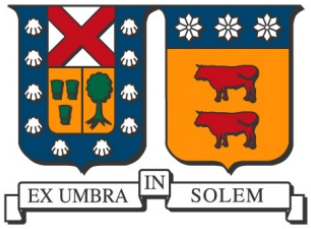
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO- CHILE



"DISEÑO E IMPLEMENTACIÓN PARALELA DE UN SINTONIZADOR DE PARÁMETROS"

ESTEBAN BARRIOS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Nicolás Rojas
Profesor Correferente: Elizabeth Montero U.

Mes (del examen) - Año

DEDICATORIA

Dedico este trabajo a mis padres. Les agradezco por ser mi guía y por brindarme un ejemplo de lo que es la perseverancia y amor inquebrantable. Este trabajo es un tributo a su amor y sacrificio para permitir mis estudios. Espero que se sientan orgullosos de mis logros, tal como yo me siento agradecido y orgulloso de tenerlos como mis padres. En segundo lugar, quiero dedicarme este trabajo a mí mismo, agradecer mi esfuerzo y resiliencia por llevarme hasta aquí y por motivarme a seguir avanzando hacia un futuro prometedor. Cada paso que he dado, cada esfuerzo que he invertido y cada lección que he aprendido han sido valiosos para mi crecimiento personal y profesional.

AGRADECIMIENTOS

A toda mi familia, en especial a mis padres y hermanos, por creer en mí y ofrecer su apoyo incondicional a lo largo de toda mi carrera universitaria. Gracias por estar siempre allí, apoyándome en cada paso que he dado.

A mis amigos y compañeros que me han acompañado durante estos “seis” largos años. No podría haberlo logrado sin su motivación y confianza en mí, incluso cuando yo mismo dudaba. A pesar de los altibajos, siempre les agradeceré por formar parte de mi vida y por el apoyo incondicional que me han brindado.

A mi polola, por ser una fuente constante de apoyo y aliento en los momentos más difíciles. Su paciencia, comprensión y ánimos han sido fundamentales para mantenerme enfocado en el desarrollo de esta memoria. Gracias por estar siempre presente y ser mi mayor motivación.

También quiero expresar mi agradecimiento a todos los docentes del Departamento de Informática, cuyo incansable trabajo y enseñanzas nos han brindado las herramientas para convertirnos en buenos profesionales. Sus conocimientos y dedicación han sido fundamentales en mi formación académica.

Finalmente, quiero agradecer de manera especial a mi profesor guía, Nicolás Rojas, por su valiosa ayuda, paciencia y dedicación durante todo este período.

Cada uno de ustedes ha dejado una huella en mi vida y ha sido parte importante de mi crecimiento personal y profesional. Gracias por formar parte de este camino y ser una parte invaluable de este logro.

RESUMEN

Resumen— La optimización es la capacidad de resolver problemas de forma eficiente, lo que es asociado a la elección de la mejor opción dentro de un conjunto de posibles alternativas. La toma de esta decisión es aplicable en diversos ámbitos de nuestra vida diaria, como la producción y distribución de periódicos, el diseño de rutas para sistemas de transporte, la asignación de puertas de vuelo, entre otros. No obstante, debido a la complejidad NP-difícil de muchos de estos problemas, su solución a menudo requiere el uso de metaheurísticas. Si bien estas técnicas permiten obtener soluciones de alta calidad en un tiempo razonable, surgen desafíos al tener que determinar los valores adecuados para sus parámetros. Para abordar esta problemática se introducen los sintonizadores de parámetros, técnicas que han sido propuestas en la literatura para obtener valores adecuados exitosamente. Sin embargo, su uso emplea grande tiempos de ejecución al considerar: (1) un espacio de búsqueda enorme; y (2) múltiples ejecuciones del algoritmo objetivo en el proceso de sintonización. En este contexto, se presenta PEVOCA, una versión en paralelo del sintonizador Evolutionary Calibrator (EVOCA), diseñada para reducir su tiempo de ejecución sin comprometer la calidad de las soluciones obtenidas. PEVOCA implementa una arquitectura MIMD para obtener las aptitudes individuales de cada configuración, al hacer uso de hebras concurrentes por medio de la programación en multiprocesadores. Se evaluó sintonizando dos algoritmos: un Algoritmo Genético para resolver NK-landscapes (GA-NK) y un Algoritmo de Optimización de Colonias de Hormigas para resolver el problema de la Mochila Multidimensional (AK). Además, se ha logrado determinar en qué casos esta implementación no ofrece beneficios en comparación a su versión secuencial. En específico, este escenario se presenta cuando la sincronización de las hebras concurrentes es mayor que la ejecución de las múltiples instancias del algoritmo objetivo, en cuyo caso, los tiempos globales son significativamente inferiores a los casos complejos. Sin embargo, independientemente de si se genera una mejora, se han obtenido las mismas configuraciones para ambas implementaciones. Todo ello demuestra la eficiencia y eficacia de PEVOCA, como una herramienta prometedora para la búsqueda de configuraciones adecuadas.

Palabras Clave— Optimización; PEVOCA; MIMD; Pthreads

ABSTRACT

***Abstract*—** Optimization is the ability to efficiently solve problems, which is associated with selecting the best option from a set of possible alternatives. Making such decisions is applicable in various aspects of our daily lives, such as newspaper production and distribution, designing routes for transportation systems, allocating flight gates, among others. However, due to the NP-hard complexity of many of these problems, their solutions often require the use of metaheuristics. Although these techniques enable obtaining high-quality solutions in a reasonable time, challenges arise when determining suitable parameter values. To address this issue, parameter tuners are introduced, which are techniques proposed in the literature to successfully obtain appropriate values. Nevertheless, their use implies considerable execution times due to (1) an extensive search space and (2) multiple runs of the objective algorithm in the tuning process. In this context, PEVOCA is presented, a parallel version of the Evolutionary Calibrator (EVOCA) tuner designed to reduce its execution time without compromising the quality of the solutions obtained. PEVOCA implements a MIMD architecture to assess individual fitness values for each configuration by utilizing concurrent threads through multiprocessor programming.Two algorithms were evaluated using the tuning process: a Genetic Algorithm to solve NK-landscapes (GA-NK) and an Ant Colony Optimization Algorithm to solve the Multidimensional Knapsack Problem (AK). Additionally, it has been determined under which circumstances this implementation does not provide benefits compared to its sequential implementation. Specifically, this scenario occurs when the synchronization of concurrent threads is higher than the execution of multiple instances of the target algorithm, in which case the overall times are significantly lower than complex cases. However, regardless of whether an improvement is achieved, the same configurations have been obtained for both implementations. All of this demonstrates the efficiency and effectiveness of PEVOCA as a promising tool for searching suitable configurations.

***Keywords*—** Optimization; PEVOCA; MIMD; Pthreads

GLOSARIO

ENIAC Electronic Numerical Integrator And Computer

ACO Ant Colony Optimization

PSO Particle Swarm Optimization

F-race Friedman Race Algorithm

ANOVA Analysis of Variance

REVAC Relevance Estimation and Value Calibrator

ParamILS Parameter Iterated Local Search

SMAC Sequential Model-based Algorithm Configuration

I-race Iterated F-Race

EVOCA Evolutionary Calibrator

DLP Data Layer Parallelism

TLP Task Layer Parallelism

SISD Single Instruction Single Datastream

SIMD Single Instruction Multiple Datastream

MISD Multiple Instruction Single Datastream

MIMD Multiple Instruction Multiple Datastream

EA Evolutionary Algorithms

dEA Distributed Evolutionary Algorithms

cEA Cellular Evolutionary Algorithms

PEVOCA Parallel Evolutionary Calibrator

AK Ant Knapsack Problem

GA-NK Genetic Algorithm for NK landscape problem

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

Los problemas de optimización son fundamentales en diversas áreas de nuestra vida diaria y en numerosos campos de estudio. Para su resolución existen diferentes métodos expuestos en la literatura, pero los más destacados son: los métodos exactos, las heurísticas y las metaheurísticas.

Los métodos exactos se basan en algoritmos que garantizan encontrar la solución óptima para un problema de optimización. Estos métodos pueden ser aplicados a problemas de tamaño pequeño, pero pueden volverse ineficientes y computacionalmente costosos para problemas más complejos y extensos.

Las heurísticas son técnicas de búsqueda que, aunque no garantizan la obtención de la solución óptima, pueden proporcionar resultados aceptables en un tiempo razonable. Estas estrategias buscan soluciones aproximadas y suelen ser más eficientes para problemas de mayor tamaño y complejidad.

Las metaheurísticas, por su parte, son un conjunto de técnicas que buscan la eficiencia computacional en la implementación de heurísticas. Son modelos que pueden abordar problemas de optimización con alta complejidad y dimensionalidad. No obstante, el uso de metaheurísticas también presenta desafíos, uno de ellos es la configuración de sus parámetros, ya que sus valores pueden tener un impacto significativo en la calidad y eficiencia de las soluciones encontradas. Este proceso, conocido como sintonización de parámetros, puede requerir un tiempo considerable debido al inmenso espacio de búsqueda que se debe explorar y a la necesidad de ejecutar repetidamente la metaheurística objetivo.

En particular, se debe considerar que muchos de estos problemas complejos, que se abordan mediante el uso de metaheurísticas, demandan soluciones rápidas que deben resolverse en cuestión de minutos o incluso segundos. Un ejemplo ilustrativo es una simulación en la predicción del clima, donde cualquier retraso significativo en la simulación o un desempeño deficiente podría acarrear la obtención de resultados inviables, al proporcionar predicciones de climas que ya han ocurrido o predecir erróneamente el clima, al decir que va a llover, y ocurre un día soleado. Por lo tanto, la sintonización de parámetros cobra una importancia aún mayor, ya que valores inadecuados para las metaheurísticas pueden llevar a este tipo de comportamientos no deseados. Por ende, es fundamental que este proceso sea llevado a cabo en el menor tiempo posible.

En los últimos años, se ha observado en las arquitecturas de las CPU que la velocidad máxima del procesamiento secuencial se alcanzará prontamente debido a las limitaciones impuestas por el hardware. Por lo cual, ha surgido el enfoque de multinúcleos o multiprocesadores como una solución para acelerar los procesos y mejorar el rendimiento de los computadores. Este enfoque, como su nombre sugiere, implica el uso de múltiples procesadores para llevar a cabo una misma tarea.

En la actualidad, los chips multinúcleo se encuentran prácticamente en todos los sistemas computacionales, lo que ha permitido un aumento significativo en la velocidad de procesamiento. Esta realidad brinda una nueva posibilidad para acelerar los proceso vinculados a la sintonización de parámetros, al hacer uso del paralelismo en su implementación.

En este contexto, se presenta PEVOCA, una nueva versión del sintonizador Evolutionary Calibrator (EVOCA), diseñada para reducir el tiempo de ejecución de la sintonización de parámetros al hacer uso de paralelismo. También, en esta nueva versión se compromete a mantener la calidad de las soluciones obtenidas de su versión original.

Este trabajo se compone de cinco capítulos que abordan diversos aspectos de la solución implementada. En el Capítulo 1, se contextualiza el problema a nivel general y se resalta el beneficio derivado de emplear paralelismo, estableciendo los objetivos para este nuevo acercamiento. El Capítulo 2 expone el marco conceptual, haciendo énfasis en la definición general del problema de sintonización de parámetros y revisando trabajos previos que ofrecen diferentes enfoques para su resolución. Se realiza una exhaustiva revisión en la literatura, abarcando la definición, clasificación e implementaciones en paralelo de algoritmos evolutivos, así como los diferentes tipos, arquitecturas y formas de programar el paralelismo. En el Capítulo 3, se brinda un análisis detallado del sintonizador EVOCA, junto con el diseño de una arquitectura en paralelo, donde se describe cómo será implementada y su correspondiente funcionamiento. El Capítulo 4 presenta los resultados obtenidos durante el proceso de validación de la propuesta, incluyendo la comparación de dos escenarios distintos para evaluar las dos versiones de EVOCA. Por último, el Capítulo 5 concluye acerca de los resultados alcanzados, resaltando las limitaciones de la solución propuesta y sugiriendo diferentes enfoques para futuros trabajos en esta área.

CAPÍTULO 1
DEFINICIÓN DEL PROBLEMA

1.1. Contexto del Problema

La esencia de los problemas de optimización radica en la búsqueda de la solución óptima, ya sea el máximo o el mínimo, entre un conjunto dado de posibilidades. Un ejemplo clásico de ello, es el problema de la mochila o knapsack problem (Karp, 1972). En este problema, se tiene una mochila con capacidad limitada y un conjunto de objetos con un peso (o volumen) determinado y una ganancia asociada. El objetivo es determinar qué objetos se deben incluir en la mochila para maximizar la ganancia total de ella, sin exceder su capacidad. La ganancia puede estar relacionada a diferentes características de los objetos, por ejemplo, en un escenario logístico, puede estar asociado al valor monetario de los objetos transportados, mientras que en un contexto de supervivencia en la naturaleza, puede representar la capacidad que tienen los objetos de satisfacer necesidades básicas. En este contexto, encontrar la mejor solución implica seleccionar los elementos más adecuados del conjunto de objetos que cumplen con las condiciones mencionadas. Este proceso se denomina optimización, y existen diferentes métodos para abordar su resolución, entre ellos están: los métodos exactos, las heurísticas y las metaheurísticas.

Los métodos exactos son técnicas de optimización que buscan encontrar la mejor solución de un problema de manera exhaustiva, exploran todas las posibles soluciones en el espacio de búsqueda[1]. Lo que implica que la solución encontrada sea el óptimo global del problema, al no existir una mejor opción disponible.

Por el contrario, las heurísticas son procedimientos que buscan encontrar una solución aproximada del problema. Es decir, no pueden garantizar que la solución encontrada sea el óptimo global del problema, ni pueden establecer qué tan cerca o lejos se encuentran de él. Sin embargo, las heurísticas si aseguran que la solución encontrada es la mejor opción posible de acuerdo a su procedimiento. A este tipo de soluciones se les denomina como un óptimo local del problema, ya que representa la mejor solución encontrada dentro de un subconjunto del espacio de búsqueda.

Las metaheurísticas son métodos que se encuentran en un nivel de abstracción más alto que las heurísticas. Estas técnicas son estrategias de diseño que buscan guiar los procedimientos heurísticos para obtener un mejor desempeño en la búsqueda de soluciones. Se pueden considerar como plantillas sobre las cuales implementar algoritmos heurísticos, que resuelven problemas de optimización. Además, dada la forma en como abordan dicha resolución, o en otras palabras, dado en como recorren el espacio de búsqueda, se pueden clasificar de acuerdo a dos categorías:

1. Basados en solución: Estos métodos describen una trayectoria en el espacio de búsqueda, al ir mejorando una única solución durante el proceso de búsqueda. Algunos ejemplos de

[1]Un espacio de búsqueda es el conjunto de todas las posibles soluciones factibles de un problema dado.

este tipo de métodos son el algoritmo de recocido simulado (Kirkpatrick, Gelatt, y Vecchi, 1983), el algoritmo búsqueda tabú (Glover, 1986), el algoritmo de búsqueda adaptativa al azar codiciosa (Feo y Resende, 1989), el algoritmo de búsqueda local iterativa (Laurenso, Martin, y Stutzle, 1994), entre otros.

2. Basados en población: Estos métodos describen la evolución de un conjunto de puntos en el espacio de búsqueda, al ir mejorando una población de soluciones durante el proceso de búsqueda. Algunos ejemplos de este tipo de métodos son el algoritmo genético (Holland, 1992), el algoritmo de optimización por enjambre de partículas (Kennedy y Eberhart, 1995), el algoritmo de optimización por colonias de hormigas (Dorigo, Maniezzo, y Colorni, 1996), entre otros.

