631                                 offspring[indexOffspring], dataset,
632                                 row, column, auxSpiderPtr);
633                 indexOffspring++;
634                 populationPtr->lengthOffspring = indexOffspring;
635             }
636         }
637     }   // end-mating
638 }
639
640 // using a factor
641 void createReplacementRoulette(PopulationPtr populationPtr,
642                                 double *replacementRoulette, int length){
643     // Sum fitness of all spiders
644     double factor = 1;
645     double total = 0;
646     for(int i = 0; i < populationPtr->lengthSpider; i++){
647         if (populationPtr->spiders[i]->weight > populationPtr->
648             medianWeight){
649             total += populationPtr->spiders[i]->fitness;
650         }
651         else{
652             total += populationPtr->spiders[i]->fitness * factor;
```

```
653         }
654     }
655 
656     // calculate values of the roulette
657     replacementRoulette[0] = populationPtr->spiders[0]->fitness / total;
658     for(int i = 1; i < populationPtr->lengthSpider; i++){
659         if (populationPtr->spiders[i]->weight > populationPtr->
660             medianWeight){
661             replacementRoulette[i] = replacementRoulette[i - 1] +
662                                      populationPtr->spiders[i]->
663                                      fitness / total;
664         }
665         else{
666             replacementRoulette[i] = replacementRoulette[i - 1] +
667             (populationPtr->spiders[i]->fitness * factor) / total;
668         }
669     }
670 }
671 
672 /** Apply the male cooperative operator **/
673 void maleCooperativeOperator(PopulationPtr populationPtr,
674                              double **dataset, int row, int column,
675                              AuxPopulationPtr auxPopulationPtr){
676     double w = 0, d = 0, d2 = 0;
677     double vibfi = 0;
678     double alpha = 0, delta = 0, rand = 0, cons = 0;
679     int index = 0;
680 
681     // Calculate the median weight of male population
682     calculateMedianWeightOfMales(populationPtr, auxPopulationPtr);
683     // Calculate the male spider with weighted mean
684     for(int i = 0; i < populationPtr->numClusters; i++){
685         for(int j = 0; j < populationPtr->pointDimension; j++){
686             auxPopulationPtr->clusCentersPtr[i]->point[j] = 0;
687         }
688     }
```

```
689
690     calculateMaleSpiderWeightedMean(populationPtr, auxPopulationPtr->
691                                     spiderWeightedMeanPtr, auxPopulationPtr);
692
693     for(int i = populationPtr->numberFemales; i < populationPtr->
694         lengthSpider; i++){    // males
695         // keeping the best spider the same
696         if (i == populationPtr->indexBest){
697             continue;
698         }
699
700         // calculate vibfi: vibration of nearest female
701         index = nearestFemaleTo(populationPtr, i);
702         w = populationPtr->spiders[index]->weight;
703         d = calculateDistanceS(populationPtr->spiders[i], populationPtr->
704                                spiders[index]);
705         d2 = pow(d, 2);
706         vibfi = w / pow(M_E, d2);
707
708         // Define if movement is attraction to females or to the mean
709         alpha = random_();
710         delta = random_();
711         rand = random_();
712         if (populationPtr->spiders[i]->weight > populationPtr->
713             medianWeight){   // male is dominant(D)
714             // Sum expression with alpha
715             diffSpiders(populationPtr->spiders[index], populationPtr->
716                         spiders[i], auxPopulationPtr->clusCentersPtr,
717                         populationPtr->numClusters);
718             cons = alpha * vibfi;
719             mulClusterCentersByConstant(auxPopulationPtr->clusCentersPtr,
720                                         populationPtr->numClusters, cons);
721             // sum = 1
722             sumSpider(populationPtr->spiders[i], auxPopulationPtr->
723                         clusCentersPtr, populationPtr->numClusters, 1);
724
```

```
725             // Sum expression with delta
726             for(int j = 0; j < populationPtr->numClusters; j++){
727                 for(int k = 0; k < populationPtr->pointDimension; k++){
728                     auxPopulationPtr->clusCentersPtr[j]->point[k] =
729                     delta * (rand - 0.5);
730                 }
731             }
732             sumSpider(populationPtr->spiders[i], auxPopulationPtr->
733             clusCentersPtr, populationPtr->numClusters, 1);  // sum = 1
734         }
735         else{
736             // male is not dominant(ND)
737             // Sum expression with alpha
738             diffSpiders(auxPopulationPtr->spiderWeightedMeanPtr,
739                         populationPtr->spiders[i], auxPopulationPtr->
740                         clusCentersPtr, populationPtr->numClusters);
741             mulClusterCentersByConstant(auxPopulationPtr->clusCentersPtr,
742                                         populationPtr->numClusters, alpha);
743             // sum = 1
744             sumSpider(populationPtr->spiders[i], auxPopulationPtr->
745                         clusCentersPtr, populationPtr->numClusters, 1);
746         }
747     }
748 }
749
750 /** Replace offspring into spiders **/
751 void replacement(PopulationPtr populationPtr,
752                     AuxPopulationPtr auxPopulationPtr){
753     double rand = 0;
754     // Create replacement roulette for all spiders, giving more prob. to worst spi.
755     createReplacementRoulette(populationPtr, auxPopulationPtr->
756                                 replacementRoulette,
757                                 populationPtr->lengthSpider);
758     // Replace worst spider by offspring by comparing its fitness
759     for(int i = 0; i < populationPtr->lengthOffspring; i++){
760         rand = random_();    // 0.0 <= rand < 1.0
```

```
761         // Go through the replacement roulette
762         for(int j = 0; j < populationPtr->lengthSpider; j++){
763             if (rand < auxPopulationPtr->replacementRoulette[j]){
764                 // replace spider "j" if it is worst than offspring "i"
765                 if (populationPtr->offspring[i]->fitness < populationPtr->
766                     spiders[j]->fitness){
767                     for(int k = 0; k < populationPtr->numClusters; k++){
768                         for(int h = 0; h < populationPtr->
769                             pointDimension; h++){
770                             populationPtr->spiders[j]->centers[k]->
771                             point[h] = populationPtr->offspring[i]->
772                                         centers[k]->point[h];
773                         }
774                     }
775                     populationPtr->spiders[j]->fitness = populationPtr->
776                                                          offspring[i]->fitness;
777                     populationPtr->spiders[j]->weight = 0;
778                     for(int k = 0; k < populationPtr->offspring[i]->
779                         lengthDatasetClusters; k++){
780                         populationPtr->spiders[j]->
781                         datasetClusters[k] = populationPtr->offspring[i]->
782                                              datasetClusters[k];
783                     }
784                 }
785                 break;
786             }
787         }
788     }
789 }
790
791 void savePopulation(PopulationPtr populationPtr){
792     for(int i = 0; i < populationPtr->lengthSpider; i++){
793         for(int j = 0; j < populationPtr->numClusters; j++){
794             for(int k = 0; k < populationPtr->pointDimension; k++){
795                 if (j * k == (populationPtr->numClusters - 1) *
796                     (populationPtr->pointDimension - 1)) {
```

```
797                     printf("%f", populationPtr->spiders[i]->
798                             centers[j]->point[k]);
799                 } else {
800                     printf("%f,", populationPtr->spiders[i]->
801                             centers[j]->point[k]);
802                 }
803             }
804         }
805         printf("\n");
806     }
807 }
808
809 /** Show best fitness **/
810 void showBestFitness(PopulationPtr populationPtr, int numberGeneration){
811     printf("Generation: %d, Best Fitness [%d]: %f, Worst Fitness [%d]: %f \n",
812         numberGeneration, populationPtr->indexBest,
813         populationPtr->spiders[populationPtr->indexBest]->fitness,
814         populationPtr->indexWorst,
815         populationPtr->spiders[populationPtr->indexWorst]->fitness);
816 }
817
818 /** Show Population **/
819 void showPopulation(PopulationPtr populationPtr){
820     printf("\nPOPULATION\n");
821     for(int i = 0; i < populationPtr->lengthSpider; i++){
822         printf("%d: [", i);
823         for(int j = 0; j < populationPtr->numClusters; j++){
824             printf("(");
825             for(int k = 0; k < populationPtr->pointDimension; k++){
826                 printf("%f;", populationPtr->spiders[i]->centers[j]->
827                         point[k]);
828             }
829             printf("),");
830         }
831         printf("\n");
832         // printf("] Fitness: %f \n", populationPtr->spiders[i]->fitness);
```

```
833     }
834     printf("\n");
835 }
836 
837 /** Show some parameters **/
838 void showParameters(PopulationPtr populationPtr, int numberGenerations,
839                     int numberDataset){
840     printf("\n");
841     printf("-------------------------------\n");
842     printf("---- S. Spider Algorithm -----\n");
843     printf("-------------------------------\n");