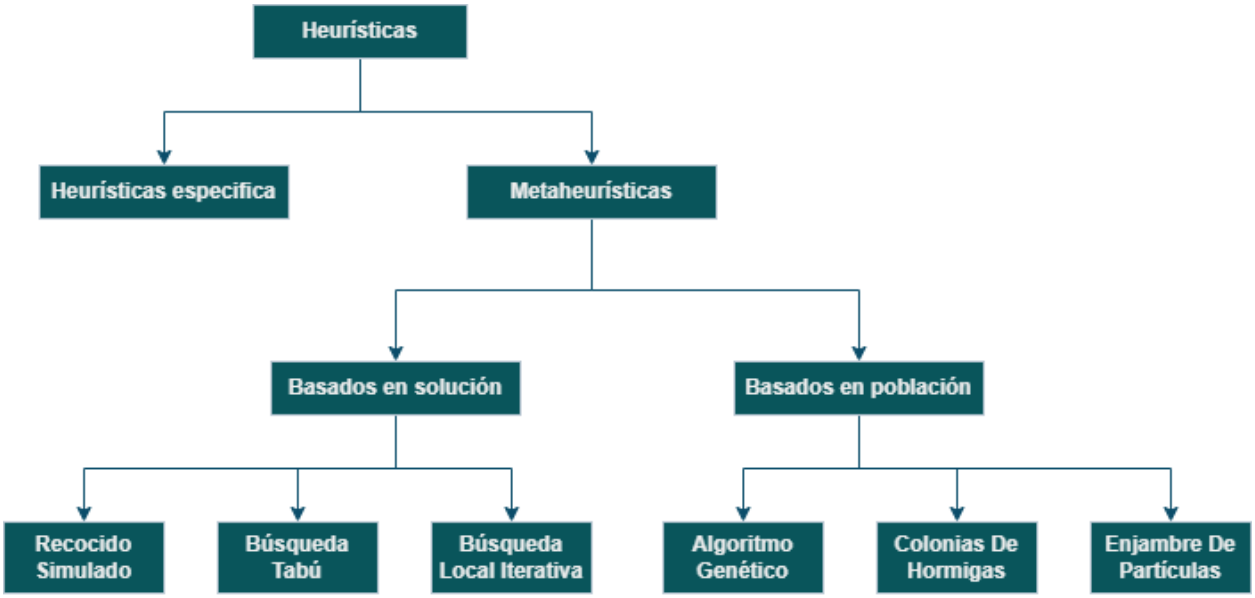


Figura 1: Taxonomía de heurísticas
Fuente: Elaboración propia.

En la figura 1 se muestra la taxonomía de las heuristicas, en donde se debe considerar dos enfoques principales para resolver problemas de optimización al hacer uso de este metodo: la aplicación de heurísticas específicas y el uso de metaheurísticas. Tambíen, se preseta la subdivisión de las metaheurísticas en los dos tipos mencionados previamente, y se ofrecen tres ejemplos ilustrativos para cada categoría.

En la actualidad, la aplicación de estos métodos se puede ver reflejada en la resolución de problemas de muchas disciplinas. Se aprecian en la producción y distribución de periódicos (Chiang,

Russell, Xu, y Zepeda, 2009), en la planificación de sesiones de radioterapia a pacientes con cáncer (Petrovic, Morshed, y Petrovic, 2011), en el diseño de rutas para sistemas de transporte público (Fan y Mumford, 2010), en la asignación de las puertas de vuelos (Marinelli, Dell'Orco, y Sassanelli, 2015), entre otros. Sin embargo, los problemas de optimización de la vida real a menudo se clasifican como NP-difícil.[2] Esto implica que el uso de métodos exactos para su resolución se vuelva inviable, ya que se requiere de tiempos extremadamente largos para converger a una solución. Una de las principales razones de ello se asocia a los espacios de búsqueda enormes, por lo cuál, una posibilidad para abordar esta situación considera el uso de metaheurísticas. Estas permiten reducir significativamente la complejidad computacional al no considerar la totalidad del espacio de búsqueda. No obstante, al utilizar este tipo de métodos, se requiere, además, configurar previamente ciertos parámetros para obtener un buen rendimiento del algoritmo. Por ejemplo: la tasa de mutación en un algoritmo genético, la temperatura inicial en un algoritmo recocido simulado, el largo de la lista de nodos excluidos en una búsqueda tabú, etc.

Encontrar una configuración adecuada de parámetros para estos algoritmos también puede ser considerado un problema de optimización y es un proceso complejo que en la mayoría de los casos considera altos tiempos de ejecución debido a: (1) Un espacio de búsqueda enorme, ya que implica considerar todos los posibles dominios de los parámetros; y (2) Múltiples ejecuciones del algoritmo objetivo (algoritmo a sintonizar) durante el proceso de búsqueda, debido al manejo de la estocasticidad del sintonizador. En otras palabras, dado que un sintonizador no siempre producirá la misma salida cuando se le proporcionen los mismos parámetros iniciales, se hace necesario manejar esta aleatoriedad mediante el cálculo de múltiples instancias que se ejecutan bajo los mismos parámetros. De esta manera, con los valores obtenidos se puede validar la aptitud de una configuración en particular, ya sea mediante el promedio de las evaluaciones o mediante otro procedimiento.

En la literatura se han propuesto diversas estrategias para abordar este problema y se pueden agrupar de acuerdo al momento de su ejecución:

- Sintonización de parámetros (offline): Los parámetros se configuran previo al proceso de búsqueda y permanecen inmutables. Ejemplos que se pueden encontrar son Irace (López-Ibáñez, Dubois-Lacoste, Cáceres, Birattari, y Stützle, 2016), EVOCA (Riff y Montero, 2013), SMAC (Hutter, Hoos, y Leyton-Brown, 2011), REVAC (Nannen y Eiben, 2007a), entre otros.
- Control de parámetros (online): Consiste en ir adaptando los parámetros del sintonizador durante su proceso de búsqueda. Por ejemplo, si se toma el caso de un algoritmo recocido simulado, el control de parámetros durante su búsqueda sería ir ajustando los parámetros de la temperatura inicial y de la tasa de enfriamiento a medida que se van obteniendo soluciones. Este tipo de ajuste se realiza típicamente para abordar problemas y algoritmos específicos relacionados con un contexto particular, como se ha evidenciado en estudios como (Bouneffouf y Claeys, 2021) y (Chau, Thonnat, Bremond, y Corvee, 2014).

[2] Un problema es clasificado como NP-difícil, si algún problema NP puede ser transformado a él en tiempo polinomial. Por otro lado, un problema es clasificado como NP si no puede ser resuelto en tiempo polinomial.

Esta memoria se enfocará en la sintonización de parámetros, ya que con estos se puede garantizar la calidad de la solución obtenida al aplicar una misma configuración de parámetros a un grupo de instancias del problema y se podrá apreciar más en detalle en el capítulo 2.

1.2. Computación Paralela

Durante los últimos 77 años, desde la creación de la primera computadora electrónica de propósito general (ENIAC), se ha observado un progreso increíble en los sistemas computacionales. En la actualidad, es posible adquirir un teléfono móvil con un rendimiento, una memoria principal y un almacenamiento en disco superiores a los de un computador adquirido en 1985. Esta rápida evolución ha sido impulsada, tanto por los avances en la tecnología de su fabricación, como por las innovaciones en el diseño de los computadores. Sin embargo, aunque las mejoras tecnológicas han sido bastante constantes, existen limitaciones inherentes al hardware y a la tecnología disponible. Es por ello, que la mejora en el rendimiento de los sistemas computacionales solo ha sido posible principalmente por la introducción de paralelismo en la arquitectura de los sistemas.

Para comprender este fenómeno, es necesario destacar dos cambios significativos que ocurren el mercado de computadoras que facilitaron el éxito comercial de una nueva arquitectura. Primero, la eliminación virtual de la programación en lenguaje ensamblador, que redujo la necesidad de compatibilidad de código-objeto. Segundo, la creación de sistemas operativos estandarizados e independientes del proveedor, como UNIX y su clon, Linux, que redujo el costo y el riesgo asociados al lanzamiento de una nueva arquitectura. Estos cambios sentaron las bases para una de las primeras aproximaciones al paralelismo a principios de la década de 1980, a través del desarrollo de un conjunto de arquitecturas con instrucciones más simples conocidas como RISC (Reduced Instruction Set Computer). Estas máquinas basadas en RISC se centraron en dos técnicas principales para mejorar el rendimiento: la explotación del paralelismo a nivel de instrucción, inicialmente a través de la segmentación en etapas y posteriormente mediante la emisión de múltiples instrucciones, y el uso de cachés, inicialmente en formas básicas y posteriormente con organizaciones y optimizaciones más sofisticadas (Hennessy y Patterson, 2011).

Las computadoras basadas en RISC elevaron tanto el nivel de rendimiento, que obligó a las arquitecturas anteriores a mantenerse al día o desaparecer. Por ejemplo, el equipo Digital Equipment Vax no pudo mantenerse al ritmo y fue reemplazado por una arquitectura RISC. Mientras que Intel también enfrentó el desafío al traducir las instrucciones 80x86 en instrucciones internamente similares a RISC, lo que le permitió adoptar muchas de las innovaciones pioneras en los diseños RISC.

En la Figura 2 se presenta el crecimiento del rendimiento de los procesadores desde finales de la década de 1970. Se observa que, antes de mediados de la década de 1980, el incremento en el rendimiento del procesador se basaba principalmente en avances tecnológicos y promediaba alrededor del 25 % anual. Desde entonces, existe un aumento en el crecimiento hasta el 52 % anual que se le atribuye a las ideas arquitectónicas y organizativas más avanzadas (introducción

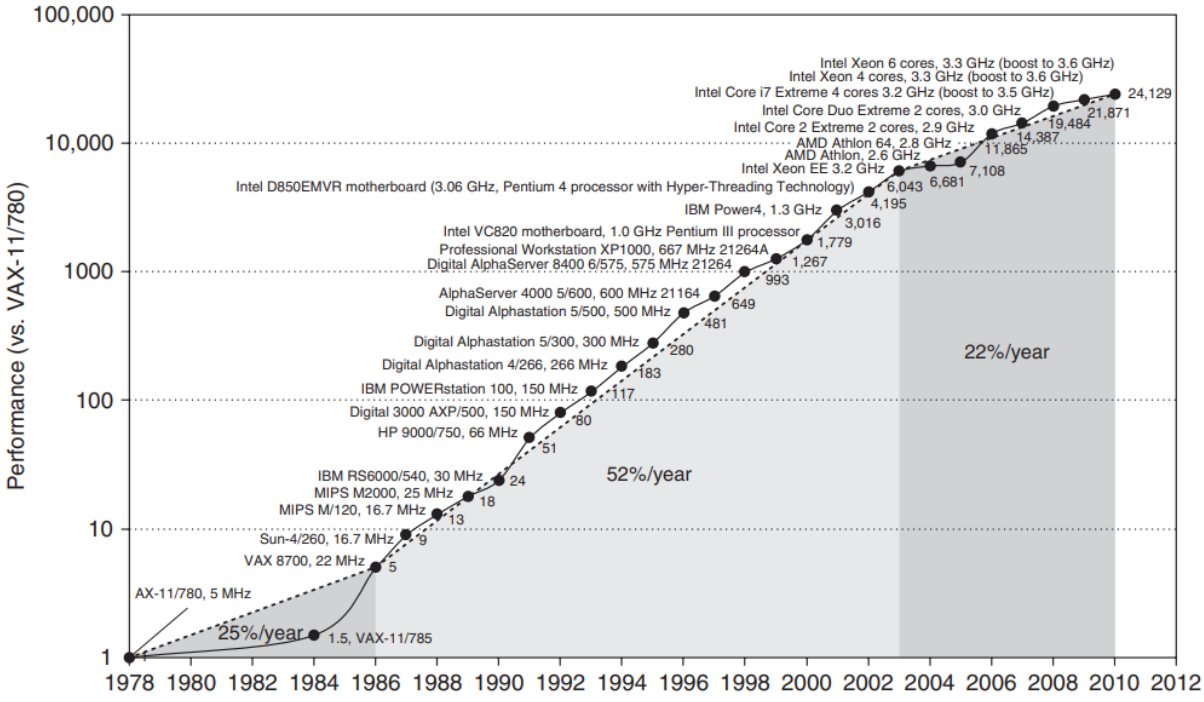


Figura 2: Crecimiento del rendimiento de los procesadores relativo al VAX 11/780
Fuente: (Hennessy y Patterson, 2011).

del paralelismo). Para el año 2003, este crecimiento condujo a una diferencia de rendimiento de aproximadamente en 25 %, en comparación a que si se hubiera continuado con el mismo ritmo. Sin embargo, desde ese año, la mejora en el rendimiento de los procesadores individuales ha disminuido a menos del 22 % por año, o aproximadamente 5 veces más lento de lo que habría sido si hubiéramos continuado al ritmo del 52 % por año. Esta desaceleración se debe principalmente a dos obstáculos principales: la disipación máxima de energía de los chips refrigerados por aire y la limitada capacidad de aprovechar eficientemente el paralelismo a nivel de instrucción (Hennessy y Patterson, 2011). Por esta razón, en el 2004, Intel canceló sus proyectos de alto rendimiento para un solo procesador y se sumó a otros fabricantes al afirmar que el camino hacia un mayor rendimiento residiría en la incorporación de múltiples núcleos por chip en lugar de procesadores únicos más rápidos.

Es base a esta limitaciones inherentes en el hardware, existe la constante afirmación de que la velocidad máxima del procesamiento secuencial se alcanzará prontamente. Es por ello que surge la necesidad de nuevos enfoques en paralelo. Es más, existen muchas áreas de aplicación donde el poder de cálculo de una computadora simple es insuficiente para obtener resultados deseables. En el área industrial, por ejemplo, si una simulación se demora algunas semanas en alcanzar resultados, es usualmente inaceptable en ambientes de diseño, ya que los tiempos con los que cuenta el diseñador son cortos. También, existen problemas en donde se requiere cumplir con

fechas límites, como por ejemplo, en la predicción del tiempo donde una demora en la simulación implica su inviabilidad (Aguilar, Leiss, y cols., 2004).

El procesamiento paralelo es una forma de procesar la información que permite la ejecución simultánea de múltiples procesos. Su uso esta relacionado principalmente a: (1) La necesidad de mayor potencia de cálculo; (2) Una mejor relación costo/rendimiento; y (3) Los modelos de procesamiento paralelo son más idóneos para la problemática enfrentada. Se puede definir la sintonización de parámetros, como uno de estos casos en donde una simulación puede tomar hasta algunas semanas para alcanzar buenos resultados, aún cuando se utilizan diferentes estrategias para reducir el tiempo computacional de la búsqueda de una configuración adecuada. Por lo cual, se puede beneficiar con el uso de la computación en paralelo en su implementación.

1.3. Objetivos

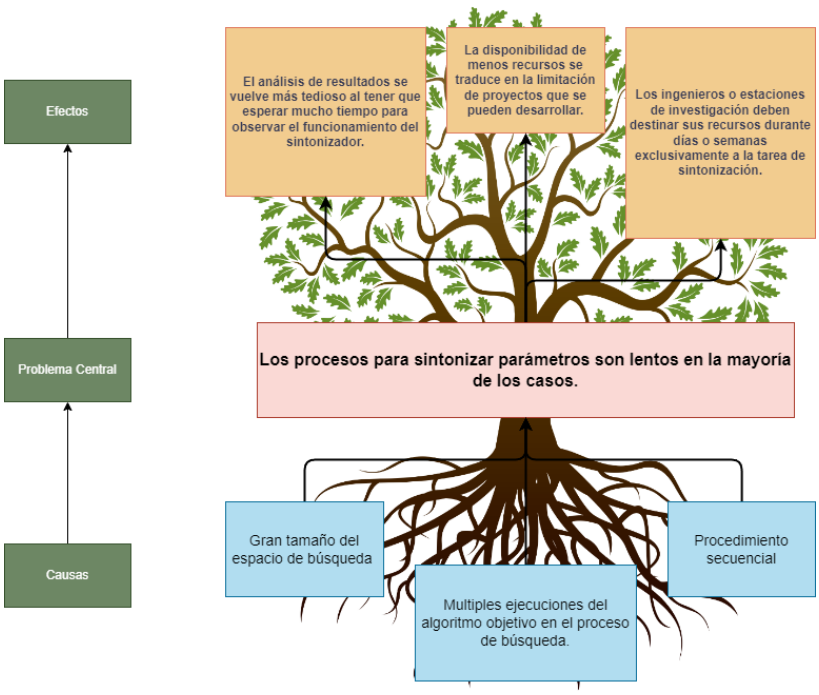


Figura 3: Árbol del problema.
Fuente: Elaboración propia.

Un sintonizador de parámetros desempeña un papel crucial en la optimización de algoritmos, ya que busca encontrar la combinación adecuada de valores de parámetros que maximicen el rendimiento de los algoritmos objetivo. Sin embargo, a medida que los algoritmos se vuelven más complejos y la cantidad de parámetros aumenta, el tiempo necesario para realizar esta búsqueda exhaustiva se vuelve excesivo.

En la figura 3 se muestran las principales causas y efectos que presenta la demora en el proceso

de los sintonizadores. En esta se puede observar que las causas asociadas son: (1) un espacio de búsqueda enorme;(2) múltiples ejecuciones del algoritmo objetivo en el proceso de búsqueda; y (3) por lo discutido anteriormente, se puede considerar otra razón plausible el uso de un proceso secuencial en el proceso de búsqueda. Por otro lado, se muestran que los principales efectos que produce esta demora están asociados a los tiempos que debe dedicar el diseñador, ya que se traduce directamente en un aumento en el tiempo que deben dedicar para encontrar una configuración adecuada. Es más, al considerar que los sintonizadores se pueden demorar días y hasta semanas en obtener resultados para una única simulación, este tiempo se debe esperar repetidas veces para analizar los resultados de su algoritmo. Esto trae como consecuencia, que el tiempo que se debe gastar en el proceso de mejora sea muy tedioso y lento. Lo que a su vez, produce que se destinen recursos por una mayor cantidad de tiempo para alcanzar las metas correspondientes, y se generen demoras en el desarrollo de nuevos proyectos de investigación por la demora de los antiguos.

En esta memoria se propone llevar a cabo una implementación paralela de un sintonizador de parámetros con el fin de reducir el tiempo requerido para encontrar configuraciones adecuadas en los algoritmos objetivo. Para abordar este desafío, se propone la implementación de una estrategia de paralelización al sintonizador Evolutionary Calibrator (EVOCA). La paralelización implica la división del proceso de búsqueda en múltiples subprocesos que se ejecutan concurrentemente, lo que permite explorar diferentes combinaciones de parámetros de manera más eficiente. Al distribuir la carga de trabajo entre varios procesadores o núcleos, se busca lograr una aceleración en el tiempo total requerido para encontrar las configuraciones adecuadas.

OBJETIVO GENERAL

- Diseñar e implementar una paralelización del sintonizador de parámetros EVOCA para reducir su tiempo de ejecución sin pérdida de calidad de las configuraciones obtenidas.

OBJETIVOS ESPECÍFICOS

- Verificar la literatura relacionada a los sintonizadores existentes, sobre el paralelismo y los algoritmos evolutivos.
- Diseñar una arquitectura en paralelo para Evoca.
- Implementar arquitectura en paralelo para Evoca.
- Evaluar la implementación paralela en diferentes algoritmos objetivos para verificar los resultados obtenidos y comparar con su versión secuencial considerando instancias benchmark.

CAPÍTULO 2
MARCO CONCEPTUAL

El presente capítulo tiene como objetivo comprender 3 principales ejes, por su posterior uso en esta investigación: las metaheurísticas basadas en poblaciones, Los fundamentos de la sintonización de parámetros y el paralelismo, su eficiencia e implementaciones en algoritmos evolutivos.

Para el primer eje, se explorará en profundidad el campo de los algoritmos evolutivos, una poderosa clase de metaheurísticas que son inspirados en los mecanismos de evolución biológica y selección natural. Se revisarán en detalle los fundamentos de estos algoritmos y se presentará el algoritmo genético, como ejemplo de este tipo de algoritmos. Por otro lado, se expondrá el algoritmo de optimización por colonias de hormigas, otro metaheuristica basada en poblaciones, pero que se inspira en el comportamiento de las hormigas y como estás se comunican para llevar comida a su hormiguero.

Para el segundo eje, se definirá un marco general sobre el cual se desarrolla la sintonización de parámetros y, para ampliar la visión de las diferentes estrategias empleadas en este proceso, se llevará a cabo una exhaustiva revisión de la literatura, analizando trabajos previos que han ofrecido enfoques innovadores y eficientes en la resolución de este problema, entre ellos: F-race, REVAC, ParamILS, SMAC, I-race y EVOCA. Cada uno con sus características y aplicaciones particulares, pero haciendo un énfasis especial en este último, ya que es el método que se utilizará en esta memoria.

En el último eje, se buscará comprender qué es el paralelismo y cómo su uso permite la mejora en el rendimiento de los algoritmos. Para ello, se realizará una revisión detallada sobre las distintas formas de implementar el paralelismo, destacando las arquitecturas paralelas más utilizadas en la actualidad y las técnicas de programación en multiprocesadores para aplicar su uso.

2.1. Metaheurísticas basadas en poblaciones

Las metaheurísticas basadas en poblaciones son un tipo de algoritmo de optimización que se inspira, generalmente, en procesos naturales, como la evolución biológica, el comportamiento en una colonia de hormigas, el comportamiento de las partículas, entre otros. Estos algoritmos utilizan una población de soluciones candidatas para explorar y buscar en el espacio de búsqueda de un problema de optimización.

La idea fundamental detrás de las metaheurísticas basadas en poblaciones es que, al trabajar con una población de soluciones en lugar de una sola solución, se puede aumentar la probabilidad de encontrar soluciones de mejor calidad y evitar quedar estancados en óptimos locales.

2.1.1. Algoritmos Evolutivos

Una de las grandes familias relacionadas a las metaheurísticas basadas en poblaciones, son los algoritmos evolutivos (EA's, por su siglas en inglés). Este tipo de algoritmos son métodos de búsqueda estocástica que se basan en la evolución de poblaciones de individuos al imitar el comportamiento evolutivo de la naturaleza. Esta idea se basa en la propuesta de Darwin sobre la selección natural, donde los rasgos heredables que benefician la supervivencia y la reproducción de un organismo se vuelven más comunes en una población a medida que pasa el tiempo.

Al hablar de la supervivencia de los individuos y cómo sus rasgos se heredan, desde una perspectiva genética, se consideran dos aspectos fundamentales: el genotipo y el fenotipo. El genotipo se refiere al conjunto de genes y la información genética que conforman a un individuo de cualquier especie, sus características. Mientras que, el fenotipo se relaciona con la expresión física de estas características, como los rasgos físicos observables o los comportamientos de un individuo. Por ejemplo, a partir de un determinado conjunto de genes (genotipo), se puede manifestar el color azul en los ojos (fenotipo). No obstante, es solo el genotipo el que puede ser transmitido de generación en generación, es decir, al generar nuevos individuos en una población (reproducción) las características que se pueden heredar de sus padres son de sus genotipos y no de sus fenotipos.

En el contexto de los algoritmos evolutivos, la representación de una solución candidata se asocia a ambas perspectivas. El genotipo de la representación hace referencia cómo se codifica, es decir, a la estructura de datos que contiene la información "genética" que define las características de la solución. Por otro lado, el fenotipo hace referencia a la propia solución, es decir, la interpretación del genotipo según el contexto del problema. En este sentido, existen funciones asociadas al nivel del genotipo que se conocen como operadores, los cuales permiten la generación de nuevos individuos bajo la misma codificación. Asimismo, existen funciones asociadas al fenotipo que se encargan de calcular la aptitud de la solución candidata (Talbi, 2009). La figura 4 muestra un diagrama sobre la asociación descrita anteriormente.

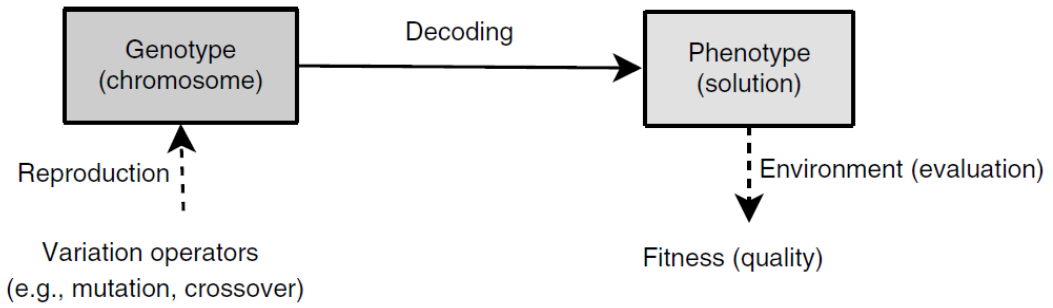


Figura 4: Interpretación de la representación de un solución en algoritmos evolutivos
Fuente: (Talbi, 2009).

Aunque existen diversas familias de algoritmos evolutivos, se puede establecer un proceso general para su funcionamiento. Para ello, primero se debe definir una representación para las soluciones del problema y una función de evaluación para calcular las aptitudes (calidad) de las soluciones. A partir de ello, es posible generar aleatoriamente un conjunto de soluciones candidatas seleccionando elementos del dominio de la función, o en otras palabras, inicializar una población.

De esta manera, en la figura 5 se presenta un diagrama completo del proceso general en donde, una vez que la población está inicializada, se calcula la aptitud de cada individuo.De estos individuos se seleccionan los candidatos con una mejor aptitud para formar la siguiente generación de individuos, ya que estos son los más probables de sobrevivir. Para generar la nueva generación, se aplican operadores de cruzamiento y/o mutación en los individuos seleccionados. Por un lado, el cruzamiento se debe aplicar a dos o más de los candidatos seleccionados, lo que genera uno o más candidatos nuevos. Por otro lado, la mutación tiene cierta probabilidad de ocurrencia y se aplica individualmente a los candidatos, al introducir cambios aleatorios en ellos, lo que también produce nuevos individuos. Por lo tanto, al ejecutar las operaciones de cruzamiento y mutación en los “padres”, se crea un conjunto de nuevos candidatos o “hijos”. (Eiben y Smith, 2015).

Posteriormente, se evalúa la aptitud de esta descendencia y se produce una competencia basada en su aptitud con los individuos antiguos para determinar cuáles serán seleccionados para formar parte de la siguiente generación. Este proceso se puede repetir hasta encontrar una solución con suficiente calidad o hasta alcanzar un límite computacional previamente establecido, que denominamos criterio de termino.

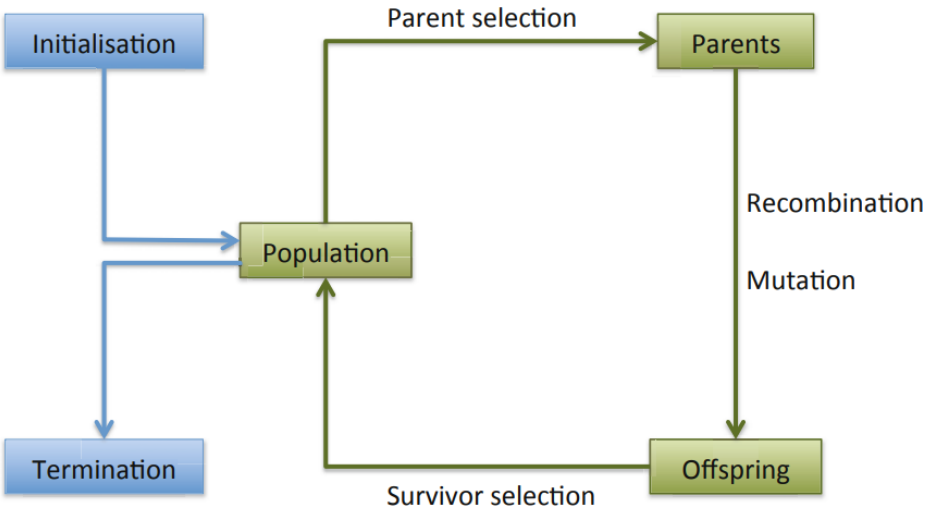


Figura 5: Proceso general de los algoritmos evolutivos
Fuente: (Eiben y Smith, 2015).

En resumen, los algoritmos evolutivos pueden ser interpretados como un proceso de mejora

iterativa sobre una población de soluciones (Talbi, 2009), en el cuál se aplican diferentes operadores sobre una población inicial para crear una población temporal, se evalúan las aptitudes los individuos resultantes y luego reemplazan la población actual al utilizar la población temporal y/o la población antigua para seleccionar que individuos estarán en su siguiente generación. Para ello se requiere definir los siguientes elementos:

1. Representación de las soluciones.
2. Función de evaluación para calcular las aptitudes.
3. Población inicial.
4. Mecanismo de selección de padres.
5. Operadores de variación, cruzamiento y mutación.
6. Mecanismo de selección de sobrevivientes.
7. Criterio de termino.

2.1.2. Algoritmos Genéticos

Los Algoritmos Genéticos (GA, por sus siglas en inglés) son un tipo de algoritmos evolutivo, que fue presentado en sus inicios en (Holland, 1992). Es un algoritmo que basa su comportamiento en como evoluciona un conjunto de genes en una población. Al ser un algoritmo evolutivo, se basa en la lógica anteriormente descrita, por lo que define:

1. Representación de las soluciones: Tradicionalmente, las soluciones se representan como cadenas binarias de 0s y 1s, pero también son posibles otras codificaciones de la solución.
2. Función de evaluación: Se define una función de evaluación que calcule la aptitud o valor objetivo de cada solución candidata en la población. Esta función mide la calidad de las soluciones y es específica para el problema que se está abordando. El objetivo es maximizar o minimizar esta función según el problema a resolver.
3. Población inicial: El tamaño de la población depende de la naturaleza del problema, pero generalmente contiene varias centenas o miles de posibles soluciones. A menudo, la población inicial se genera de forma aleatoria, abarcando todo el rango de posibles soluciones en el espacio de búsqueda.
4. Mecanismo de selección de padres: Se implementa un mecanismo de selección para elegir soluciones candidatas (padres) que participarán en el proceso de reproducción. Las soluciones con una mayor aptitud tienen una mayor probabilidad de ser seleccionadas, lo que se puede lograr mediante técnicas como la selección por ruleta o el torneo de selección.

5. Operadores de variación, cruzamiento y mutación: Se aplican operadores de variación para crear nuevos individuos en la población a partir de los padres seleccionados. El cruzamiento (crossover) implica combinar información de dos padres para generar descendientes. La mutación introduce cambios aleatorios en algunos genes de los cromosomas para aumentar la diversidad de la población. Operadores clásicos de ello son: el cruzamiento uniforme y la mutación de cadena de bits.

6. Mecanismo de selección de sobrevivientes: Una vez que se ha creado la nueva población mediante cruzamiento y mutación, se aplica un mecanismo de selección de sobrevivientes para decidir qué soluciones se mantendrán en la población actual. Las soluciones más aptas tienen más probabilidades de ser seleccionadas para la siguiente generación.