844     printf("Number of Dataset:      \t%d\n", numberDataset);
845     printf("Number of Generations:  \t%d\n", numberGenerations);
846     printf("Population Size:        \t%d\n", populationPtr->lengthSpider);
847     printf("Number of females:      \t%d\n", populationPtr->numberFemales);
848     printf("Number of males:        \t%d\n", populationPtr->numberMales);
849 }
850 
851 void showBestSpider(PopulationPtr populationPtr){
852     printf("\nBest Spider: \n");
853     printf("[");
854     for(int j = 0; j < populationPtr->numClusters; j++){
855         printf("(");
856         for(int k = 0; k < populationPtr->pointDimension; k++){
857             printf("%f;", populationPtr->spiders[populationPtr->
858                     indexBest]->centers[j]->point[k]);
859         }
860         printf("),");
861     }
862     printf("] \n");
863     printf("Index best:  \t%d,  \tFitness: \t%f \n", populationPtr->
864             indexBest, populationPtr->spiders[populationPtr->
865             indexBest]->fitness);
866     printf("Index worst: \t%d,  \tFitness: \t%f \n", populationPtr->
867             indexWorst, populationPtr->spiders[populationPtr->
868             indexWorst]->fitness);
```

```
869     /* printf("Index best:  \t%d,    \tFitness: \t%f \n", populationPtr->
870                 fitness[0]->key, populationPtr->fitness[0]->value);
871     printf("Index worst: \t%d,    \tFitness: \t%f \n", populationPtr->
872             fitness[populationPtr->lengthSpider - 1]->key, populationPtr->
873             fitness[populationPtr->lengthSpider - 1]->value);
874     for (int i = 0; i < populationPtr->lengthSpider; i++) {
875         printf("%d\t%d\t%f\n", i, populationPtr->fitness[i]->key,
876         populationPtr->fitness[i]->value);
877     } */
878 }
879
880 void showClusterBestSpider(PopulationPtr populationPtr, double **dataset,
881                            int row, int column){
882     printf("----------------------------\n");
883     printf("Cluster Generated by Best Spider: \n");
884     for(int i = 0; i < populationPtr->numClusters; i++){
885         printf("Cluster %d: ", i);
886         for(int j = 0; j < populationPtr->spiders[populationPtr->
887             indexBest]->lengthDatasetClusters; j++){
888             if (i == populationPtr->spiders[populationPtr->indexBest]->
889                 datasetClusters[j]){
890                 printf("%d, ", j);
891             }
892         }
893         printf("\nCenter %d: (", i);
894         for(int j = 0; j < populationPtr->pointDimension; j++){
895             printf("%f; ", populationPtr->spiders[populationPtr->
896                     indexBest]->centers[i]->point[j]);
897         }
898         printf(") \n\n");
899     }
900 }
901
902 void showMetricBestSpider(PopulationPtr populationPtr, int world_rank){
903     printf("Core [%d], Best metric [%d]: %f\n", world_rank,
904             populationPtr->indexBest, populationPtr->spiders[populationPtr->
```

```
905             indexBest]->fitness);
906 }
907
908 void showAllResults(PopulationPtr populationPtr, int numberGenerations,
909                     double **dataset, int row, int column){
910     showParameters(populationPtr, numberGenerations, row);
911     showBestSpider(populationPtr);
912     showClusterBestSpider(populationPtr, dataset, row, column);
913 }
914
915 void freeMemoryAuxilarPopulation(PopulationPtr populationPtr,
916                                  AuxPopulationPtr auxPopulationPtr){
917     freeMemoryArrayClusterCenter(auxPopulationPtr->clusCentersPtr,
918                                  populationPtr->numClusters);
919     freeMemoryArrayDouble(auxPopulationPtr->males);
920     freeMemoryArrayClusterCenter(auxPopulationPtr->spidersWeightsPtr,
921                                  populationPtr->numClusters);
922     freeMemoryArrayInt(auxPopulationPtr->matingGroup);
923     freeMemoryArrayDouble(auxPopulationPtr->matingRoulette);
924     freeMemorySpider(auxPopulationPtr->spiderWeightedMeanPtr);
925     freeMemoryArrayDouble(auxPopulationPtr->replacementRoulette);
926 }
927
928 void freeMemoryAuxilarSpider(PopulationPtr populationPtr,
929     AuxSpiderPtr auxSpiderPtr){
930     freeMemoryArrayDouble(auxSpiderPtr->distances);
931     freeMemoryArrayInt(auxSpiderPtr->numPointsForCluster);
932     freeMemoryMatrixDouble(auxSpiderPtr->sum,
933                            populationPtr->numClusters);
934 }
```

B.2 Algoritmo paralelo *Social Spider Optimization*

- main_psso_exp.c: Este módulo contiene la versión paralela del algoritmo de optimización basado en el comportamiento social de arañas.

- topology.c: Este módulo contiene los métodos para realizar el proceso de migración entre los núcleos.

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "clusterCenter.h"
#include "spider.h"
#include "population.h"
#include "utils.h"
#include "topology.h"

int main(int argc, char *argv[]) {
    Population population;
    AuxPopulation auxPopulation;
    AuxSpider auxSpider;
    Policy policy;
    int *columns = NULL;
    int column = 0;
    double **dataset = NULL;
    double **population_generated = NULL;

    char *fileNameColumns = argv[1];
    int numberClusters = atoi(argv[2]);
    int row = atoi(argv[3]);
    char *fileNameRows = argv[4];
    char *fileNamePolicy = argv[5];
    char *fileNamePopulation = argv[6];
    int limitCallFitness = atoi(argv[7]);
    int seed = atoi(argv[8]);
    char *fileNameOutput = argv[9];

    int numberGenerations = 100 * 1000000;
    int populationSize = 100;

```

```
34      allocateMemoryReadColumn(&columns, &column, fileNameColumns);
35      allocateMemoryReadDataset(&dataset, row, column, columns,
36                                fileNameRows);
37      allocateMemoryReadPopulation(&population_generated, 24 * populationSize,
38                                   column * numberClusters, fileNamePopulation);
39      readPolicy(&policy, fileNamePolicy);
40      // showPolicyMigration(&policy);
41
42      int lengthPoint = policy.numberEmiImm * numberClusters * column;
43      double *point = NULL;
44      allocateMemoryArrayDouble(&point, lengthPoint);
45
46      // Initialize the MPI environment
47      double t1, t2;
48      t1 = MPI_Wtime();
49
50      int world_rank;
51      int world_size;
52
53      MPI_Init(NULL, NULL);
54      MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
55      MPI_Comm_size(MPI_COMM_WORLD, &world_size);
56
57      if (world_size != 24) {
58          printf("Quitting. Number of MPI tasks no is 24.\n");
59          // MPI_Abort(MPI_COMM_WORLD, rc);
60          exit(0);
61      }
62
63      generateSeed(seed + world_rank + 1);
64      readInitialPopulation(&population, populationSize, numberClusters,
65                            dataset, row, column,
66                            policy.numberEmiImm * world_size,
67                            &auxPopulation, &auxSpider, population_generated,
68                            world_rank * populationSize, limitCallFitness);
69      calculateWeightPopulation(&population, dataset, row, column, &auxSpider);
```

```
70
71      char fileName[256];
72      sprintf(fileName, "%s", fileNameOutput);
73      FILE *fp = fopen(fileName, "a");
74
75      double best_metric = 0;
76      double best_metric_aux = 0;
77      int count_fitness = 0;
78      int sum_count_fitness = 0;
79
80      for (int k = 2; k <= numberGenerations; k++) {
81          // send and received count_fitness and sum this values
82          if (world_rank == 0) {
83              sum_count_fitness = population.countFitness;
84              for (int i = 1; i < world_size; i++) {
85                  MPI_Recv(&count_fitness, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
86                                  MPI_STATUS_IGNORE);
87                  sum_count_fitness += count_fitness;
88              }
89
90              // send sum_count_fitness to other cores
91              for (int i = 1; i < world_size; i++) {
92                  MPI_Send(&sum_count_fitness, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
93              }
94
95              if (sum_count_fitness >= limitCallFitness) {
96                  // printf("End core %d: %d\n", world_rank, sum_count_fitness);
97                  break;
98              }
99          } else {
100             count_fitness = population.countFitness;
101             MPI_Send(&count_fitness, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
102
103             MPI_Recv(&sum_count_fitness, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
104                             MPI_STATUS_IGNORE);
105             if (sum_count_fitness >= limitCallFitness) {
```

```
106                 break;
107             }
108         }
109
110         // Rank 0 receive best metric from others rank
111         if (world_rank == 0) {
112             best_metric = population.spiders[population.indexBest]->fitness;
113             for (int k = 1; k < world_size; k++) {
114                 MPI_Recv(&best_metric_aux, 1, MPI_DOUBLE, k, 0,
115                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
116                 if (best_metric_aux >= best_metric) {
117                     best_metric = best_metric_aux;
118                 }
119             }
120             // get metric and # evaluation fitness value for
121             // generation (convergence)
122             /* printf("[%d,%f,%d]\n", k - 1, best_metric, sum_count_fitness);
123             fprintf(fp, "%d,%f,%d\n", k - 1, best_metric, sum_count_fitness); */
124         } else {
125             best_metric_aux = population.spiders[population.indexBest]->fitness;
126             MPI_Send(&best_metric_aux, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
127         }
128
129         femaleCooperativeOperator(&population, &auxPopulation);
130         maleCooperativeOperator(&population, dataset, row, column,
131                                 &auxPopulation);
132         matingOperator(&population, dataset, row, column, &auxPopulation,
133                        &auxSpider);
134         replacement(&population, &auxPopulation);
135         calculateWeightPopulation(&population, dataset, row, column,
136                                   &auxSpider);
137
138         MPI_Barrier(MPI_COMM_WORLD);
139         runMigration(MPI_COMM_WORLD, world_rank, &population, point,
140                        lengthPoint, &policy, k);
141         MPI_Barrier(MPI_COMM_WORLD);
```

```
142
143         // showPopulation(&population);
144         // showBestFitness(&population, i);
145         // showBestSpider(&population);
146     }
147
148     MPI_Barrier(MPI_COMM_WORLD);
149     // showAllResults(&population, numberGenerations, dataset, row, column);
150     // showMetricBestSpider(&population, world_rank);
151     // showBestSpider(&population);
152
153     // Rank 0 receive best metric from others rank
154     // int best_rank;
155     if (world_rank == 0) {
156         best_metric = population.spiders[population.indexBest]->fitness;
157         // best_rank = 0;
158         for (int k = 1; k < world_size; k++) {
159             MPI_Recv(&best_metric_aux, 1, MPI_DOUBLE, k, 0,
160                         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
161             if (best_metric_aux >= best_metric) {
162                 best_metric = best_metric_aux;
163                 // best_rank = k;
164             }
165         }
166         t2 = MPI_Wtime();
167
168         printf("[%f,%f]\n", best_metric, t2 - t1);
169         fprintf(fp, "%f,%f\n", best_metric, t2 - t1);
170         /* char fileName[256];
171         sprintf(fileName, "%s", "output/aux_psso.out");
172         FILE *fp = fopen(fileName, "a");
173         fprintf(fp, "%f,%f\n", best_metric, t2 - t1);
174         fclose(fp); */
175         fclose(fp);
176     } else {
177         best_metric_aux = population.spiders[population.indexBest]->fitness;
```

```
178             MPI_Send(&best_metric_aux, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
179         }
180         MPI_Barrier(MPI_COMM_WORLD);
181 
182         freeMemoryReadColumn(columns);
183         freeMemoryReadDataset(dataset, row);
184         freeMemoryReadDataset(population_generated, 24 * populationSize);
185         freeMemoryArrayDouble(point);
186         freeMemoryPopulation(&population, &auxPopulation, &auxSpider);
187 
188         MPI_Finalize();
189         return 0;
190     }
```

```
1   #include "topology.h"
2   
3   void package(PopulationPtr populationPtr, double *point,
4                   PolicyPtr policyPtr, int source, int orderNode){
5       int index = 0;
6       switch (policyPtr->choiceEmi) {
7           case BEST:
8               if (policyPtr->emigration == REMOVE) {
9                   for (int k = 0; k < policyPtr->numberEmiImm; k++) {
10                      index = populationPtr->fitness[(orderNode - 1) *
11                              policyPtr->numberEmiImm + k]->key;
12                      populationPtr->indexMigration[(orderNode - 1) *
13                      policyPtr->numberEmiImm + k] = index;
14                      for(int i = 0; i < populationPtr->numClusters; i++){
15                          for(int j = 0; j < populationPtr->
16                              pointDimension; j++){
17                              point[(k * populationPtr->numClusters + i) *
18                              populationPtr->pointDimension + j] =
19                              populationPtr->spiders[index]->centers[i]->point[j];
20                          }
21                      }
22                  }
```

```
23              } else {    // CLONE
24                  for (int k = 0; k < policyPtr->numberEmiImm; k++) {
25                      index = populationPtr->fitness[k]->key;
26                      for(int i = 0; i < populationPtr->numClusters; i++){
27                          for(int j = 0; j < populationPtr->pointDimension; j++){
28                              point[(k * populationPtr->numClusters + i) *
29                              populationPtr->pointDimension + j] =
30                              populationPtr->spiders[index]->centers[i]->point[j];
31                          }
32                      }
33                  }
34              }
35          break;
36          case WORST:
37              if (policyPtr->emigration == REMOVE) {
38                  for (int k = 0; k < policyPtr->numberEmiImm; k++) {
39                      index = populationPtr->fitness[populationPtr->
40                              lengthSpider - 1 - ((orderNode - 1) *
41                              policyPtr->numberEmiImm + k)]->key;
42                      populationPtr->indexMigration[(orderNode - 1) *
43                      policyPtr->numberEmiImm + k] = index;
44                      for(int i = 0; i < populationPtr->numClusters; i++){
45                          for(int j = 0; j < populationPtr->pointDimension; j++){
46                              point[(k * populationPtr->numClusters + i) *
47                              populationPtr->pointDimension + j] =
48                              populationPtr->spiders[index]->centers[i]->point[j];
49                          }
50                      }
51                  }
52              } else {    // CLONE
53                  for (int k = 0; k < policyPtr->numberEmiImm; k++) {
54                      index = populationPtr->fitness[populationPtr->
55                              lengthSpider - 1 - k]->key;
56                      for(int i = 0; i < populationPtr->numClusters; i++){
57                          for(int j = 0; j < populationPtr->pointDimension; j++){
58                              point[(k * populationPtr->numClusters + i) *
```

```
59                              populationPtr->pointDimension + j] =
60                              populationPtr->spiders[index]->centers[i]->point[j];
61                          }
62                      }
63                  }
64              }
65          break;
66          case RANDOM:
67              if (policyPtr->emigration == REMOVE) {
68                  for (int k = 0; k < policyPtr->numberEmiImm; k++) {
69                      index = randomInt(populationPtr->lengthSpider);
70                      populationPtr->indexMigration[(orderNode - 1) *
71                      policyPtr->numberEmiImm + k] = index;
72                      for(int i = 0; i < populationPtr->numClusters; i++){
73                          for(int j = 0; j < populationPtr->pointDimension; j++){
74                              point[(k * populationPtr->numClusters + i) *
75                              populationPtr->pointDimension + j] =
76                              populationPtr->spiders[index]->centers[i]->point[j];
77                          }
78                      }
79                  }
80              } else {    // CLONE
81                  for (int k = 0; k < policyPtr->numberEmiImm; k++) {
82                      index = randomInt(populationPtr->lengthSpider);
83                      for(int i = 0; i < populationPtr->numClusters; i++){
84                          for(int j = 0; j < populationPtr->pointDimension; j++){
85                              point[(k * populationPtr->numClusters + i) *
86                              populationPtr->pointDimension + j] =
87                              populationPtr->spiders[index]->centers[i]->point[j];
88                          }
89                      }
90                  }
91              }
92          break;
93      }
94      // showArrayDoubleSend(populationPtr, policyPtr, point, source);
```

```
95  }
96
97  void unpack(PopulationPtr populationPtr, double *point,
98              PolicyPtr policyPtr, int destination, int source, int orderNode){
99      int index = 0;
100     switch (policyPtr->choiceImm) {
101         case BEST:
102             if (policyPtr->emigration == REMOVE) {
103                 for (int k = 0; k < policyPtr->numberEmiImm; k++) {
104                     index = populationPtr->indexMigration[(orderNode - 1) *
105                             policyPtr->numberEmiImm + k];
106                     for(int i = 0; i < populationPtr->numClusters; i++){
107                         for(int j = 0; j < populationPtr->pointDimension; j++){
108                             populationPtr->spiders[index]->centers[i]->
109                             point[j] = point[(k * populationPtr->
110                                         numClusters + i) * populationPtr->
111                                         pointDimension + j];
112                         }
113                     }
114                 }
115             } else {    // REPLACE
116                 for (int k = 0; k < policyPtr->numberEmiImm; k++) {
117                     index = populationPtr->fitness[(orderNode - 1) *
118                             policyPtr->numberEmiImm + k]->key;
119                     for(int i = 0; i < populationPtr->numClusters; i++){
120                         for(int j = 0; j < populationPtr->pointDimension; j++){
121                             populationPtr->spiders[index]->centers[i]->
122                             point[j] = point[(k * populationPtr->
123                                         numClusters + i) * populationPtr->
124                                         pointDimension + j];
125                         }
126                     }
127                 }
128             }
129         break;
130         case WORST:
```

```