7. Criterio de término: Las condiciones habituales para la terminación de los algoritmos genéticos son diversas. Por lo general, se considera que el algoritmo ha alcanzado su fin cuando se encuentra una solución que satisface los criterios mínimos establecidos para el problema, se ha llegado a un número predefinido de generaciones, se ha agotado la cantidad de recursos asignado, la aptitud de la mejor solución se estanca en un punto de estabilidad sin mejoras significativas, o se realiza una inspección manual para validar una solución satisfactoria. Estas condiciones pueden, inclusive, combinarse de manera flexible para determinar el término del algoritmo, adaptándose a los requerimientos específicos del problema.

En resumen, los GA son una clase de algoritmos evolutivos que se inspiran en la teoría de la evolución biológica, en particular de lo genes. Estos algoritmos buscan encontrar soluciones óptimas o cercanas al óptimo global en problemas complejos de optimización. La implementación de un GA implica definir la representación de las soluciones, establecer una función de evaluación para medir la aptitud de cada solución, y determinar el tamaño y la generación inicial de la población. Además, se aplican operadores de selección, cruzamiento y mutación para crear nuevas soluciones, y se reemplaza la población para formar una siguiente generación con individuos más aptos. La terminación del GA se basa en condiciones como encontrar una solución satisfactoria, alcanzar un número predefinido de generaciones, agotar la cantidad de recursos asignada o alcanzar una meseta en la aptitud de la mejor solución (óptimo local).

2.1.3. Algoritmo de Optimización por Colonias de Hormigas

El Algoritmo de Optimización por Colonia de Hormigas (ACO, por sus siglas en inglés) se presenta en (Dorigo y cols., 1996), como una técnica para abordar el conocido problema del vendedor viajero (TSP, por sus siglas en inglés). El objetivo del TSP es determinar la ruta más corta que un vendedor debe seguir para visitar un conjunto de ciudades exactamente una vez y regresar a su punto de partida. Para lograr este propósito, ACO modela el problema como la búsqueda del camino de menor costo en un grafo y distribuye la de búsqueda de soluciones entre hormigas artificiales, es decir, agentes con capacidades básicas muy simples que, hasta cierto punto, imitan el comportamiento de las hormigas reales.

El comportamiento de las hormigas artificiales se inspira en el de las hormigas reales y su habilidad para encontrar el camino más corto entre su hormiguero y una fuente de alimento. Para alcanzar este objetivo, las hormigas reales intercambian información mediante feromonas, una sustancia que depositan (en cantidades variables) en el suelo mientras se desplazan desde el hormiguero hacia la fuente de alimento y viceversa. Como resultado de este proceso, se genera una huella de feromonas que ejerce un poderoso efecto de atracción sobre otras hormigas, induciéndolas a seguir el camino generado por la huella. De esta manera, la ruta más corta se refuerza progresivamente y se genera una tendencia para que futuras hormigas opten por seguir este recorrido hacia la fuente de alimento.

De manera análoga, el comportamiento de las hormigas artificiales se caracteriza por dos aspectos fundamentales: (1) el depósito de cierta cantidad de feromonas en los componentes del grafo que la hormiga visita; y (2) la elección de su siguiente destino basado en el atractivo de los componentes. Notar que, existen diferentes aplicaciones de este tipo de algoritmo, por lo que se puede considerar los arcos o los vertices como los componentes mencionados anteriormente.

La actualización de las feromonas en el grafo, generalmente ocurre después de haber construido una ruta completa. De esta manera, para rutas más cortas, se deposita una mayor cantidad de feromonas y, en caso contrario, una menor cantidad. La cantidad de feromonas que se deposita lo dictamina el diseñador, pero generalmente se utiliza el siguiente formato:

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{Q}{L_k} & \text{si la hormiga utiliza el se mueve del nodo } i \text{ al nodo } j \text{ en su recorrido} \\ 0 & \text{en caso contrario} \end{cases}$$

donde Q es una constante y L_k es la longitud del recorrido que realizo una k-ésima hormiga.

De esta manera, al considerar que una cantidad $k \geq 1$ de hormigas realizarán este proceso. Se puede observar la creación de la huella de feromonas, al considerar los componentes del grafo que tienen una mayor cantidad de feromona. Además, se puede asociar el incremento de feromona $\Delta\tau$ en el componente como:

$$\Delta\tau_{ij} = \sum_{i=1}^{k} \Delta\tau_{ij}^{k}$$

Para evitar estancamientos en el proceso de búsqueda, se define un limite superior y un limite inferior de feromonas que pueden haber por componente. Además, también se introduce una tasa de evaporación (ρ) para reducir gradualmente en el tiempo la cantidad de feromonas que hay en un componente. Es decir, la actualización de la feromona τ_{ij} en una iteración $t+1$ esta dada por:

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}$$

Por otro lado, el atractivo de los componentes se determina mediante una probabilidad asociada a las feromonas previamente depositadas por la colonia. Cuanto mayor sea la cantidad de feromonas depositadas, mayor será la probabilidad de que una hormiga artificial seleccione ese componente como su próximo destino. Sin embargo, la elección de un componente por parte de una hormiga artificial no solo depende de las feromonas, sino también de heurísticas locales o conocimiento heurístico específico del problema. Este componente, traduce información específica del problema en una cantidad numérica. Por ejemplo, al considerar el problema de TSP, el diseñador puede definir η (conocimiento heurístico), como una cantidad fija igual a $\frac{1}{d}$, con d igual a la distancia del nodo vecino. Esta cantidad no se modifica durante la ejecución y entrega información asociada a la distancia de los nodos, de esta manera, distancias mayores tienen un menor valor y distancias pequeñas un mayor valor.

Luego, se define la probabilidad que tiene una hormiga k, de ir de un nodo i a un nodo j en el grafo en la iteración t, como:

$$p_{ij}^{k}(t) = \frac{\tau_{ij}(t)^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{j \in \text{vecinos}_k} \tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}$$

Donde, se considera a los nodos vecinos que son validos para la resolución del problema. Por otro lado, α y β son parámetros que controlan la importancia relativa de la feromona frente a el conocimiento heurístico. Por lo tanto, la atractividad de los componentes es un compromiso entre el conocimiento heurístico, que en este caso, indica que las ciudades cercanas deben elegirse con alta probabilidad, y la cantidad de feromona en él en una iteración t, que indica que si ha habido mucho tráfico en el componente, entonces es altamente deseable.

Finalmente, en el algoritmo ACO, las k hormigas (colonia) construyen una solución por iteración. Es decir, en cada paso de su iteración, se liberan las k hormigas artificiales que siguen el comportamiento descrito anteriormente. Esta colonia de hormigas artificiales se desplaza a través de soluciones parciales del problema. Conforme avanzan, van construyendo de forma incremental una solución, evaluando y modificando las feromonas en los componentes utilizados. Estas feromonas actúan como guías para las futuras hormigas, generando así una optimización basada en el comportamiento observado en las hormigas reales.

2.2. Sintonizadores de Parámetros

Los sintonizadores se pueden definir de manera general como:

Definición 1 *Dada una metaheurística A con parámetros $P = \{n_1, ..., n_k\}$ cuyos valores pertenecen respectivamente a los dominios $D = \{D_1, ..., D_k\}$, un conjunto de instancias del problema $I = I_{train} \cup I_{test}$, una métrica de costo C y una cantidad máxima de recursos b_{max} para sintonizar*

A. Sea Θ el conjunto de posibles configuraciones, con $\theta_i = \{n_{i1}, ..., n_{ik}\}$ una instanciación completa de parámetros, donde cada $n_{im} \in D_m$ y $\theta_i \in \Theta$. Entonces el objetivo es encontrar la mejor configuración $\theta_i \in \Theta$ según la métrica C para A considerando I_{train}, de manera que se minimice C en I_{test}, teniendo en cuenta la restricción de recursos máxima b_{max}.

La métrica de costo C a menudo se basa en el tiempo de ejecución requerido para resolver una instancia del problema, como también en la calidad de la solución obtenida por el algoritmo con esos parámetros (Hutter y cols., 2011). En cambio, la cantidad de recursos b_{max} se asocia a un límite de tiempo o un número máximo de evaluaciones del algoritmo objetivo.

En la literatura, se encuentran diversos enfoques y métodos para llevar a cabo la sintonización de parámetros. Sin embargo, para todos ellos es necesario definir conjuntos finitos de valores para cada dominio de los parámetros que se desean sintonizar, así como un conjunto finito inicial de candidatos a configuración. A continuación, se presentarán algunos de los métodos más relevantes en los últimos tiempos y para este trabajo en particular.

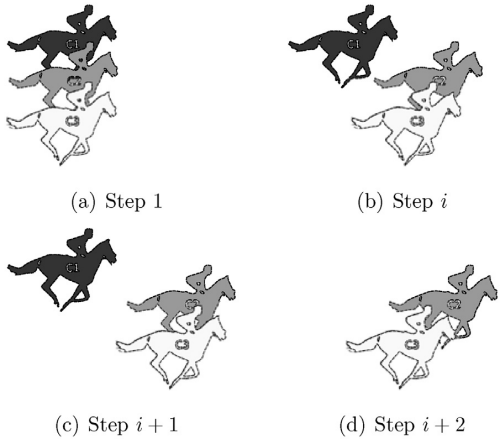
2.2.1. Friedman race

Friedman race (F-race) es un método basado en la iteración de carreras, que se presenta en (Birattari, Stützle, Paquete, Varrentrapp, y cols., 2002). Fue diseñado especialmente para sintonizar algoritmos de búsqueda estocástica. Se decide implementar este acercamiento para mitigar los desafíos que surgen al utilizar un acercamiento por fuerza bruta[3], estos son: (1) definir el número de instancias I en que se ejecutará el algoritmo, (2) fijar el número de ejecuciones que se deben realizar para alcanzar una configuración adecuada y (3) utilizar los mismos recursos computacionales (ineficiencia) en todas las configuraciones, independiente de si el rendimiento de esta fue adecuado o no. F-race es un proceso iterativo, que en cada paso evalúa a un conjunto finito $\Theta_m \subseteq \Theta$ de posibles configuraciones y va descartando los candidatos menos prominentes de dicho conjunto tan pronto se reúna suficiente evidencia estadística para demostrarlo. La Figura 6, se muestra el proceso anterior, en donde se observa que en un step inicial, todos los candidatos comienzan desde la misma posición y a medida que se avanza en la carrera (paso i + n), se junta evidencia que el primer candidato no es bueno por lo que se descarta. Para ello, se basa en la prueba de Friedman, un método estadístico para probar hipótesis también conocido como análisis bidireccional de varianza de Friedman (ANOVA) por rangos. Esta es una prueba no paramétrica[4] para encontrar diferencias entre los candidatos a configuración a través de las múltiples instancias.

Para dar un contexto de la prueba, en primer lugar, se definen los costos asociados a cada configuración $\theta_j \in \Theta_m$ con la notación $c^k(\theta_j, I)$, donde k indica el número de instancias de I en la

[3]Un acercamiento por fuerza bruta, implica la búsqueda secuencial uno a uno de una configuración hasta recorrer todo el espacio de búsqueda o se cumpla algún criterio de termino.

[4]No paramétrico significa que la prueba no asume que los datos provienen de una distribución particular (ej. la distribución normal).



(a) Step 1 (b) Step i

(c) Step $i+1$ (d) Step $i+2$

Figura 6: Ilustración del proceso de eliminación de candidatos en f-race
Fuente: ("A beginner's guide to tuning methods", 2014).

cuál se ha evaluado la configuración θ_j. Por ejemplo, si se evalúa el algoritmo objetivo con una configuración θ_j en dos instancias diferentes, se tiene:

$$c^2(\theta_j, I) = (c(\theta_j, I_1), c(\theta_j, I_2))$$

De esta manera, para k evaluaciones se tienen k costos asociados a una configuración. Por lo que, se utilizará el l-ésimo elemento con costo $c(\theta_j, I_l)$ para indicar el mejor costo encontrado hasta el momento para dicha configuración.

Luego, si se tiene que el algoritmo F-race alcanza un paso i en su iteración, con un conjunto de candidatos $\Theta_{i-1} \subseteq \Theta_m$. Se tienen $n = |\Theta_{i-1}|$ posibles configuraciones en carrera. Por lo tanto, el test de Friedman asume que los costos observados son n variables i-variadas mutuamente independientes llamados bloques $(c^i(\theta_1, I_l), c^i(\theta_2, I_l), ..., c^i(\theta_n, I_l))$. Esto dado que se ha evaluado en i instancias las n posibles configuraciones. De esta manera, cada bloque representa los resultados computacionales de cada candidato a configuración en su iteración i.

Dentro de cada bloque, se clasifica con un rango los costos asociados $c^i(\theta, I_l)$, desde el más pequeño hasta el más grande, es decir, desde 1 a n, y en casos de empate, se utiliza el promedio de los rangos. Ahora, si se considera que para cada configuración $\theta_j \in \Theta_{i-1}$, R_{lj} sea el rango asociado a θ_j en el bloque l, y $R_j = \sum_{l=1}^{k} R_{lj}$ la suma de los rangos sobre todas las instancias utilizadas hasta el momento $(I_1, I_2, ..., I_i)$, entonces el test de Friedman considera la siguiente estadística:

$$T = \frac{(n-1)\sum_{j=1}^{n}(R_j - \frac{k(n+1)}{2})^2}{\sum_{l=1}^{k}\sum_{j=1}^{n} R_{lj}^2 - \frac{kn(n+1)^2}{4}}$$

Bajo la hipótesis nula de que todas las posibles clasificaciones de los candidatos dentro de cada bloque son igualmente probables. Se considera T aproximadamente χ^2 con $n-1$ grados de libertad. Por lo cual, si la T observada excede el cuartil $1-\alpha$ de tal distribución, se rechaza la hipótesis nula, con un nivel de significancia α. Al rechazar la hipótesis nula, se acepta la hipótesis de que al menos un candidato tiende a rendir mejor que al menos otro.

Cuando sucede esta situación, se realizan comparaciones por pares entre las configuraciones de parámetros y la configuración de parámetros mejor clasificada. Todos los candidatos que resultan significativamente peores que la configuración mejor clasificada se descartan y no aparecerán en la siguiente iteración Θ_k.

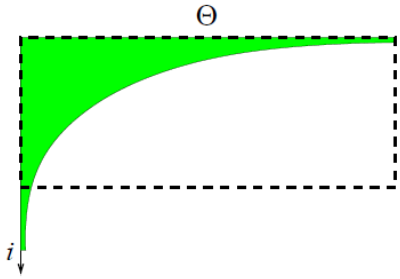


Figura 7: Comparación de uso de recursos computacionales entre acercamiento por fuerza bruta y F-race.
Fuente: (Birattari y cols., 2002).

La eliminación de candidatos deficientes acelera el procedimiento general de sintonización y permite evaluar configuraciones prometedoras en más instancias, como también obtener estimaciones más confiables de su comportamiento. Esto principalmente ya que se van a considerar menos candidatos a medida que existe una mayor cantidad de iteraciones y, a al contrario de su versión en fuerza bruta, se libera el uso de memoria y se maneja de mejor manera los recursos disponibles. En la figura 7 es posible notar está dinámica al considerar el eje y como el número de iteraciones que lleva el algoritmo y Θ como el conjunto de candidatos a solución que aún están carrera. De esta manera la superficie del rectángulo discontinuo representa la cantidad de cálculo realizado por fuerza bruta, mientras que el área sombreada la de F-race.