131             if (policyPtr->emigration == REMOVE) {
132                 for (int k = 0; k < policyPtr->numberEmiImm; k++) {
133                     index = populationPtr->indexMigration[(orderNode - 1) *
134                             policyPtr->numberEmiImm + k];
135                     for(int i = 0; i < populationPtr->numClusters; i++){
136                         for(int j = 0; j < populationPtr->pointDimension; j++){
137                             populationPtr->spiders[index]->centers[i]->
138                                 point[j] = point[(k * populationPtr->
139                                             numClusters + i) * populationPtr->
140                                             pointDimension + j];
141                         }
142                     }
143                 }
144             } else {    // REPLACE
145                 for(int k = 0; k < policyPtr->numberEmiImm; k++){
146                     index = populationPtr->fitness[populationPtr->
147                             lengthSpider - 1 - ((orderNode - 1) *
148                             policyPtr->numberEmiImm + k)]->key;
149                     for(int i = 0; i < populationPtr->numClusters; i++){
150                         for(int j = 0; j < populationPtr->pointDimension; j++){
151                             populationPtr->spiders[index]->centers[i]->
152                             point[j] = point[(k * populationPtr->
153                                         numClusters + i) * populationPtr->
154                                         pointDimension + j];
155                         }
156                     }
157                 }
158             }
159         break;
160         case RANDOM:
161             if (policyPtr->emigration == REMOVE) {
162                 for (int k = 0; k < policyPtr->numberEmiImm; k++) {
163                     index = populationPtr->indexMigration[(orderNode - 1) *
164                             policyPtr->numberEmiImm + k];
165                     for(int i = 0; i < populationPtr->numClusters; i++){
166                         for(int j = 0; j < populationPtr->pointDimension; j++){
```

```
                            populationPtr->spiders[index]->centers[i]->
                            point[j] = point[(k * populationPtr->
                                        numClusters + i) * populationPtr->
                                        pointDimension + j];
                        }
                    }
                }
            } else {    // REPLACE
                for(int k = 0; k < policyPtr->numberEmiImm; k++){
                    index = randomInt(populationPtr->lengthSpider);
                    for(int i = 0; i < populationPtr->numClusters; i++){
                        for(int j = 0; j < populationPtr->pointDimension; j++){
                            populationPtr->spiders[index]->centers[i]->
                            point[j] = point[(k * populationPtr->
                                        numClusters + i) * populationPtr->
                                        pointDimension + j];
                        }
                    }
                }
            }
        break;
    }
    // showArrayDoubleReceived(populationPtr, policyPtr, point,
                                destination, source, orderNode);
}

void ringTopology(MPI_Comm communicator, int world_rank,
                    PopulationPtr populationPtr, double *point,
                    int lengthPoint, PolicyPtr policyPtr) {
    MPI_Status status;
    int world_size;
    MPI_Comm_size(communicator, &world_size);

    package(populationPtr, point, policyPtr, world_rank, 1);
    MPI_Send(point, lengthPoint, MPI_FLOAT,
            (world_rank + 1) % world_size, 0, communicator);
```

```
203     MPI_Barrier(communicator);
204     MPI_Recv(point, lengthPoint, MPI_FLOAT,
205             (world_rank + world_size - 1) % world_size, 0, communicator, &status);
206     unpack(populationPtr, point, policyPtr, world_rank, status.MPI_SOURCE, 1);
207 }
208
209 void treeTopology(MPI_Comm communicator, int world_rank,
210                     PopulationPtr populationPtr, double *point,
211                     int lengthPoint, PolicyPtr policyPtr) {
212     MPI_Status status;
213     int world_size;
214     MPI_Comm_size(communicator, &world_size);
215
216     if(world_rank == 0){
217         package(populationPtr, point, policyPtr, world_rank, 1);
218         MPI_Send(point, lengthPoint, MPI_FLOAT, (2 * world_rank) + 1,
219                     0, communicator);
220         package(populationPtr, point, policyPtr, world_rank, 2);
221         MPI_Send(point, lengthPoint, MPI_FLOAT, (2 * world_rank) + 2,
222                     0, communicator);
223         MPI_Barrier(communicator);
224         MPI_Recv(point, lengthPoint, MPI_FLOAT, (2 * world_rank) + 1,
225                     0, communicator, &status);
226         unpack(populationPtr, point, policyPtr, world_rank,
227                 status.MPI_SOURCE, 1);
228         MPI_Recv(point, lengthPoint, MPI_FLOAT, (2 * world_rank) + 2, 0,
229                     communicator, &status);
230         unpack(populationPtr, point, policyPtr, world_rank,
231                 status.MPI_SOURCE, 2);
232     }else {
233         if(world_rank >= 1 && world_rank <= 6){
234             package(populationPtr, point, policyPtr, world_rank, 1);
235             MPI_Send(point, lengthPoint, MPI_FLOAT, (world_rank - 1) / 2,
236                         0, communicator);
237             package(populationPtr, point, policyPtr, world_rank, 2);
238             MPI_Send(point, lengthPoint, MPI_FLOAT, (2 * world_rank) + 1,
```

```
239                     0, communicator);
240             package(populationPtr, point, policyPtr, world_rank, 3);
241             MPI_Send(point, lengthPoint, MPI_FLOAT, (2 * world_rank) + 2,
242                     0, communicator);
243             MPI_Barrier(communicator);
244             MPI_Recv(point, lengthPoint, MPI_FLOAT, (world_rank - 1) / 2,
245                     0, communicator, &status);
246             unpack(populationPtr, point, policyPtr, world_rank,
247                     status.MPI_SOURCE, 1);
248             MPI_Recv(point, lengthPoint, MPI_FLOAT, (2 * world_rank) + 1,
249                     0, communicator, &status);
250             unpack(populationPtr, point, policyPtr, world_rank,
251                     status.MPI_SOURCE, 2);
252             MPI_Recv(point, lengthPoint, MPI_FLOAT, (2 * world_rank) + 2,
253                     0, communicator, &status);
254             unpack(populationPtr, point, policyPtr, world_rank,
255                     status.MPI_SOURCE, 3);
256         }else {
257             if(world_rank >= 7 && world_rank <= 15){
258                 package(populationPtr, point, policyPtr, world_rank, 1);
259                 MPI_Send(point, lengthPoint, MPI_FLOAT, (world_rank - 1) / 2,
260                         0, communicator);
261                 package(populationPtr, point, policyPtr, world_rank, 2);
262                 MPI_Send(point, lengthPoint, MPI_FLOAT, world_rank + 8,
263                     0, communicator);
264                 MPI_Barrier(communicator);
265                 MPI_Recv(point, lengthPoint, MPI_FLOAT, (world_rank - 1) / 2,
266                         0, communicator, &status);
267                 unpack(populationPtr, point, policyPtr, world_rank,
268                         status.MPI_SOURCE, 1);
269                 MPI_Recv(point, lengthPoint, MPI_FLOAT, world_rank + 8,
270                         0, communicator, &status);
271                 unpack(populationPtr, point, policyPtr, world_rank,
272                         status.MPI_SOURCE, 2);
273             }else {
274                 package(populationPtr, point, policyPtr, world_rank, 1);
```

```
275                 MPI_Send(point, lengthPoint, MPI_FLOAT, world_rank - 8,
276                             0, communicator);
277                 MPI_Barrier(communicator);
278                 MPI_Recv(point, lengthPoint, MPI_FLOAT, world_rank - 8,
279                             0, communicator, &status);
280                 unpack(populationPtr, point, policyPtr, world_rank,
281                         status.MPI_SOURCE, 1);
282             }
283         }
284     }
285 }
286
287 void netATopology(MPI_Comm communicator, int world_rank,
288                     PopulationPtr populationPtr, double *point,
289                     int lengthPoint, PolicyPtr policyPtr) {
290     MPI_Status status;
291     int world_size;
292     MPI_Comm_size(communicator, &world_size);
293
294     if (world_rank == 1 || world_rank == 2) {
295         package(populationPtr, point, policyPtr, world_rank, 1);
296         MPI_Send(point, lengthPoint, MPI_FLOAT,
297                 (world_rank - 1 + 24) % world_size, 0, communicator);
298         package(populationPtr, point, policyPtr, world_rank, 2);
299         MPI_Send(point, lengthPoint, MPI_FLOAT,
300                 (world_rank + 1) % world_size, 0, communicator);
301         package(populationPtr, point, policyPtr, world_rank, 3);
302         MPI_Send(point, lengthPoint, MPI_FLOAT,
303                 (world_rank + 4) % world_size, 0, communicator);
304         MPI_Barrier(communicator);
305         MPI_Recv(point, lengthPoint, MPI_FLOAT,
306                 (world_rank -1 + 24) % world_size, 0,
307                 communicator, &status);
308         unpack(populationPtr, point, policyPtr, world_rank,
309                 status.MPI_SOURCE, 1);
310         MPI_Recv(point, lengthPoint, MPI_FLOAT,
```

```
311                 (world_rank + 1) % world_size, 0, communicator, &status);
312         unpack(populationPtr, point, policyPtr, world_rank,
313                 status.MPI_SOURCE, 2);
314         MPI_Recv(point, lengthPoint, MPI_FLOAT,
315                 (world_rank + 4) % world_size, 0, communicator, &status);
316         unpack(populationPtr, point, policyPtr, world_rank,
317                 status.MPI_SOURCE, 3);
318     } else {
319         if (world_rank == 21 || world_rank == 22) {
320             package(populationPtr, point, policyPtr, world_rank, 1);
321             MPI_Send(point, lengthPoint, MPI_FLOAT,
322                     (world_rank - 4 + 24) % world_size, 0, communicator);
323             package(populationPtr, point, policyPtr, world_rank, 2);
324             MPI_Send(point, lengthPoint, MPI_FLOAT,
325                     (world_rank -1 + 24) % world_size, 0, communicator);
326             package(populationPtr, point, policyPtr, world_rank, 3);
327             MPI_Send(point, lengthPoint, MPI_FLOAT,
328                     (world_rank + 1) % world_size, 0, communicator);
329             MPI_Barrier(communicator);
330             MPI_Recv(point, lengthPoint, MPI_FLOAT,
331                     (world_rank -4 + 24) % world_size, 0,
332                     communicator, &status);
333             unpack(populationPtr, point, policyPtr,
334                     world_rank, status.MPI_SOURCE, 1);
335             MPI_Recv(point, lengthPoint, MPI_FLOAT,
336                     (world_rank - 1 + 24) % world_size, 0,
337                     communicator, &status);
338             unpack(populationPtr, point, policyPtr, world_rank,
339                     status.MPI_SOURCE, 2);
340             MPI_Recv(point, lengthPoint, MPI_FLOAT,
341                     (world_rank + 1) % world_size, 0, communicator, &status);
342             unpack(populationPtr, point, policyPtr, world_rank,
343                     status.MPI_SOURCE, 3);
344         } else {
345             if (world_rank % 4 == 0) {
346                 package(populationPtr, point, policyPtr, world_rank, 1);
```

```
347                 MPI_Send(point, lengthPoint, MPI_FLOAT,
348                         (world_rank - 4 + 24) % world_size, 0, communicator);
349                 package(populationPtr, point, policyPtr, world_rank, 2);
350                 MPI_Send(point, lengthPoint, MPI_FLOAT,
351                         (world_rank + 1) % world_size, 0, communicator);
352                 package(populationPtr, point, policyPtr, world_rank, 3);
353                 MPI_Send(point, lengthPoint, MPI_FLOAT,
354                         (world_rank + 4) % world_size, 0, communicator);
355                 MPI_Barrier(communicator);
356                 MPI_Recv(point, lengthPoint, MPI_FLOAT,
357                         (world_rank - 4 + 24) % world_size, 0,
358                         communicator, &status);
359                 unpack(populationPtr, point, policyPtr, world_rank,
360                         status.MPI_SOURCE, 1);
361                 MPI_Recv(point, lengthPoint, MPI_FLOAT,
362                         (world_rank + 1) % world_size, 0, communicator,
363                         &status);
364                 unpack(populationPtr, point, policyPtr, world_rank,
365                         status.MPI_SOURCE, 2);
366                 MPI_Recv(point, lengthPoint, MPI_FLOAT,
367                         (world_rank + 4) % world_size, 0, communicator,
368                         &status);
369                 unpack(populationPtr, point, policyPtr, world_rank,
370                         status.MPI_SOURCE, 3);
371             } else {
372                 if ((world_rank + 1) % 4 == 0) {
373                     package(populationPtr, point, policyPtr, world_rank, 1);
374                     MPI_Send(point, lengthPoint, MPI_FLOAT,
375                             (world_rank - 4 + 24) % world_size, 0,
376                             communicator);
377                     package(populationPtr, point, policyPtr, world_rank, 2);
378                     MPI_Send(point, lengthPoint, MPI_FLOAT,
379                             (world_rank - 1 + 24) % world_size, 0,
380                             communicator);
381                     package(populationPtr, point, policyPtr, world_rank, 3);
382                     MPI_Send(point, lengthPoint, MPI_FLOAT,
```

```
383                         (world_rank  + 4) % world_size, 0, communicator);
384                 MPI_Barrier(communicator);
385                 MPI_Recv(point, lengthPoint, MPI_FLOAT,
386                         (world_rank - 4 + 24) % world_size, 0,
387                         communicator, &status);
388                 unpack(populationPtr, point, policyPtr, world_rank,
389                         status.MPI_SOURCE, 1);
390                 MPI_Recv(point, lengthPoint, MPI_FLOAT,
391                         (world_rank - 1 + 24) % world_size, 0,
392                         communicator, &status);
393                 unpack(populationPtr, point, policyPtr, world_rank,
394                         status.MPI_SOURCE, 2);
395                 MPI_Recv(point, lengthPoint, MPI_FLOAT,
396                         (world_rank + 4) % world_size, 0, communicator,
397                         &status);
398                 unpack(populationPtr, point, policyPtr, world_rank,
399                         status.MPI_SOURCE, 3);
400             } else {
401                 package(populationPtr, point, policyPtr, world_rank, 1);
402                 MPI_Send(point, lengthPoint, MPI_FLOAT,
403                         (world_rank - 4 + 24) % world_size, 0,
404                         communicator);
405                 package(populationPtr, point, policyPtr, world_rank, 2);
406                 MPI_Send(point, lengthPoint, MPI_FLOAT,
407                         (world_rank - 1 + 24) % world_size, 0,
408                         communicator);
409                 package(populationPtr, point, policyPtr, world_rank, 3);
410                 MPI_Send(point, lengthPoint, MPI_FLOAT,
411                         (world_rank + 1) % world_size, 0, communicator);
412                 package(populationPtr, point, policyPtr, world_rank, 4);
413                 MPI_Send(point, lengthPoint, MPI_FLOAT,
414                         (world_rank + 4) % world_size, 0, communicator);
415                 MPI_Barrier(communicator);
416                 MPI_Recv(point, lengthPoint, MPI_FLOAT,
417                         (world_rank - 4 + 24) % world_size, 0,
418                         communicator, &status);
```

```
419                     unpack(populationPtr, point, policyPtr, world_rank,
420                             status.MPI_SOURCE, 1);
421                     MPI_Recv(point, lengthPoint, MPI_FLOAT,
422                             (world_rank - 1 + 24) % world_size, 0,
423                             communicator, &status);
424                     unpack(populationPtr, point, policyPtr, world_rank,
425                             status.MPI_SOURCE, 2);
426                     MPI_Recv(point, lengthPoint, MPI_FLOAT,
427                             (world_rank + 1) % world_size, 0, communicator,
428                             &status);
429                     unpack(populationPtr, point, policyPtr, world_rank,
430                             status.MPI_SOURCE, 3);
431                     MPI_Recv(point, lengthPoint, MPI_FLOAT,
432                             (world_rank + 4) % world_size, 0, communicator,
433                             &status);
434                     unpack(populationPtr, point, policyPtr, world_rank,
435                             status.MPI_SOURCE, 4);
436                 }
437             }
438         }
439     }
440 }
441
442 void netBTopology(MPI_Comm communicator, int world_rank,
443                     PopulationPtr populationPtr, double *point,
444                     int lengthPoint, PolicyPtr policyPtr) {
445     MPI_Status status;
446     int world_size;
447     MPI_Comm_size(communicator, &world_size);
448
449     if (world_rank == 1 || world_rank == 2) {
450         package(populationPtr, point, policyPtr, world_rank, 1);
451         MPI_Send(point, lengthPoint, MPI_FLOAT,
452                 (world_rank - 1 + 24) % world_size, 0, communicator);
453         package(populationPtr, point, policyPtr, world_rank, 2);
454         MPI_Send(point, lengthPoint, MPI_FLOAT,
```

```
455                 (world_rank + 1) % world_size, 0, communicator);
456         package(populationPtr, point, policyPtr, world_rank, 3);
457         MPI_Send(point, lengthPoint, MPI_FLOAT,
458                 (world_rank + 4) % world_size, 0, communicator);
459         MPI_Barrier(communicator);
460         MPI_Recv(point, lengthPoint, MPI_FLOAT,
461                 (world_rank -1 + 24) % world_size, 0,
462                 communicator, &status);
463         unpack(populationPtr, point, policyPtr, world_rank,
464                 status.MPI_SOURCE, 1);
465         MPI_Recv(point, lengthPoint, MPI_FLOAT,
466                 (world_rank + 1) % world_size, 0,
467                 communicator, &status);
468         unpack(populationPtr, point, policyPtr, world_rank,
469                 status.MPI_SOURCE, 2);
470         MPI_Recv(point, lengthPoint, MPI_FLOAT,
471                 (world_rank + 4) % world_size, 0,
472                 communicator, &status);
473         unpack(populationPtr, point, policyPtr, world_rank,
474                 status.MPI_SOURCE, 3);
475     } else {
476         if (world_rank == 21 || world_rank == 22) {
477             package(populationPtr, point, policyPtr, world_rank, 1);
478             MPI_Send(point, lengthPoint, MPI_FLOAT,
479                     (world_rank - 4 + 24) % world_size, 0, communicator);
480             package(populationPtr, point, policyPtr, world_rank, 2);
481             MPI_Send(point, lengthPoint, MPI_FLOAT,
482                     (world_rank -1 + 24) % world_size, 0, communicator);