2.2.2. Relevance Estimation and Value Calibration

En el trabajo realizado en (Nannen y Eiben, 2007b), se presenta el método Relevance Estimation and Value Calibration (REVAC) para sintonizar parámetros. REVAC tiene como principal objetivo ayudar en la sintonización de parámetros en algoritmos evolutivos. Este método esta basado en la teoría de la información para medir la relevancia de los parámetros. Es decir, en lugar de estimar el rendimiento del algoritmo para diferentes valores de parámetros o rangos de valores, se estima el rendimiento esperado del algoritmo cuando los valores de los parámetros son escogidos de una función de densidad de probabilidad (PDF por sus siglas en ingles) C con una entropía de

Shannon maximizada. El concepto básico de entropía en teoría de la información fue introducido en (Shannon, 1948) y se asocia a la información faltante promedio de una fuente aleatoria o, en otras palabras, a la incertidumbre que existe en un experimento. En este contexto, la entropía de Shannon maximizada es un indicador sobre la relevancia que tienen los parámetros, en donde se mide cuánta información es necesaria para alcanzar un cierto nivel de rendimiento y cómo se distribuye esta información entre los diferentes parámetros. Bajo este concepto, REVAC tiene por objetivo:

- Configurar la entropía de la distribución C lo más alta posible para un rendimiento dado.
- Fijar el rendimiento esperado del algoritmo objetivo lo más alto posible.

Sea A un algoritmo evolutivo con k parámetros, este método busca refinar iterativamente la distribución conjunta $C(\vec{\theta_j})$ sobre los posibles vectores de parámetros $\vec{\theta_j} = \{n_{j1}, ..., n_{jk}\}$. REVAC comienza desde una distribución uniforme C^0 sobre el espacio inicial de búsqueda $\Theta = d_1 \times d_2 \times \cdots \times d_k$ y se mueve en el espacio de búsqueda con una probabilidad cada vez mayor a regiones de Θ que logren aumentar el rendimiento esperado del algoritmo objetivo. Sin embargo, también busca suavizar lo mejor posible la distribución C al maximizar la entropía.

$$X = \begin{array}{c|cccc} & d_1 & d_2 & \dots & d_k \\ \hline \vec{\theta_1} & n_{11} & n_{12} & \dots & n_{1k} \\ \vec{\theta_2} & n_{21} & n_{22} & \dots & n_{2k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \vec{\theta_m} & n_{m1} & n_{m2} & \dots & n_{mk} \end{array}$$

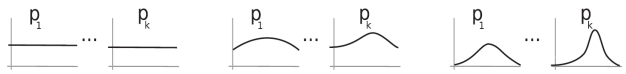
Tabla 1: Tabla X de m vectores con k parámetros.
Fuente: Elaboración propia.

Para entender como opera este método, es necesario considerar 2 perspectivas sobre un conjunto finito de m posibles vectores de parámetros de A (ver tabla 1). Por un lado, desde la perspectiva horizontal, se tienen m posibles vectores candidatos que resultan en $X = \{\vec{\theta_1}, \dots, \vec{\theta_m}\}$. Mientras que, desde la perspectiva vertical, cada columna i de X tiene m valores de cada dominio $d_1, \dots, d_k$. Con estos valores se puede definir una función de densidad marginal $D(x)$ para cada parámetro i. Para ello, se normaliza el intervalo que se encuentran los m valores y se considera la densidad sobre $m + 1$ intervalos entre 2 vecinos cualquiera (incluyendo limites) x_a, x_b como $D(x) = \frac{m+1}{|x_a - x_b|}$ con $\int_0^1 D(x) = 1$. Obtener el diferencial de entropía (Shannon) se traduce en resolver:

$$H(D(x)) = -\int_0^1 D(x) \log D(X)$$

Notar que con H(D) = 0 se representa la distribución uniforme sobre [0, 1] y que mientras menor sea la entropía, la curvatura de la distribución será más definida (cima más aguda). En este caso, la estrechez de los picos se considera un indicador de la importancia del valor del parámetro en el rendimiento del algoritmo. Por lo anterior, es posible asociar las k columnas de la tabla 1 como k funciones de densidad marginales que resulta en $X = \{D_1, \ldots, D_k\}$.

De esta manera, refinar iterativamente la distribución conjunta $C(\vec{\theta_j})$ se puede ver como la función de mejorar iterativamente una tabla inicial X_0 que se extrajo de la distribución uniforme sobre Θ. Por lo que se debe generar una nueva tabla X_{t+1} a partir de un X_t dada, que puede describirse tanto desde la perspectiva horizontal como desde la vertical. La perspectiva horizontal se relaciona directamente con la mejora en el rendimiento esperado del algoritmo objetivo, mientras que la perspectiva vertical se asocia a la maximización de la entropía de Shannon.



(a) Step 1 (b) Step i (c) Final step

Figura 8: Ilustración del proceso de mejora iterativa de la función de densidad de probabilidad C
Fuente: ("A beginner's guide to tuning methods", 2014).

En la figura 8, se puede ver una ilustración sobre como cambian las diferentes distribuciones marginales de cada parámetro i a medida que se avanza en las iteraciones, esto refleja que al moverse a zonas con un mayor rendimiento esperado en el algoritmo, algunos parámetros obtienen una mayor importancia en el resultado, por ejemplo, P_k se considera más importante que P_1 en el paso final. Y esto a su vez, genera que tengan una entropía más negativa, sin embargo, al maximizar dicha entropía ya se considera lo más suave posible que puede estar dicha distribución en el rendimiento dado. Por lo que, el refinamiento se traduce directamente en los dos objetivos mencionados en un principio.

2.2.3. ParamILS

El método ParamILS fue propuesto en (Hutter, Hoos, y Stützle, 2007) y esta basado en el procedimiento manual para encontrar una configuración de parámetros adecuada. En general, el procedimiento manual consta de: (1) comenzar desde una configuración arbitraria en el algoritmo objetivo, (2) experimentar con dicha solución al modificar uno de sus parámetros y aceptar dicha modificación cuando exista una mejora y (3) repetir el paso 2 hasta que no existan mejoras en ningún parámetro. Este tipo de procedimiento es análogo a ejecutar manualmente una búsqueda local en el espacio de búsqueda Θ. En concreto, corresponde a un procedimiento iterativo de primera mejora (Hill Climbing) en donde se utiliza una función objetivo que cuantifica el rendimiento logrado por el algoritmo objetivo dada una configuración y considera como vecindario en cada paso de su iteración, las posibles configuraciones que se generan al modificar el valor de un único parámetro a la vez. Sin embargo, este procedimiento se detiene tan pronto como alcanza

un óptimo local[5]. Es por ello, que se plantea el uso de un acercamiento más sofisticado conocido como búsqueda local iterada (ILS por sus siglas en inglés).

ILS es una modificación de los métodos de búsqueda local, en donde se genera una cadena de óptimos locales al iterar sobre un ciclo que consta de: (1) una perturbación en la solución para escapar de óptimos locales, (2) un procedimiento de búsqueda local secundario y (3) un criterio de aceptación para decidir si mantener o rechazar una nueva solución candidata. En este contexto, ParamILS es un método que utiliza ILS para encontrar una configuración de parámetros en el espacio Θ. Para ello, emplea una combinación de configuraciones predeterminadas y aleatorias para su candidato inicial y bajo la estructura de ILS: (1) utiliza un número fijo (s) de movimiento aleatorios como perturbación para escapar de óptimos locales, (2) considera una primera mejora iterativa como procedimiento de búsqueda local secundario y (3) siempre acepta configuraciones de parámetros mejores o igualmente buenas en rendimiento, pero reinicia la búsqueda de forma al azar con una probabilidad fijada de $p_{reinicio}$. Además, como se menciona anteriormente, se basa en un vecindario de intercambio único, esto quiere decir, que se considera cambiar solo un parámetro a la vez.

Cabe destacar que, en la practica, existen muchos casos en donde el algoritmo objetivo presenta parámetros que solo adquieren relevancia cuando otros parámetros de "nivel superior" toman ciertos valores. A estos casos, se les denomina a parámetros condicionales y ParamILS se ocupa de estos parámetros condicionales al excluir todas las posibles configuraciones de la vecindad de una configuración θ_i en el que se difiera solo por el valor de un parámetro condicional que no es relevante para θ_i.

2.2.4. Sequential Model-based Algorithm Configuration

El estudio realizado por (Hutter y cols., 2011) propone a Sequential Model-based Algorithm Configuration (SMAC), un nuevo enfoque para la sintonización de parámetros basado en modelos. En primer lugar, se propone utilizar la optimización basada en modelos secuenciales (SMBO, por sus siglas en inglés) para abordar el problema general de la sintonización de parámetros. Este enfoque implica la creación de un modelo de regresión de la función objetivo, el cual busca representar la relación entre los parámetros a sintonizar y su rendimiento. El objetivo de este modelo es predecir el rendimiento que tendrá el algoritmo objetivo para una determinada configuración de parámetros. Para lograrlo, el modelo se ajusta a un conjunto finito de entrenamiento $\Theta_m \subseteq \Theta$ para que pueda predecir el rendimiento esperado de una nueva configuración de parámetros. Luego, se utiliza un algoritmo de selección de parámetros basado en la incertidumbre del modelo para elegir la siguiente configuración de parámetros a evaluar. En otras palabras, selecciona la próxima configuración de parámetros en función de su probabilidad de ser la mejor configuración posible. De esta manera, SMBO se puede asociar a un proceso que consta de dos partes donde: (1) se utiliza un modelo probabilístico para predecir qué combinaciones de parámetros son más prometedoras y (2) se utiliza esta información para guiar la búsqueda hacia las combinaciones de parámetros más

[5]Configuración de parámetros que es mejor o igual en rendimiento a todas las configuraciones en su vecindario

prometedoras. Este proceso se repite varias veces hasta encontrar una configuración adecuada.

SMBO, al estar basado en la literatura de optimización de funciones de "caja negra" [6], ha heredado una serie de limitaciones. Estas incluyen un enfoque en algoritmos deterministas, un uso de diseños experimentales iniciales costosos, la dependencia de modelos computacionalmente costosos y la suposición de que todas las ejecuciones del algoritmo objetivo tienen los mismos costos de ejecución. Además, SMBO tiene tres limitaciones clave que impiden su uso para resolver problemas de sintonización de parámetros generales: solo admite parámetros numéricos, solo optimiza el rendimiento del algoritmo para instancias individuales (no se utiliza un conjunto I de instancias) y carece de mecanismos para terminar antes de tiempo las ejecuciones de algoritmos con bajo rendimiento.

En este contexto, se presenta el método SMAC una extensión de SMBO que busca eliminar sus dos primeras limitaciones y, por lo tanto, hacerlo aplicable a problemas más generales de sintonización de parámetros. Para ello, SMAC aborda estas limitaciones al utilizar una serie de técnicas:

1. **Modelo de regresión basado en árboles:** SMAC utiliza un modelo de regresión basado en árboles para modelar el rendimiento de las diferentes configuraciones de parámetros. Este modelo es capaz de manejar datos no lineales y categóricos, lo que le permite adaptarse a una amplia variedad de problemas.

2. **Optimización bayesiana secuencial:** SMAC utiliza un proceso de optimización bayesiana secuencial para actualizar el modelo estadístico con los nuevos datos recopilados. Este proceso permite que el modelo ajustado se adapte a medida que se recopila más información y se hace más precisa la predicción del rendimiento de nuevas configuraciones.

3. **Criterio de selección de configuración:** SMAC utiliza un criterio de selección de configuración basado en la probabilidad de mejora esperada de la función objetivo. Este criterio tiene en cuenta tanto la media como la varianza de los rendimientos previos de las diferentes configuraciones, lo que permite que SMAC seleccione configuraciones con un alto potencial de mejora.

En general, SMAC aborda las limitaciones de SMBO al permitir una exploración más amplia del espacio de búsqueda, utilizando modelos más flexibles y precisos para modelar el rendimiento, y actualizando continuamente el modelo con los nuevos datos recopilados. Esto permite que SMAC encuentre configuraciones de parámetros con un mejor rendimiento en la función objetivo en menos evaluaciones que SMBO.

[6] Las optimizaciones de funciones de caja negra son un tipo de problema de optimización en el que se busca encontrar los valores óptimos de una función objetivo desconocida, para la cual no se tiene información sobre su forma o propiedades analíticas. La función objetivo se considera una caja negra porque su comportamiento interno es desconocido y solo se pueden observar sus entradas y salidas (Pelikan, 2005).

Cabe destacar que, estos métodos aún no implementan un criterio de terminación anticipada para las ejecuciones de algoritmos objetivos de bajo rendimiento como, por ejemplo, el mecanismo de límite adaptativo de PARAMILS (Hutter, Hoos, Leyton-Brown, y Stützle, 2009). Por lo tanto, hasta ahora se espera que tenga un desempeño deficiente en algunos escenarios de configuración con grandes tiempos de captura.

2.2.5. Iterated race

El método Iterated race (I-race) fue introducido en (López-Ibáñez, Dubois-Lacoste, Pérez Cáceres, Birattari, y Stützle, 2016). Al igual que F-race (Birattari y cols., 2002), también utiliza una estrategia iterativa basada en carreras para encontrar la mejor configuración de algoritmo para un conjunto de instancias de prueba. En general, se puede definir la iteración de carreras como un método que consta de tres pasos: (1) considerar un muestra inicial de candidatos a configuración de acuerdo con una distribución particular, (2) seleccionar los candidatos más prominentes de dicha muestra a través de una competencia de carreras, y (3) actualizar la distribución de muestreo para sesgar los resultados hacia mejores configuraciones. Luego, estos tres pasos se repiten hasta que se cumple un criterio de término.

En particular, este método propone una implementación en donde cada parámetro a configurar tiene asociada su propia distribución. De esta manera, se considera una distribución normal truncada para parámetros numéricos y una distribución discreta para parámetros categóricos. Además, se consideran las restricciones y condiciones entre los parámetros, es decir, al muestrear el espacio (dominio) de parámetros, los parámetros se deben considerar en un orden determinado por la dependencia de las condiciones. Los parámetros no condicionales se consideran primero en la configuración y, luego, los parámetros condicionales a ellos (en caso de cumplir con la condición).

Por otro lado, al seguir la lógica planteada anteriormente sobre las carreras iteradas, en este método se plantea: (1) Considerar una nueva muestra de configuraciones $\Theta_i \subseteq \Theta$ obtenidas de una distribución en particular en cada paso de su iteración, (2) Encontrar las mejores configuraciones de esta nueva muestra por medio de carreras y (3) Actualizar las distribuciones asociadas a cada parámetro.

La actualización de las distribuciones consiste en modificar la media y la desviación estándar en el caso de la distribución normal, o los valores de probabilidad discreta en las distribuciones discretas. Con ello, se tiene por objetivo sesgar las distribuciones para aumentar la probabilidad de muestrear, en futuras iteraciones, los valores de los parámetros en las mejores configuraciones encontradas hasta el momento.

En caso de las carreras realizadas entre las configuraciones, estás siguen un comportamiento similar a lo explicado en F-race. Sin embargo, existen 2 alternativas para seleccionar qué configuraciones deben descartarse durante la carrera: (1) considerar la prueba de Friedman descrita en F-race y (2) se plantea el uso de la prueba T pareada como opción alternativa. Ambas pruebas

estadísticas se utilizan con un nivel de significancia predeterminado de 0,05.

Por último, se incorpora un mecanismo de "reinicio suave" para evitar una convergencia temprana. Este caso ocurre cuando, no existe diversidad en las configuraciones muestreadas, por lo cuál se converge rápidamente y las configuraciones candidatas recién generadas serán muy similares a las ya probadas. Esto genera un estancamiento en un óptimo local sin poder diversificar en el espacio de búsqueda, al solo tener asociado en la muestra configuraciones de parámetros con valores similares.