483             package(populationPtr, point, policyPtr, world_rank, 3);
484             MPI_Send(point, lengthPoint, MPI_FLOAT,
485                     (world_rank + 1) % world_size, 0, communicator);
486             MPI_Barrier(communicator);
487             MPI_Recv(point, lengthPoint, MPI_FLOAT,
488                     (world_rank - 4 + 24) % world_size, 0,
489                     communicator, &status);
490             unpack(populationPtr, point, policyPtr, world_rank,
```

```
491                     status.MPI_SOURCE, 1);
492             MPI_Recv(point, lengthPoint, MPI_FLOAT,
493                     (world_rank - 1 + 24) % world_size, 0,
494                     communicator, &status);
495             unpack(populationPtr, point, policyPtr, world_rank,
496                     status.MPI_SOURCE, 2);
497             MPI_Recv(point, lengthPoint, MPI_FLOAT,
498                     (world_rank + 1) % world_size, 0,
499                     communicator, &status);
500             unpack(populationPtr, point, policyPtr, world_rank,
501                     status.MPI_SOURCE, 3);
502         } else {
503             if (world_rank % 6 == 0 || world_rank % 4 == 0) {
504                 package(populationPtr, point, policyPtr, world_rank, 1);
505                 MPI_Send(point, lengthPoint, MPI_FLOAT,
506                         (world_rank - 4 + 24) % world_size, 0, communicator);
507                 package(populationPtr, point, policyPtr, world_rank, 2);
508                 MPI_Send(point, lengthPoint, MPI_FLOAT,
509                         (world_rank + 1) % world_size, 0, communicator);
510                 package(populationPtr, point, policyPtr, world_rank, 3);
511                 MPI_Send(point, lengthPoint, MPI_FLOAT,
512                         (world_rank + 4) % world_size, 0, communicator);
513                 MPI_Barrier(communicator);
514                 MPI_Recv(point, lengthPoint, MPI_FLOAT,
515                         (world_rank - 4 + 24) % world_size, 0,
516                         communicator, &status);
517                 unpack(populationPtr, point, policyPtr, world_rank,
518                         status.MPI_SOURCE, 1);
519                 MPI_Recv(point, lengthPoint, MPI_FLOAT,
520                         (world_rank + 1) % world_size, 0, communicator,
521                         &status);
522                 unpack(populationPtr, point, policyPtr, world_rank,
523                         status.MPI_SOURCE, 2);
524                 MPI_Recv(point, lengthPoint, MPI_FLOAT,
525                         (world_rank + 4) % world_size, 0, communicator,
526                         &status);
```

```
527                 unpack(populationPtr, point, policyPtr, world_rank,
528                         status.MPI_SOURCE, 3);
529             } else {
530                 if (world_rank == 9 || world_rank == 10 ||
531                     world_rank == 13 || world_rank == 14) {
532                     package(populationPtr, point, policyPtr, world_rank, 1);
533                     MPI_Send(point, lengthPoint, MPI_FLOAT,
534                             (world_rank - 4 + 24) % world_size, 0,
535                             communicator);
536                     package(populationPtr, point, policyPtr, world_rank, 2);
537                     MPI_Send(point, lengthPoint, MPI_FLOAT,
538                             (world_rank - 1 + 24) % world_size, 0,
539                             communicator);
540                     package(populationPtr, point, policyPtr, world_rank, 3);
541                     MPI_Send(point, lengthPoint, MPI_FLOAT,
542                             (world_rank + 1) % world_size, 0, communicator);
543                     package(populationPtr, point, policyPtr, world_rank, 4);
544                     MPI_Send(point, lengthPoint, MPI_FLOAT,
545                             (world_rank + 4) % world_size, 0, communicator);
546                     MPI_Barrier(communicator);
547                     MPI_Recv(point, lengthPoint, MPI_FLOAT,
548                             (world_rank - 4 + 24) % world_size, 0,
549                             communicator, &status);
550                     unpack(populationPtr, point, policyPtr, world_rank,
551                             status.MPI_SOURCE, 1);
552                     MPI_Recv(point, lengthPoint, MPI_FLOAT,
553                             (world_rank - 1 + 24) % world_size, 0,
554                             communicator, &status);
555                     unpack(populationPtr, point, policyPtr, world_rank,
556                             status.MPI_SOURCE, 2);
557                     MPI_Recv(point, lengthPoint, MPI_FLOAT,
558                             (world_rank + 1) % world_size, 0, communicator,
559                             &status);
560                     unpack(populationPtr, point, policyPtr, world_rank,
561                             status.MPI_SOURCE, 3);
562                     MPI_Recv(point, lengthPoint, MPI_FLOAT,
```

```
563                             (world_rank + 4) % world_size, 0, communicator,
564                             &status);
565                     unpack(populationPtr, point, policyPtr, world_rank,
566                             status.MPI_SOURCE, 4);
567                 } else {
568                     package(populationPtr, point, policyPtr, world_rank, 1);
569                     MPI_Send(point, lengthPoint, MPI_FLOAT,
570                             (world_rank - 4 + 24) % world_size, 0,
571                             communicator);
572                     package(populationPtr, point, policyPtr, world_rank, 2);
573                     MPI_Send(point, lengthPoint, MPI_FLOAT,
574                             (world_rank - 1 + 24) % world_size, 0,
575                             communicator);
576                     package(populationPtr, point, policyPtr, world_rank, 3);
577                     MPI_Send(point, lengthPoint, MPI_FLOAT,
578                             (world_rank  + 4) % world_size, 0, communicator);
579                     MPI_Barrier(communicator);
580                     MPI_Recv(point, lengthPoint, MPI_FLOAT,
581                             (world_rank -4 + 24) % world_size, 0,
582                             communicator, &status);
583                     unpack(populationPtr, point, policyPtr, world_rank,
584                             status.MPI_SOURCE, 1);
585                     MPI_Recv(point, lengthPoint, MPI_FLOAT,
586                             (world_rank - 1 + 24) % world_size, 0,
587                             communicator, &status);
588                     unpack(populationPtr, point, policyPtr, world_rank,
589                             status.MPI_SOURCE, 2);
590                     MPI_Recv(point, lengthPoint, MPI_FLOAT,
591                             (world_rank + 4) % world_size, 0, communicator,
592                             &status);
593                     unpack(populationPtr, point, policyPtr, world_rank,
594                             status.MPI_SOURCE, 3);
595                 }
596             }
597         }
598     }
```

```
599 }
600
601 void torusTopology(MPI_Comm communicator, int world_rank,
602                    PopulationPtr populationPtr, double *point,
603                    int lengthPoint, PolicyPtr policyPtr) {
604     MPI_Status status;
605     int world_size;
606     MPI_Comm_size(communicator, &world_size);
607     if (world_rank % 4 == 0){
608         package(populationPtr, point, policyPtr, world_rank, 1);
609         MPI_Send(point, lengthPoint, MPI_FLOAT,
610                 (world_rank - 4 + 24) % 24, 0, communicator);
611         package(populationPtr, point, policyPtr, world_rank, 2);
612         MPI_Send(point, lengthPoint, MPI_FLOAT,
613                 (world_rank + 1) % 24, 0, communicator);
614         package(populationPtr, point, policyPtr, world_rank, 3);
615         MPI_Send(point, lengthPoint, MPI_FLOAT,
616                 (world_rank + 3) % 24, 0, communicator);
617         package(populationPtr, point, policyPtr, world_rank, 4);
618         MPI_Send(point, lengthPoint, MPI_FLOAT,
619                 (world_rank + 4) % 24, 0, communicator);
620         MPI_Barrier(communicator);
621         MPI_Recv(point, lengthPoint, MPI_FLOAT,
622                 (world_rank - 4 + 24) % 24, 0, communicator, &status);
623         unpack(populationPtr, point, policyPtr, world_rank,
624                 status.MPI_SOURCE, 1);
625         MPI_Recv(point, lengthPoint, MPI_FLOAT,
626                 (world_rank + 1) % 24, 0, communicator, &status);
627         unpack(populationPtr, point, policyPtr, world_rank,
628                 status.MPI_SOURCE, 2);
629         MPI_Recv(point, lengthPoint, MPI_FLOAT,
630                 (world_rank + 3) % 24, 0, communicator, &status);
631         unpack(populationPtr, point, policyPtr, world_rank,
632                 status.MPI_SOURCE, 3);
633         MPI_Recv(point, lengthPoint, MPI_FLOAT,
634                 (world_rank + 4) % 24, 0, communicator, &status);
```

```
635         unpack(populationPtr, point, policyPtr, world_rank,
636                 status.MPI_SOURCE, 4);
637     } else {
638         if (world_rank % 4 == 3){
639             package(populationPtr, point, policyPtr, world_rank, 1);
640             MPI_Send(point, lengthPoint, MPI_FLOAT,
641                     (world_rank - 4 + 24) % 24, 0, communicator);
642             package(populationPtr, point, policyPtr, world_rank, 2);
643             MPI_Send(point, lengthPoint, MPI_FLOAT,
644                     (world_rank - 3 + 24) % 24, 0, communicator);
645             package(populationPtr, point, policyPtr, world_rank, 3);
646             MPI_Send(point, lengthPoint, MPI_FLOAT,
647                     (world_rank - 1 + 24) % 24, 0, communicator);
648             package(populationPtr, point, policyPtr, world_rank, 4);
649             MPI_Send(point, lengthPoint, MPI_FLOAT,
650                     (world_rank + 4) % 24, 0, communicator);
651             MPI_Barrier(communicator);
652             MPI_Recv(point, lengthPoint, MPI_FLOAT,
653                     (world_rank - 4 + 24) % 24, 0, communicator, &status);
654             unpack(populationPtr, point, policyPtr, world_rank,
655                     status.MPI_SOURCE, 1);
656             MPI_Recv(point, lengthPoint, MPI_FLOAT,
657                     (world_rank - 3 + 24) % 24, 0, communicator, &status);
658             unpack(populationPtr, point, policyPtr, world_rank,
659                     status.MPI_SOURCE, 2);
660             MPI_Recv(point, lengthPoint, MPI_FLOAT,
661                     (world_rank - 1 + 24) % 24, 0, communicator, &status);
662             unpack(populationPtr, point, policyPtr, world_rank,
663                     status.MPI_SOURCE, 3);
664             MPI_Recv(point, lengthPoint, MPI_FLOAT,
665                     (world_rank + 4) % 24, 0, communicator, &status);
666             unpack(populationPtr, point, policyPtr, world_rank,
667                     status.MPI_SOURCE, 4);
668         } else{
669             package(populationPtr, point, policyPtr, world_rank, 1);
670             MPI_Send(point, lengthPoint, MPI_FLOAT,
```

```
671                     (world_rank - 4 + 24) % 24, 0, communicator);
672             package(populationPtr, point, policyPtr, world_rank, 2);
673             MPI_Send(point, lengthPoint, MPI_FLOAT,
674                     (world_rank - 1 + 24) % 24, 0, communicator);
675             package(populationPtr, point, policyPtr, world_rank, 3);
676             MPI_Send(point, lengthPoint, MPI_FLOAT,
677                     (world_rank + 1) % 24, 0, communicator);
678             package(populationPtr, point, policyPtr, world_rank, 4);
679             MPI_Send(point, lengthPoint, MPI_FLOAT,
680                     (world_rank + 4) % 24, 0, communicator);
681             MPI_Barrier(communicator);
682             MPI_Recv(point, lengthPoint, MPI_FLOAT,
683                     (world_rank - 4 + 24) % 24, 0, communicator, &status);
684             unpack(populationPtr, point, policyPtr, world_rank,
685                     status.MPI_SOURCE, 1);
686             MPI_Recv(point, lengthPoint, MPI_FLOAT,
687                     (world_rank - 1 + 24) % 24, 0, communicator, &status);
688             unpack(populationPtr, point, policyPtr, world_rank,
689                     status.MPI_SOURCE, 2);
690             MPI_Recv(point, lengthPoint, MPI_FLOAT,
691                     (world_rank + 1) % 24, 0, communicator, &status);
692             unpack(populationPtr, point, policyPtr, world_rank,
693                     status.MPI_SOURCE, 3);
694             MPI_Recv(point, lengthPoint, MPI_FLOAT,
695                     (world_rank + 4) % 24, 0, communicator, &status);
696             unpack(populationPtr, point, policyPtr, world_rank,
697                     status.MPI_SOURCE, 4);
698         }
699     }
700 }
701
702 void graphTopology(MPI_Comm communicator, int world_rank,
703                     PopulationPtr populationPtr, double *point,
704                     int lengthPoint, PolicyPtr policyPtr) {
705     MPI_Status status;
706     int world_size;
```

```
707     MPI_Comm_size(communicator, &world_size);
708     // ayuda a ver que porción del arreglo de fitness (ordenado) se
709     // empaquetara para enviar, según en numberEmiImm
710     int orderNode = 1;
711     for (int i = 0; i < world_size; i++) {
712         if (i != world_rank) {
713             package(populationPtr, point, policyPtr, world_rank, orderNode);
714             MPI_Send(point, lengthPoint, MPI_FLOAT, i, 0, communicator);
715             orderNode++;
716         }
717     }
718     MPI_Barrier(communicator);
719     orderNode = 1;
720     for (int i = 0; i < world_size; i++) {
721         if (i != world_rank) {
722             MPI_Recv(point, lengthPoint, MPI_FLOAT, i, 0,
723                         communicator, &status);
724             unpack(populationPtr, point, policyPtr, world_rank,
725                     status.MPI_SOURCE, orderNode);
726             orderNode++;
727         }
728     }
729 }
730
731 void dinamicTopology(MPI_Comm communicator, int world_rank,
732                         PopulationPtr populationPtr, double *point,
733                         int lengthPoint, PolicyPtr policyPtr) {
734     MPI_Status status;
735     int world_size;
736     MPI_Comm_size(communicator, &world_size);
737
738     // Calculate ranking
739     double average_;
740     double standardDeviation_;
741
742     double *data = NULL;
```

```
743     allocateMemoryArrayDouble(&data, world_size);
744
745     // GOOD = 0, MEDIUM = 1, BAD = 2 (for RAND)
746     int qualification;
747     int qualification1;
748     int qualification2;
749
750     if (world_rank == 0) {
751         DictionaryPtr *averageList;
752         DictionaryPtr *standardDeviationList;
753         DictionaryPtr *ranking;
754
755         allocateMemoryArrayDictionary(&averageList, world_size);
756         allocateMemoryArrayDictionary(&standardDeviationList, world_size);
757         allocateMemoryArrayDictionary(&ranking, world_size);
758
759         average_ = average(populationPtr);
760         standardDeviation_ = standardDeviation(populationPtr);
761         averageList[world_rank]->key = world_rank;
762         averageList[world_rank]->value = average_;
763         standardDeviationList[world_rank]->key = world_rank;
764         standardDeviationList[world_rank]->value = standardDeviation_;
765
766         // received avarage and standart deviation from other cores
767         for (int k = 1; k < world_size; k++) {
768             MPI_Recv(data, 2, MPI_DOUBLE, k, 0, communicator, &status);
769             averageList[k]->key = k;
770             averageList[k]->value = data[0];
771             standardDeviationList[k]->key = k;
772             standardDeviationList[k]->value = data[1];
773         }
774
775         insertSort(averageList, world_size);
776         insertSort(standardDeviationList, world_size);
777
778         // generate array for ranking
```

```
779         for (int k = 0; k < world_size; k++) {
780             int key = averageList[k]->key;
781             ranking[key]->key = key;
782             ranking[key]->value += k + 1;
783             key = standardDeviationList[k]->key;
784             ranking[key]->key = key;
785             ranking[key]->value += world_size - k;
786         }
787         insertSort(ranking, world_size);
788 
789         // load ranking array to data array to send
790         for (int k = 0; k < world_size; k++) {
791             /* printf("%d %d %f\n", k, averageList[k]->key,
792                         averageList[k]->value);
793             printf("%d %d %f\n", k, standardDeviationList[k]->key,
794                     standardDeviationList[k]->value);
795             printf("%d %d %f\n", k, ranking[k]->key, ranking[k]->value); */
796             data[k] = ranking[k]->key;
797             // printf("%f\n", data[k]);
798         }
799         // printf("\n");
800         // re orden element for good, medium and bad usind random
801         // index on data array (ranking)
802         int index_rand;
803         float temp_data;
804         for (int i = 0; i < world_size / 8; i++) {
805             for (int j = 0; j < world_size / 3; j++) {
806                 temp_data = data[i * 8 + j];
807                 index_rand = randomInt(8) + i * 8;
808                 data[i * 8 + j] = data[index_rand];
809                 data[index_rand] = temp_data;
810             }
811         }
812 
813         /* for (int k = 0; k < world_size; k++) {
814             printf("%f\n", data[k]);
```

```
815         }
816         printf("---\n"); */
817 
818         // send data array (ranking) to other cores
819         for (int i = 1; i < world_size; i++) {
820             MPI_Send(data, world_size, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
821         }
822 
823         // send random for RAND model (GOOD, MEDIUM and RAND) in dinamic topology
824         qualification = randomInt(3);
825         for (int i = 1; i < world_size; i++) {
826             MPI_Send(&qualification, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
827         }
828 
829         freeMemoryArrayDictionary(averageList, world_size);
830         freeMemoryArrayDictionary(standardDeviationList, world_size);
831         freeMemoryArrayDictionary(ranking, world_size);
832     } else {
833         average_ = average(populationPtr);
834         standardDeviation_ = standardDeviation(populationPtr);
835         data[0] = average_;
836         data[1] = standardDeviation_;
837         MPI_Send(data, 2, MPI_DOUBLE, 0, 0, communicator);
838 
839         // received data array (ranking)
840         MPI_Recv(data, world_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
841                  MPI_STATUS_IGNORE);
842 
843         // received  random for RAND model in dinamic topology
844         MPI_Recv(&qualification, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
845                  MPI_STATUS_IGNORE);
846     }
847     MPI_Barrier(communicator);
848 
849     // search element into data array
850     int indexRank = 0;
```

```
851     for (int k = 0; k < world_size; k++) {
852         if (world_rank == data[k]) {
853             indexRank = k;
854             break;
855         }
856     }
857 
858     switch (policyPtr->topology) {
859         case SAME:
860             if (indexRank % 2 == 0) {