2.2.6. Evolutionary Calibrator

Evolutionary Calibrator (EVOCA) es un método basado en los algoritmos evolutivos y se presenta en (Riff y Montero, 2013). El método EVOCA utiliza cromosomas como representación de una configuración candidata. Un cromosoma es un conjunto de genes y, en este contexto, cada gen busca representar uno de los parámetros que se desean sintonizar en el algoritmo objetivo. El cromosoma se muestra como una cadena de caracteres, donde cada carácter corresponde a un parámetro específico, y su valor se encuentra asociado a su dominio. En la figura 9 se muestra un ejemplo del cromosoma utilizado, donde cada V_i con $i = 1, ..., N$ corresponde a un valor para cada parámetro a sintonizar, dicho valor se obtiene de los dominios respectivos de cada parámetro y su composición representa una posible configuración que puede utilizar el algoritmo objetivo.

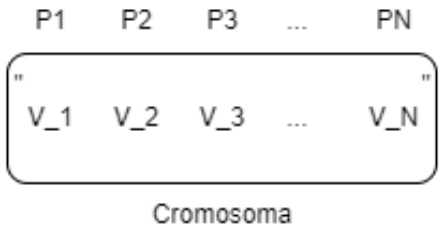


Figura 9: Ilustración de representación utilizada en EVOCA.
Fuente: Elaboración propia.

Con esta representación, se procede con el paso (1) mencionado anteriormente. Para ello, EVOCA define los siguientes conceptos:

- Dominio inicial

  Definición 2 *Sean P_j los parámetros numéricos a sintonizar del algoritmo objetivo, donde $P_j \subseteq P$ y sean los dominios D_j de los parámetros P_j conjuntos finitos entre el rango $\{d_{il}, d_{iu}\} \forall i = 1, ..., k$ con una precisión de $p_i = \{p_1, p_2, ..., p_k\}$ para cada uno de ellos. Se definen sus dominios iniciales como:*

$$ID_{P_j} = \{d_{il}, d_{il} + p_i, d_{il} + 2 \cdot p_i, \cdots, d_{iu}\} \forall i = 1, ..., k$$

Por ejemplo, si se desea sintonizar la probabilidad de cruzamiento uniforme en un algoritmo genético, se debe definir un conjunto de valores finitos entre 0 y 1. Por lo cual, si se establece que la precisión para dicho parámetro es igual a 0.1, se tiene:

$$ID_{cruzamiento} = \{0.0, 0.1, 0.2, \cdots, 0.9, 1.0\}$$

Notar que para los parámetros categóricos y los parámetros con dominios discretos presentan un conjunto finito de posibles valores. Por lo cuál, no es necesario definir un nuevo conjunto para estos casos.

- Cardinalidad

  Definición 3 *Se define la cardinalidad de los dominios iniciales como:*

  $$IS_{P_i} = Card(ID_{P_i})$$

- Tamaño de la población

  Definición 4 *Sea k el número de parámetros a sintonizar en el algoritmo objetivo. Donde los tamaños de sus dominios iniciales estará dado por $IS_{P_i} \forall i = 1, ..., k$, se define el tamaño de la población como:*

  $$popsize = max_{\{i=1,...,k\}} IS_{P_i}$$

EVOCA ofrece dos opciones para configurar el tamaño de su población inicial. La primera opción, contempla que el diseñador fije un valor para el tamaño de la población previo al proceso de sintonización, mientras que la segunda opción utiliza la función “popsize”. Al utilizar esta última opción, EVOCA determina el tamaño de la población inicial en función del tamaño de los dominios iniciales (IS) de los parámetros que se van a sintonizar. Se tiene por objetivo incluir todos los valores permitidos para cada parámetro, de manera independiente, en la población inicial. Para los parámetros categóricos y los parámetros con dominios discretos, esto se logra directamente. Sin embargo, en el caso de los parámetros numéricos (reales), se debe definir un intervalo de evaluación de acuerdo con un valor de precisión (p_i) establecido por el diseñador. En consecuencia, el tamaño de la población se establece como el tamaño del dominio más grande entre todos los parámetros a sintonizar.

Para inicializar la población, EVOCA realiza lo siguiente:

1. Para cada parámetro $j \in P$, se genera una lista de valores extraídos de manera aleatoria de sus dominios iniciales (ID_j).

2. Produce una configuración candidata extrayendo de manera cíclica un valor de las listas de cada parámetro.

3. Repite el paso anterior, hasta crear $popsize$ configuraciones en la población inicial.

Luego, procede a determinar cuál es la calidad de cada configuración en la población inicial. Para ello, se realiza una prueba de rendimiento con cada una de ellas utilizando R semillas aleatorias. Esto quiere decir que, para cada solución candidata, se evaluará su rendimiento en R instancias aleatorias y su calidad estará dada por el promedio obtenido en estas evaluaciones.

Una vez que la población ha sido inicializada, EVOCA procede a llevar a cabo el paso (2) y (3). Para ello, se aplican dos nuevos operadores:

1. Cruzamiento uniforme + ruleta (roulette wheel)

   Este operador consiste en la generación de un nuevo individuo mediante el cruzamiento de genes entre todas las configuraciones presentes en la población. Para ello, se utiliza una ruleta que selecciona, basándose en la calidad (fitness) de las soluciones candidatas, qué configuración proporcionará el valor para cada gen del nuevo individuo. La ruleta se genera en función de la calidad de las soluciones, lo que significa que un individuo con una mayor calidad tendrá una mayor probabilidad de ser seleccionado, mientras que uno con una menor calidad tendrá menos probabilidades. La figura 10 muestra una ilustración del procedimiento descrito anteriormente, en donde se observa que para individuos con una mayor aptitud se presenta una mayor probabilidad de ser elegidos al tener una mayor envergadura en la ruleta. De esta manera, para cada gen del nuevo individuo, se realiza una selección mediante la ruleta para obtener una configuración de la población y “heredar” el valor de su gen. Por último, se evalúa la calidad de este nuevo individuo con las R semillas aleatorias y se reemplaza al individuo de peor calidad en la población.

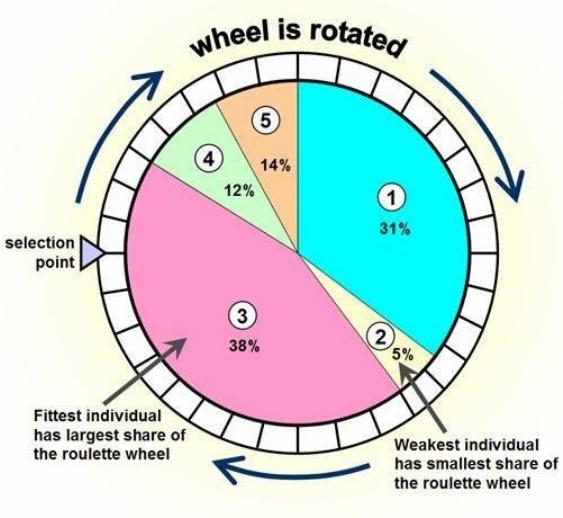


Figura 10: Procedimiento de ruleta.
Fuente: (Aarti, 2019).

2. Mutación + Hill Climbing de primera mejora

   El operador de mutación toma una copia del descendiente generado por el operador de cruzamiento e intenta mejorarlo al modificar uno de sus valores de parámetro. Para ello, se selecciona aleatoriamente uno de los valores y se implementa un procedimiento de tipo "Hill Climbing"de primera mejora. En este procedimiento, se elige al azar un nuevo valor dentro del dominio original del parámetro. El desempeño del individuo mutado se evalúa utilizando R semillas aleatorias y, si el cambio del gen produce alguna mejora, se reemplaza al segundo peor individuo en la población actual. En otras palabras, si el descendiente generado por el operador de mutación tiene un mejor desempeño que el generado por el operador de cruzamiento, este nuevo individuo también se incluye en la población. En caso de obtener un desempeño inferior, este procedimiento se repite n veces, siendo n la cantidad de valores en el dominio de dicho parámetro (IS_{P_i}). Esto tiene como objetivo mejorar la calidad de la población mediante la exploración de nuevos valores de parámetros que no estaban inicialmente presentes en la población.

Finalmente, se lleva a cabo un ciclo para encontrar una configuración adecuada de parámetros, el cual finaliza según el criterio de término propuesto por el diseñador. En cada iteración del ciclo, siempre se aplican ambos operadores mencionados anteriormente, por lo que no hay una probabilidad asociada a su ocurrencia y, en consecuencia, como máximo dos individuos en la población pueden ser reemplazados.

El principal objetivo de EVOCA es proporcionar una herramienta simple para la sintonización de parámetros, en lugar de buscar un sintonizador aplicable a problemas generales, que ha sido la dinámica predominante en años anteriores. Esto permite a los diseñadores de metaheurísticas

encontrar una configuración adecuada de parámetros sin necesidad de tener un amplio conocimiento sobre métodos de sintonización. En particular, este enfoque plantea una sintonización que permite el ajuste de parámetros tanto categóricos como numéricos, lo que brinda flexibilidad en la configuración de parámetros para adaptarse a diferentes tipos de problemas y dominios de aplicación.

2.3. Paralelismo

El paralelismo en la computación se refiere a su capacidad de llevar a cabo múltiples tareas o cálculos de forma simultánea. En lugar de ejecutar las instrucciones secuencialmente, se permite que varias instrucciones o tareas se realicen al mismo tiempo mediante el paralelismo, lo cual puede acelerar considerablemente el rendimiento y mejorar la eficiencia en comparación con un procesamiento secuencial. Esta técnica es ampliamente utilizada en distintos campos de la computación, tales como la programación en multiprocesadores, la computación distribuida, el procesamiento de grandes volúmenes de datos (big data), el aprendizaje automático (machine learning) y la simulación de sistemas complejos (Hennessy y Patterson, 2011).

Para aprovechar al máximo el paralelismo, se requiere el diseño de algoritmos y programas específicos que puedan dividir eficientemente las tareas tanto, en unidades paralelas, como en la sincronización y la coordinación entre ellas. Para ello, se debe comprender que existen distintos niveles de paralelismo en las aplicaciones, los cuales se clasifican en dos tipos:

1. **Paralelismo a nivel de datos (DLP)**: surge debido a la presencia de muchos elementos de datos que se pueden operar al mismo tiempo.
2. **Paralelismo a nivel de tareas (TLP)**: surge porque se crean tareas de trabajo que pueden operar de manera independiente y en gran medida en paralelo.

Además, el hardware de los computadores, a su vez, puede aprovechar estos dos tipos de paralelismo en las aplicaciones de cuatro formas diferentes:

1. **Paralelismo a nivel de instrucciones**: aprovecha el paralelismo a nivel de datos a niveles moderados con la ayuda del compilador utilizando técnicas como la segmentación (pipelining) y a niveles medios utilizando técnicas como la ejecución especulativa (speculative execution).
2. **Arquitecturas vectoriales y unidades de procesamiento gráfico (GPUs)**: aprovecha el paralelismo a nivel de datos aplicando una sola instrucción a una colección de datos en paralelo.
3. **Paralelismo a nivel de hilos (thread-level parallelism)**: aprovecha tanto el paralelismo a nivel de datos como el paralelismo a nivel de tareas en un modelo de hardware estrechamente acoplado que permite la interacción entre hilos paralelos.

4. **Paralelismo a nivel de solicitudes (request-level parallelism)**: aprovecha el paralelismo entre tareas ampliamente desacopladas especificadas por el programador o el sistema operativo.

Para comprender las cuatro formas en que el hardware puede admitir el paralelismo a nivel de datos y el paralelismo a nivel de tareas, es necesario comprender la arquitectura de los computadores. Esta se define como el estudio de la organización e interconexión de los componentes de los sistemas computacionales. Los computadores se construyen a partir de diversos módulos o bloques básicos, como memorias, unidades aritméticas, elementos de procesamiento y buses. Estos bloques permiten la construcción de cualquier tipo de computador, desde los más pequeños hasta los supercomputadores más grandes.

El comportamiento funcional de los componentes en diferentes computadores es similar. Por ejemplo, el sistema de memoria se encarga del almacenamiento, la unidad central de procesamiento realiza operaciones y las interfaces de entrada y salida transfieren datos entre el procesador y los dispositivos correspondientes. Sin embargo, sus diferencias radican en la forma en que los módulos se interconectan entre sí, las características de rendimiento de los módulos y el método por el cual el sistema computacional se controla mediante operaciones.

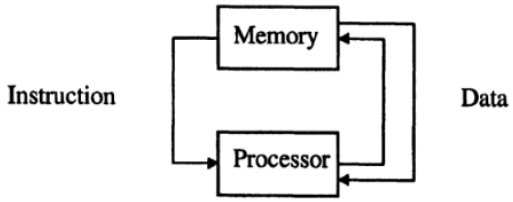


Figura 11: Interconexión memoria-procesador Von Neumann.
Fuente: (Roosta, 2012).

Dentro de los componentes básicos de una arquitectura computacional, dos de ellos desempeñan un papel fundamental: el procesador y la memoria. Uno de los primeros acercamientos para definir su comportamiento fue el modelo de computación de Von Neumann (von Neumann, 1993). El funcionamiento de este modelo se basa en el concepto de almacenamiento secuencial de instrucciones. En la figura 11 se muestra su funcionamiento, donde las instrucciones y los datos se almacenan en la memoria principal en forma de secuencia de bits. El procesador (CPU) recupera una instrucción de memoria, la decodifica para determinar la operación a realizar, y luego ejecuta esa instrucción utilizando los datos correspondientes almacenados en la memoria. Por último, si requiere, modifica o almacena nuevamente en memoria el resultado de dicha operación.

En este modelo establece un flujo de trabajo secuencial en el que la CPU ejecuta un flujo instrucciones (algoritmo) una tras otra, siguiendo un programa almacenado en memoria. A medida que se ejecutan las instrucciones, los datos son transferidos entre la memoria y la CPU a través del bus de datos (Roosta, 2012).

No obstante, este modelo constituye una representación básica de lo que ocurre actualmente, y puede ser ampliado según la clasificación realizada en (Flynn, 1972). Michael Flynn estudió los esfuerzos de la computación paralela y encontró una clasificación simple cuyas abreviaturas todavía se utilizan hoy en día. Observó el paralelismo en las instrucciones y flujos de datos requeridos por las instrucciones en el componente más limitado del multiprocesador, y clasificó todas las computadoras en una de cuatro categorías.

2.3.1. Arquitecturas Paralelas

1. **Flujo de instrucción único, flujo de datos único (SISD, por sus siglas en inglés):**

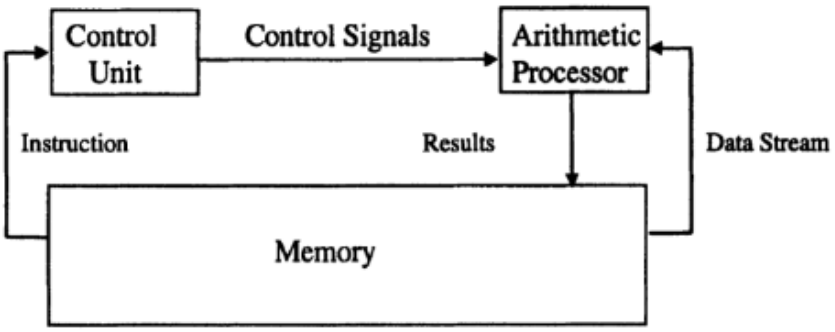


Figura 12: Modelo SISD de computadores
Fuente: (Roosta, 2012).

Un computador de esta clase consta de una sola unidad de procesamiento que ejecuta una instrucción a la vez y obtiene o almacena un elemento de datos a la vez. La figura 12 muestra su comportamiento, en donde exhibe un comportamiento similar al descrito en el modelo de Von Neumann. En este, la unidad de control emite una instrucción obtenida desde la unidad de memoria. Posteriormente, la operación se envía al procesador aritmético, que opera sobre un dato obtenido nuevamente desde la unidad de memoria y lo guarda en caso de ser requerido.