861                 package(populationPtr, point, policyPtr, world_rank, 1);
862                 MPI_Send(point, lengthPoint, MPI_FLOAT,
863                             data[indexRank + 1], 0, communicator);
864                 MPI_Barrier(communicator);
865                 MPI_Recv(point, lengthPoint, MPI_FLOAT,
866                             data[indexRank + 1], 0, communicator, &status);
867                 unpack(populationPtr, point, policyPtr, world_rank,
868                         status.MPI_SOURCE, 1);
869             } else {
870                 package(populationPtr, point, policyPtr, world_rank, 1);
871                 MPI_Send(point, lengthPoint, MPI_FLOAT,
872                             data[indexRank - 1], 0, communicator);
873                 MPI_Barrier(communicator);
874                 MPI_Recv(point, lengthPoint, MPI_FLOAT,
875                             data[indexRank - 1], 0, communicator, &status);
876                 unpack(populationPtr, point, policyPtr, world_rank,
877                         status.MPI_SOURCE, 1);
878             }
879         break;
880         case GOODBAD:
881             if (indexRank >= 0 && indexRank < 8) {  // GOOD
882                 package(populationPtr, point, policyPtr, world_rank, 1);
883                 MPI_Send(point, lengthPoint, MPI_FLOAT,
884                             data[indexRank + 16], 0, communicator);
885                 MPI_Barrier(communicator);
886                 MPI_Recv(point, lengthPoint, MPI_FLOAT,
```

```
887                         data[indexRank + 16], 0, communicator, &status);
888                 unpack(populationPtr, point, policyPtr, world_rank,
889                         status.MPI_SOURCE, 1);
890             } else {    // BAD
891                 if (indexRank >= 16 && indexRank < 24) {
892                     package(populationPtr, point, policyPtr, world_rank, 1);
893                     MPI_Send(point, lengthPoint, MPI_FLOAT,
894                                 data[indexRank - 16], 0, communicator);
895                     MPI_Barrier(communicator);
896                     MPI_Recv(point, lengthPoint, MPI_FLOAT,
897                                 data[indexRank - 16], 0, communicator,
898                                 &status);
899                     unpack(populationPtr, point, policyPtr, world_rank,
900                             status.MPI_SOURCE, 1);
901                 } else {    // MEDIUM
902                     if (indexRank % 2 == 0) {
903                         package(populationPtr, point, policyPtr, world_rank, 1);
904                         MPI_Send(point, lengthPoint, MPI_FLOAT,
905                                     data[indexRank + 1], 0, communicator);
906                         MPI_Barrier(communicator);
907                         MPI_Recv(point, lengthPoint, MPI_FLOAT,
908                                     data[indexRank + 1], 0, communicator,
909                                     &status);
910                         unpack(populationPtr, point, policyPtr,
911                                 world_rank, status.MPI_SOURCE, 1);
912                     } else {
913                         package(populationPtr, point, policyPtr,
914                                 world_rank, 1);
915                         MPI_Send(point, lengthPoint, MPI_FLOAT,
916                                     data[indexRank - 1], 0, communicator);
917                         MPI_Barrier(communicator);
918                         MPI_Recv(point, lengthPoint, MPI_FLOAT,
919                                     data[indexRank - 1], 0, communicator,
920                                     &status);
921                         unpack(populationPtr, point, policyPtr,
922                         world_rank, status.MPI_SOURCE, 1);
```

```
                    }
                }
            }
            break;
            case RAND:
                // GOOD = 0, MEDIUM = 1, BAD = 2
                if (qualification == 0) {
                    qualification1 = 1; // MEDIUM
                    qualification2 = 2; // BAD
                } else {
                    if (qualification == 1) {
                        qualification1 = 0; // GOOD
                        qualification2 = 2; // BAD
                    } else {
                        qualification1 = 0; // GOOD
                        qualification2 = 1; // MEDIUM
                    }
                }
                if (indexRank >= qualification1 * 8 && indexRank <
                    (qualification1 + 1) * 8) {  // model 1
                    package(populationPtr, point, policyPtr, world_rank, 1);
                    MPI_Send(point, lengthPoint, MPI_FLOAT,
                                data[indexRank + ((qualification2 -
                                qualification1) * 8)], 0, communicator);
                    MPI_Barrier(communicator);
                    MPI_Recv(point, lengthPoint, MPI_FLOAT,
                            data[indexRank + ((qualification2 -
                            qualification1) * 8)], 0, communicator, &status);
                    unpack(populationPtr, point, policyPtr, world_rank,
                            status.MPI_SOURCE, 1);
                } else {
                    if (indexRank >= qualification2 * 8 && indexRank <
                        (qualification2 + 1) * 8) {  // model 2
                        package(populationPtr, point, policyPtr, world_rank, 1);
                        MPI_Send(point, lengthPoint, MPI_FLOAT,
                                    data[indexRank - ((qualification2 -
```

```
959                             qualification1) * 8)], 0, communicator);
960                     MPI_Barrier(communicator);
961                     MPI_Recv(point, lengthPoint, MPI_FLOAT,
962                             data[indexRank - ((qualification2 -
963                             qualification1) * 8)], 0, communicator,
964                             &status);
965                     unpack(populationPtr, point, policyPtr, world_rank,
966                             status.MPI_SOURCE, 1);
967                 } else {    // model 3
968                     if (indexRank % 2 == 0) {
969                         package(populationPtr, point, policyPtr,
970                                 world_rank, 1);
971                         MPI_Send(point, lengthPoint, MPI_FLOAT,
972                                 data[indexRank + 1], 0, communicator);
973                         MPI_Barrier(communicator);
974                         MPI_Recv(point, lengthPoint, MPI_FLOAT,
975                                 data[indexRank + 1], 0, communicator,
976                                 &status);
977                         unpack(populationPtr, point, policyPtr,
978                                 world_rank, status.MPI_SOURCE, 1);
979                     } else {
980                         package(populationPtr, point, policyPtr,
981                                 world_rank, 1);
982                         MPI_Send(point, lengthPoint, MPI_FLOAT,
983                                 data[indexRank - 1], 0, communicator);
984                         MPI_Barrier(communicator);
985                         MPI_Recv(point, lengthPoint, MPI_FLOAT,
986                                 data[indexRank - 1], 0, communicator,
987                                 &status);
988                         unpack(populationPtr, point, policyPtr,
989                         world_rank, status.MPI_SOURCE, 1);
990                     }
991                 }
992             }
993         break;
994         default:
```

```
995             break;
996         }
997         freeMemoryArrayDouble(data);
998     }
999
1000    void runMigration(MPI_Comm communicator, int world_rank,
1001                      PopulationPtr populationPtr, double *point,
1002                      int lengthPoint, PolicyPtr policyPtr,
1003                      int numberGeneration) {
1004        if(numberGeneration % policyPtr->intervalEmiImm == 0){
1005            switch (policyPtr->topology) {
1006                case RING:
1007                    ringTopology(communicator, world_rank, populationPtr,
1008                                 point, lengthPoint, policyPtr);
1009                break;
1010
1011                case TREE:
1012                    treeTopology(communicator, world_rank, populationPtr,
1013                                 point, lengthPoint, policyPtr);
1014                break;
1015                case NETA:
1016                    netATopology(communicator, world_rank, populationPtr,
1017                                 point, lengthPoint, policyPtr);
1018                break;
1019
1020                case NETB:
1021                    netBTopology(communicator, world_rank, populationPtr,
1022                                 point, lengthPoint, policyPtr);
1023                break;
1024
1025                case TORUS:
1026                    torusTopology(communicator, world_rank, populationPtr,
1027                                 point, lengthPoint, policyPtr);
1028                break;
1029
1030                case GRAPH:
```

```
1031                graphTopology(communicator, world_rank, populationPtr,
1032                                point, lengthPoint, policyPtr);
1033            break;
1034
1035            case SAME:
1036            case GOODBAD:
1037            case RAND:
1038                dinamicTopology(communicator, world_rank, populationPtr,
1039                                point, lengthPoint, policyPtr);
1040            break;
1041        }
1042    }
1043 }
```

Anexo C

Reconocimiento de SIMBig 2020

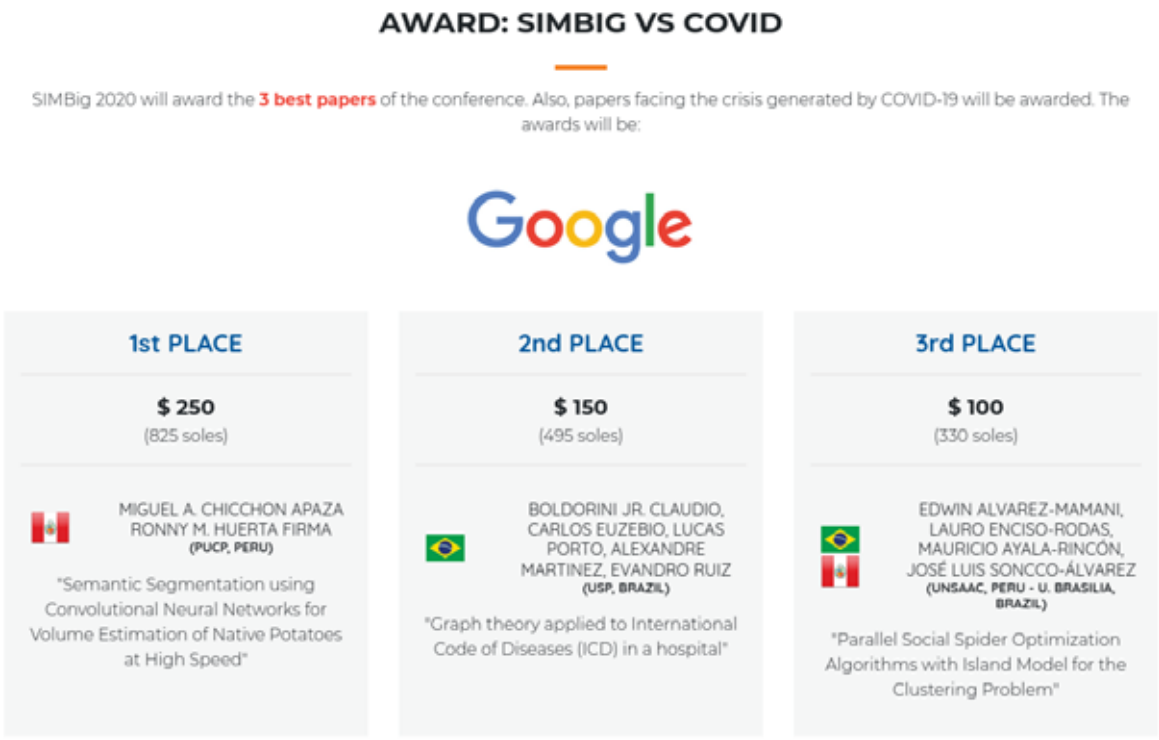


Fig. C.1 Mejor 3° *paper* en SIMBig 2020

Fuente: https://simbig.org/SIMBig2020/#award