2. **Flujo de instrucción único, múltiples flujos de datos (SIMD, por sus siglas en inglés):**

En esta categoría, un computador consta de una sola unidad de procesamiento que ejecuta una instrucción a la vez, pero tiene más de un elemento de procesamiento de datos. La figura 13 representa el comportamiento de este caso, donde la unidad de control emite una misma instrucción para todos los elementos de procesamiento de datos, los cuales ejecutan esta operación en diferentes elementos. En este escenario, se emplea el concepto de ejecución lock-step, en el cual cada elemento de procesamiento tiene su propio flujo de datos (Roosta, 2012). La ejecución lock-step implica que la misma instrucción se ejecutará simultáneamente “en paralelo” en todos los elementos de procesamiento. Es decir, la misma instrucción se ejecuta en diferentes flujos de datos. Actualmente, este comportamiento

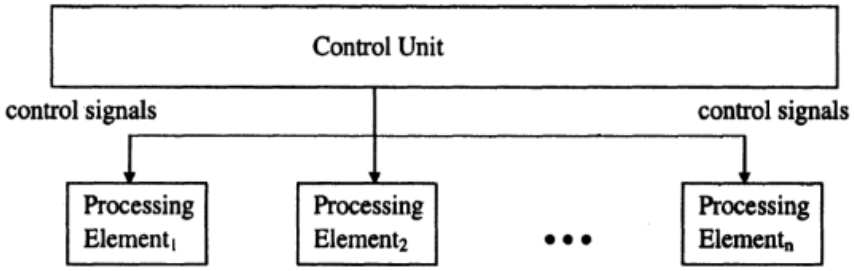


Figura 13: Modelo SIMD de computadores
Fuente: (Roosta, 2012).

se observa directamente en el funcionamiento de las unidades de procesamiento gráfico (GPU).

3. **Múltiples flujos de instrucción, flujo de datos único (MISD, por sus siglas en inglés):**

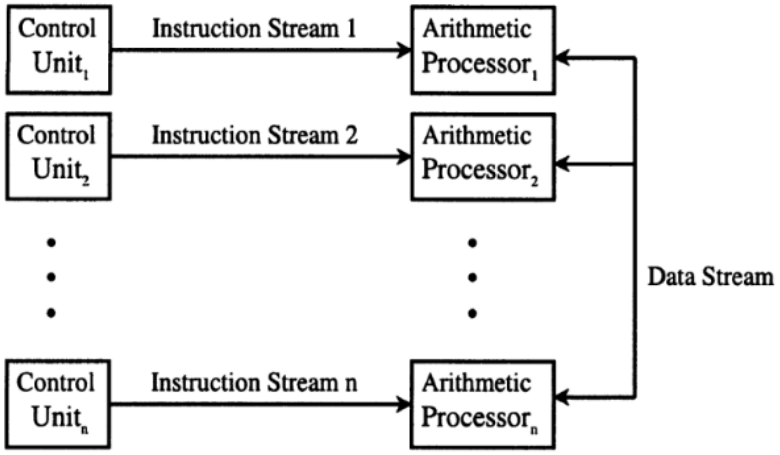


Figura 14: Modelo MISD de computadores
Fuente: (Roosta, 2012).

Las máquinas de esta categoría, presentan N elementos de procesamiento, cada uno con su propia unidad de control y comparten una unidad de memoria común donde residen los datos. La figura 14 muestra su comportamiento, donde existen N flujos de instrucciones y un flujo de datos, lo que permite a estas computadoras ejecutar diferentes programas en el mismo elemento de datos. Esto implica que, en cada paso, un dato recibido de la memoria es operado por todos los procesadores simultáneamente, cada uno de acuerdo con la instrucción que recibe de su control (Akl, 1989).

Hasta la fecha, no se ha construido ningún multiprocesador comercial de este tipo, pero completa esta clasificación simple.

4. **Múltiples flujos de instrucción, múltiples flujos de datos (MIMD, por sus siglas en inglés):**

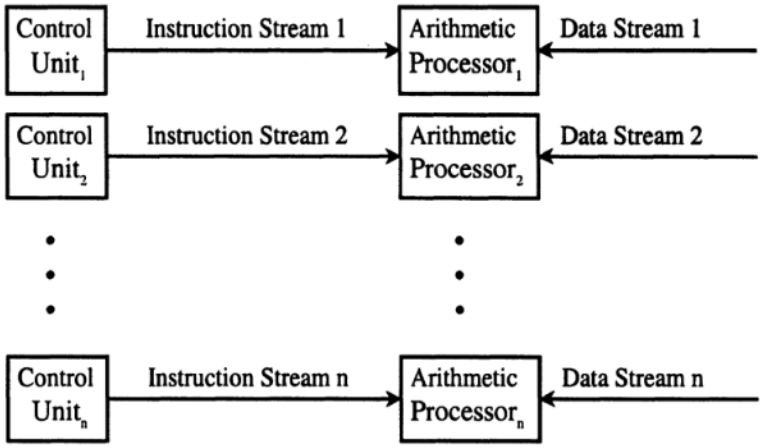


Figura 15: Modelo MIMD de computadores
Fuente: (Roosta, 2012).

Las computadoras de esta clase tienen más de un procesador y cada uno puede ejecutar un programa diferente (flujo de instrucciones múltiples), en su propio elemento de datos (flujo de datos múltiples). En la figura 15 podemos apreciar este comportamiento en donde varias unidades de control emiten sus propias instrucciones sobre procesadores aritméticos diferentes con sus propios datos obtenidos de la unidad de memoria. Por otro lado, al igual que con las computadoras SIMD, la comunicación entre procesadores se realiza a través de una memoria compartida o una red de interconexión. Las computadoras MIMD que comparten una memoria común se denominan multiprocesadores mientras que aquellos con una interconexión se conocen como sistemas distribuidos. La distinción entre ellas, generalmente se basa en la distancia física que separa los procesadores y, por lo tanto, es a menudo un tema subjetivo (Akl, 1989).

Luego de analizar estas diferentes clasificaciones sobre la arquitectura de computadores, es fácil asociar el paralelismo a arquitecturas SIMD y MIMD. Sin embargo, esta taxonomía es un modelo general de lo que ocurre actualmente, ya que muchos procesadores paralelos son híbridos de las clases SISD, SIMD y MIMD. Por ejemplo, en la actualidad es común encontrar computadores de escritorio y notebooks con múltiples núcleos en su procesador (CPU), lo que se puede asociar a una arquitectura MIMD, sin embargo, también presentan incorporado una unidad de procesamiento gráfico (GPU) que actúa como una arquitectura SIMD.

2.3.2. Programación en Multiprocesadores

La programación en multiprocesadores hace referencia al desarrollo de software que se ejecuta en sistemas con múltiples procesadores o núcleos y utiliza un paralelismo a nivel de hilos. En estos sistemas, los procesadores trabajan de forma paralela y se comunican a través de una memoria compartida. Por lo cuál, este tipo de programación se centra en la coordinación de múltiples hilos o subprocesos concurrentes que se pueden ejecutar en diferentes procesadores. La ejecución de estos múltiples hilos concurrentes, se puede entender como múltiples flujos de instrucciones que ocurren de manera independiente y se superponen en el tiempo (Sun Microsystems, Inc., 2008). Es por ello, que se da la ilusión de una ejecución simultánea, no obstante, esto solo ocurre cuando se realiza una ejecución concurrente de exactamente un hilo por procesador. De esta manera, en un comienzo, los hilos se ejecutan de manera simultánea y en paralelo.

La mejora de pasar desde un sistema con único procesador a uno que presenta n procesadores, en teoría, debería proporcionar un aumento de potencia computacional de aproximadamente n veces. Lamentablemente, en la práctica, esto nunca sucede. La principal razón de ello, es que en la mayoría de los problemas computacionales del mundo real no se pueden paralelizar los hilos de manera efectiva sin recurrir a los costos asociados a la comunicación y sincronización entre procesadores (Herlihy y Shavit, 2012). En la programación concurrente, se define la ley de Amdahl para comprender cuál es esta mejora, de pasar de un proceso secuencial a uno paralelo.

Definición 5 *La Ley de Amdahl define la aceleración (Speedup) de una tarea, como la relación entre el tiempo que tarda un único procesador en completar la tarea (medido por un reloj) y el tiempo que tardan n procesadores en completar la misma tarea. Además, se propone que dicha aceleración estará determinada por:*

$$Speedup_N = \frac{T_1}{T_N} = \frac{1}{\frac{F_{parallel}}{N} + F_{sequential}} = \frac{1}{\frac{F_{parallel}}{N} + (1 - F_{parallel})} \tag{1}$$

donde $F_{parallel}$ es la fracción de la tarea que se puede ejecutar en paralelo, $F_{sequential}$ es la fracción de la tarea que se ejecuta de forma secuencial, es decir, no se puede ejecutar en paralelo y N es el número de procesadores utilizados en el sistema.

Además, se define que la eficiencia en la aceleración anterior estará dada por:

$$Efficiency_N = \frac{Speedup_N}{N} \tag{2}$$

La cual podría llegar a ser tan alto como 1, pero en la practica probablemente nunca lo será.

Por otro lado, con esta ley es posible definir la porción paralela ($F_{parallel}$) de la implementación dado los tiempos finales de cada ejecución, con la siguiente formula:

$$F_{parallel} = \frac{N}{N-1} \cdot \frac{T_1 - T_n}{T_1} \tag{3}$$

En la figura 16, se presenta una ilustración visual sobre la ley de Amdahl, donde se observa que a medida que aumenta el número de núcleos utilizados en la implementación paralela, la fracción paralela ($F_{parallel}$) tiende a reducirse, sin embargo, hay una fracción secuencial ($F_{sequential}$) que se mantendrá constaste en la ejecución.

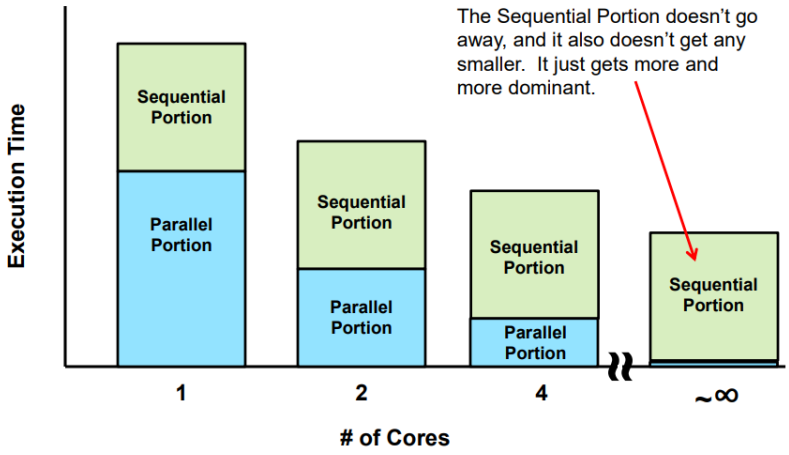


Figura 16: Explicación visual de la ley de Amdahl
Fuente: (Bailey, 2017).

Siempre existirá una fracción de las operaciones totales que es intrínsecamente secuencial y que no puede ser paralelizada, independientemente de los esfuerzos realizados. Esta fracción incluye tareas tales como la lectura de datos, la configuración de cálculos, la lógica de control, el almacenamiento de resultados, entre otras. Por lo tanto, al momento de paralelizar una tarea esta tiende a alcanzar un limite máximo en su aceleración, la cual esta dada por:

$$maxSpeedup_N = \lim_{n\to\infty} Speedup_N = \frac{1}{F_{sequential}} = \frac{1}{1 - F_{parallel}} \tag{4}$$

Al momento de paralelizar una aplicación secuencial, existirán partes significativas de la tarea que pueden determinarse como fácilmente ejecutables en paralelo, ya que no requieren ningún tipo de coordinación o comunicación. Actualmente, no existe alguna métrica o receta que indique cuales son estas partes. Es más, el diseñador debe utilizar su comprensión acumulada sobre la aplicación para saber que partes son las más accesibles o tienen una mejor predisposición para una ejecución en paralelo. Afortunadamente, en muchos casos es evidente cómo encontrar dichas partes (Herlihy y Shavit, 2012).

De todas maneras, el problema más importante que se enfrenta este tipo de programación, radica en cómo lidiar con las partes restantes del programa. Es decir, en como resolver los casos donde las partes de la tarea no se pueden paralelizar fácilmente, ya sea debido a que el programa debe acceder a datos compartidos, requiere de una coordinación y/o comunicación entre los demás subprocesos. El manejo de todos estos casos es lo que llamamos “sincronización”.

La sincronización se refiere a las relaciones entre eventos durante una tarea, ya sea cualquier número de eventos y cualquier tipo de relación (antes, durante, después). A menudo existen restricciones asociadas a la sincronización, como los casos mencionados anteriormente, en donde existe un requisito en el orden de los eventos (Downey, 2016). Algunos ejemplos incluyen:

1. Serialización: El evento A debe ocurrir antes del evento B.
2. Exclusión mutua: Los eventos A y B no deben ocurrir al mismo tiempo.

Existen varios métodos para manejar la sincronización, pero se analizarán tres de ellos debido a su importancia para esta memoria: la exclusión mutua (mutex), las colas de trabajo y las variables de condición.

- **Exclusión mutua (mutex)**

  La exclusión mutua o mutex es quizás la forma más prevalente de coordinación en la programación en multiprocesadores (Herlihy y Shavit, 2012). Este tipo de mecanismo permite la coordinación de la escritura y lectura en la memoria compartida, entre varios subprocesos concurrentes. Para ello, debe garantizar que ocurra una exclusión mutua entre subprocesos al momento de acceder a un mismo recurso compartido. Es decir, solo uno de ellos puede tener acceso al recurso en un momento dado. Puede interpretarse como un candado (lock, en inglés), cuando el mutex esta cerrado, solo uno de los subprocesos tiene permiso para acceder al recurso compartido, si otro subproceso intenta adquirir el mismo recurso mientras el mutex está cerrado, se bloqueará hasta que el mutex esté disponible nuevamente. De esta manera se entrega un permiso exclusivo para el acceso. Una vez que el subproceso ha terminado de utilizar el recurso, se libera el mutex, para permitir que otros subprocesos puedan acceder al recurso compartido de manera segura.

  Generalmente, se denomina a la parte de la tarea que accede a memoria compartida como “sección critica” dado que es allí en donde se pueden producir problemas de sincronización, como las condiciones de carrera, donde múltiples subprocesos acceden al mismo recurso simultáneamente y pueden generar resultados inconsistentes o incorrectos. El uso de mutex ayuda a evitar este tipo de problemas asegurando la coherencia y la integridad de los datos, al garantizar que solo un subproceso pueda acceder al recurso compartido a la vez.

  Aunque existen diferentes maneras de implementar un mutex (lock), hay ciertas propiedades que un buen algoritmo de bloqueo siempre debe cumplir. Sea T_{ai} el intervalo durante el cual un subproceso a ejecuta una sección crítica por i-ésima vez, las propiedades que debe cumplir el mutex son:

1. Exclusión Mutua: Las secciones críticas de diferentes hilos no se superponen en el tiempo, es decir, las secciones criticas nunca se ejecutarán en paralelo por dos hilos diferentes. Para los hilos a y b, y los enteros j y k, se cumple que $T_{ak} \rightarrow T_{bj}$ o $T_{bj} \rightarrow T_{ak}$.
2. Ausencia de Interbloqueo (Deadlock): Si algún hilo intenta adquirir el bloqueo, algún hilo logrará adquirirlo. En caso de que un hilo solicite el mutex pero nunca lo adquiere, implica que otros hilos deben estar completando un número infinito de secciones críticas.
3. Ausencia de Privación (Starvation): Cada hilo que intenta adquirir el bloqueo eventualmente tiene éxito. En otras palabras, ningún hilo debería quedarse indefinidamente sin poder adquirir el bloqueo, sin importar cuánto tiempo haya transcurrido.

- **Colas de trabajo**

  Una cola de trabajo es una estructura de datos utilizada en la programación concurrente para administrar y coordinar la ejecución de las tareas. Es un mecanismo comúnmente utilizado en entornos en los que se requiere procesar múltiples tareas de manera eficiente y distribuida entre varios subprocesos o hebras de ejecución.

  La idea principal detrás de esta estrategia es almacenar en una cola, todos los trabajos pendientes que deben ser realizados por los subprocesos. Cada trabajo en la cola representa una tarea independiente que puede ser ejecutada por una hebra y sigue una política “FIFO” de encolar y desencolar, es decir, el primer trabajo que entra, es el primero en salir. La dinámica de la cola de trabajos implica la interacción entre una hebra principal, conocida como “maestro”, y varios subprocesos llamados “trabajadores”. El maestro es responsable de encolar los trabajos en la cola mientras se van generan o cuando se requiere de su ejecución. Los trabajadores, por otro lado, desencolan los trabajos de la cola y los ejecutan de forma concurrente.

  El proceso comienza cuando el maestro encola los trabajos disponibles. Luego, los trabajadores, que están a la espera, desencolan los trabajos de la cola y los ejecutan de manera independiente. Una vez que un trabajo ha sido completado, el trabajador busca en nuevamente en la cola trabajos disponibles y repite el proceso. Esto se lleva a cabo hasta que no haya más trabajos en la cola y todos los subprocesos hayan terminado su ejecución.

  La ventaja de utilizar una cola de trabajo es que permite distribuir la carga de trabajo entre múltiples subprocesos definidos por el diseñador, de esta manera se puede definir la cantidad de subprocesos a ejecutar igual al número de núcleos en la CPU y obtener una ejecución paralela en lugar de concurrente. Además, proporciona un mecanismo de sincronización y coordinación entre los subprocesos, de tal manera que estos no necesitan esperar a que los demás terminen su ejecución para realizar otro trabajo. Esto lo realiza, evitando conflictos y garantizando que cada trabajo es procesado exactamente una vez.

- **Variables de condición**

  Cuando un hilo tiene acceso exclusivo a un dato compartido, puede encontrarse en una situación en la que no puede hacer más que esperar hasta que otro hilo cambie dicho dato.

Aunque el dato puede ser correcto y consistente, es posible que no sea de interés para el hilo en ese momento. Por ejemplo, si un hilo que accede a una cola y encuentra que está vacía, debe esperar hasta que se agregue una nueva entrada.

La cola y otros datos compartidos deben estar protegidos por un mutex. Por lo tanto, para determinar el estado actual de la cola, como si es que está vacía, un hilo debe bloquear el mutex y verificar su estado. Luego, debe desbloquear el mutex y esperar a que otro hilo cambie dato compartido si es que no es de su interes.

Es en este proceso donde surge un problema, si el hilo que tiene que esperar sigue en ejecución mientras otro hilo bloquea el mutex e inserta una entrada en la cola, este último no tiene como saber que hay un hilo en espera. Por lo tanto, el hilo en espera, que ya ha verificado que la cola está vacía, estará a la espera sin saber que la cola ya no está vacía. Para solucionar esta problemática, se crean las variables de condición.

Una variable de condición proporciona una forma de notificar y esperar eventos específicos en relación una condición particular (Butenhof, 1997). Se utiliza para comunicar información sobre el estado de los datos compartidos. Por ejemplo, permite señalar cuando una cola esta vacía o cuando esta deja de estarlo. En otras palabras, es una señal que permite indicar que se debe hacer algo más o se puede hacer algo dentro de los datos compartidos por las diferentes subprocesos.

Figura 17: Analogía de las condiciones de variable
Fuente: (Butenhof, 1997).

En la figura 17 se presenta una analogía sobre su comportamiento, donde 3 marineros (subprocesos) comparten el mismo bote (recurso compartido) mientras navegan en alta mar. Para coordinar sus tareas, tiene un sistema similar al de las variable de condición que les permite:

1. Señalar una condición a un miembro: Cuando un marinero contacta a otro compañero que se encuentra dormido para indicarle que es el momento de despertar y participar activamente remando, se lleva a cabo una comunicación de una condición hacia un marinero en específico.
2. Esperar una condición: Cuando uno de los marinos ha estado dedicado a remar sin pausa y, como resultado, se encuentra exhausto. Él confía en que otro compañero lo

relevará en el momento oportuno para permitirle descansar. En esta situación, el marinero aguarda pacientemente a que se cumpla una condición determinada (es decir, que otro compañero lo releve) antes de poder tomarse un merecido descanso.

3. Comunicar una condición a todo el grupo: Por otro lado, si se considera una situación en la que se descubre un agujero en el bote. Cuando un marinero del grupo se de cuenta de que el agua está ingresando más rápido, emite un llamado de auxilio mediante un grito, con el fin de alertar a todos los demás marineros sobre el problema. En este caso, se comunica una condición a todo el grupo de manera inmediata.

Si bien, las variables de condición permiten que los hilos intercambien información sobre los cambios en el estado de los datos compartidos, como las condiciones "lleno" o "vacío" de una cola. Estas no proporcionan exclusión mutua. Es por ello, que una variable de condición es un mecanismo que se utiliza siempre junto con un mutex y, por extensión, con los datos compartidos que están protegidos por él. Esperar en una variable de condición significa liberar atómicamente el mutex asociado y esperar hasta que un hilo mande una señal (para despertar a un hilo en espera) o emita una difusión (para despertar a todos los hilos en espera) con la variable de condición. Es importante destacar dos cosas, con respecto a su uso: (1) el mutex siempre debe estar bloqueado cuando se espera en una variable de condición y (2) cuando un hilo se despierta de una espera, siempre se reanuda con el mutex bloqueado.

2.4. Algoritmos Evolutivos en paralelo

En términos generales, la mayoría de las familias de algoritmos evolutivos adoptan una estructura centralizada o panmíctica[7] para su población. Esto significa que la población se trata como un único grupo de individuos. Lo que implica: (1) la selección de sus individuos (para su reproducción) ocurre a un nivel global, (2) cualquier individuo tiene la posibilidad de reproducirse con cualquier otro y (3) cualquier individuo puede ser reemplazado por otro nuevo en la siguiente generación (Alba y Tomassini, 2002).

Además, es posible clasificar a los algoritmos evolutivos centralizados según el grado de concentración de nuevos individuos en su paso reproductivo. La figura 18 detalla esta clasificación, donde en el extremo de grado más alto, se encuentra el modelo "generacional", en el cual la población temporal o nueva de individuos reemplaza por completo a la población anterior. Mientras que, en el extremo opuesto, se encuentra el modelo de "estado estacionario", que considera el reemplazo de solo uno o dos individuos de la población anterior. En la región intermedia de estos dos extremos, existe una gran variedad de modelos para decidir cuál será la nueva población denominados genéricamente como algoritmos de "brecha generacional". En estos modelos, un determinado porcentaje de los individuos de la generación antigua son reemplazados por los nuevos individuos de la descendencia. De esta manera, la selección generacional y de estado estacionario se consideran dos subclases especiales de los algoritmos de brecha generacional (Syswerda, 1991).

[7] Población en la que el apareamiento es libre y al azar.

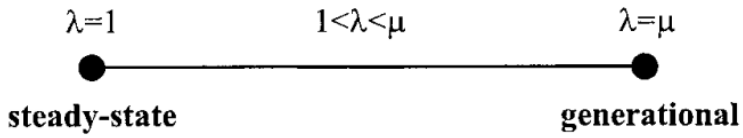


Figura 18: Grado λ de inserción de nuevos individuos sobre la población μ.
Fuente: (Alba y Tomassini, 2002).

Por otra parte, se presenta una alternativa para estructurar la población en los algoritmos evolutivos que difiere del enfoque centralizado. En este caso, se requiere una organización espacial de los individuos de la población. Para lograr esto, se descentraliza la población en múltiples subpoblaciones. Esta configuración implica lo siguiente: (1) la selección de individuos para la reproducción ocurre a nivel local dentro de cada subpoblación, en lugar de a nivel global, (2) cada individuo de una subpoblación solo tiene la posibilidad de reproducirse con otros individuos que pertenezcan a la misma subpoblación y (3) al determinar qué individuos se mantendrán en la nueva generación, solo se permite que los individuos de la generación anterior sean reemplazados por nuevos individuos si pertenecen a la misma subpoblación. En resumen, se limitan los ciclos reproductivos para que ocurran dentro de cada subpoblación.

Entre los algoritmos evolutivos descentralizados más conocidos se encuentran los algoritmos evolutivos distribuidos (dEA, por sus siglas en inglés) y los algoritmos evolutivos celulares (cEA, por sus siglas en inglés). En los dEA, también conocidos como algoritmos basados en islas o Island Model, se divide su población en múltiples poblaciones (o demes). De esta manera, se puede considerar su población como múltiples islas, donde cada una de ellas, realiza un ciclo reproductivo sin afectar a las otras islas. Luego, después de un número fijo de generaciones, conocido como época, se seleccionan individuos de cada isla para realizar un intercambio con individuos de islas vecinas. En los dEA, este proceso que se denomina migración y para su implementación, requiere previamente definir: una política de migración, la topología de las islas, el momento en que ocurre la migración (época) y qué individuos se intercambian. También, debe sincronizar cada isla durante su ejecución. Dado que el calculo en la tasa de comunicación suele ser alto, ocasionalmente se les llama algoritmos de granularidad gruesa.

Por otro lado, en los cEA, la población de individuos se divide en un mayor número de subpoblaciones más pequeñas y superpuestas. Por lo cual, a veces se les denomina algoritmos de granuladidad fina, al no requerir de intercambios. En este enfoque, la estructura de la población puede organizarse espacialmente de forma unidimensional (1-D) o bidimensional (2-D), donde los individuos se organizan en una línea o en una cuadrícula, respectivamente. Cada individuo tiene su propio grupo de vecinos con los que puede interactuar determinado por su cercanía espacial. De esta manera, un individuo puede pertenecer a varios de estos grupos al mismo tiempo, lo que permite interacción y intercambio de información entre diferentes grupos de individuos. Estos grupos individuales se denominan estanques, y el ciclo reproductivo se lleva a cabo dentro de cada uno de ellos.

En la figura 19 se muestra una representación de las diferentes formas de distribuir la población de acuerdo a los dos métodos anteriores y se contrasta con su versión centralizada.

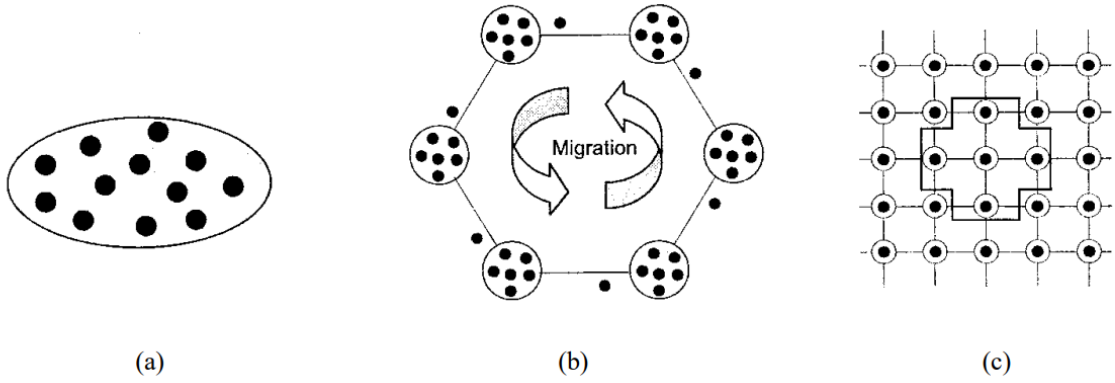


Figura 19: Ilustración de diferentes tipos de EA: (a) EA centralizado, (b) dEA y (c) cEA
Fuente: (Alba y Tomassini, 2002).

En la literatura se ha estudiado diversas estrategias para diseñar algoritmos evolutivos más eficientes. Estas estrategias incluyen la exploración de nuevos operadores de variación, el uso de enfoques híbridos que combinan técnicas evolutivas con otros métodos de optimización y el uso de modelos en paralelo para acelerar los cálculos. Sin embargo, el uso de la capacidad inherente que presentan los algoritmos evolutivos para su paralelización, es lo que ha hecho que este tipo de métodos sean tan populares (Alba y Tomassini, 2002).

En los estudios clásicos de algoritmos evolutivos paralelos (ej. PGA's, por sus siglas en inglés) se reconocieron tres formas de paralelización que siguen vigentes hoy en día: (1) maestro-esclavo de una población centralizada, (2) única población de granularidad fina (cEA) y (3) múltiples subpoblaciones de granularidad gruesa (dEA) (Cantú-Paz, 2000).

Le primera forma, considera el uso de un algoritmo evolutivo con una población centralizada. Dado que en este tipo de algoritmos, la selección y el cruce consideran a toda la población, también se les conoce como algoritmos paralelos globales. El enfoque de paralelismo global consiste en evaluar en paralelo a los individuos de la población por medio de subprocesos (esclavos), pero utilizar una selección sobre la población centralizada de manera secuencial en el procesador principal (maestro), que guía a el algoritmo base. Esto implica seleccionar a los candidatos más prometedores sobre una única población y obtener su descendencia de forma secuencial, mientras que el calculo de sus aptitudes se realiza de forma paralela en una arquitectura SIMD o MIMD, dependiendo de la complejidad dichos cálculos.

La segunda forma, implica el uso de los algoritmos evolutivos con una población celular (cEA). E este modelo requiere computadoras masivamente paralelas. La selección y reproducción están restringidos a un pequeño vecindario, pero los vecindarios se superponen, lo que permite cierta

interacción entre todos los individuos. Para su implementación, simplemente requiere evaluar diferentes individuos (con su vencidario) de forma paralela por medio de subprocesos y, en el caso ideal, tener un solo individuo para cada elemento de procesamiento disponible. En este esquema, la selección de padres, el cruzamiento y los operadores de variación, ocurren entre dos cromosomas dentro de los vecindarios. Sus características más importantes son la selección local y el cruce local.

En la última forma, se consideran a los algoritmos evolutivo con una población distribuida (dEA). Estos son más sofisticados, ya que consisten en varias subpoblaciones que intercambian individuos ocasionalmente, como se menciona anteriormente. Los algoritmos genéticos de múltiples demes son muy populares, pero también son la clase de algoritmos genéticos paralelos más difíciles de entender, porque los efectos de la migración no se comprenden completamente. Los algoritmos genéticos de múltiples demes introducen cambios fundamentales en el funcionamiento del algoritmo y no se puede saber si empeorarán o no las soluciones obtenidas. Para su implementación se puede utilizar un maestro que guié y realice la sincronización entre las islas durante sus migraciones. De esta manera, se puede asociar múltiples subprocesos (esclavos) para la ejecución de un ciclo reproductivo en cada una de las islas. En el mejor de los casos, se considera una isla por subproceso.

Dado que el tamaño de los demes es menor que la población utilizada en un incio, se espera que el algoritmo genético paralelo converga más rápido. Sin embargo, al comparar el rendimiento de ambas implementaciones, también debemos considerar la calidad de las soluciones encontradas en cada caso. Por lo tanto, si bien es cierto que los demes más pequeños convergen más rápido, también es cierto que la calidad de la solución podría ser peor (Cantú-Paz, 2000).

Es importante enfatizar que mientras el método de paralelización global no afecta el comportamiento del algoritmo, los últimos dos métodos si cambian la forma en que funciona. A continuación, se presentarán 3 estudios en donde se abordan las diferentes implementaciones paralelas de algoritmos genéticos.

2.4.1. High Performance Genetic Algorithm for Land Use Planning

En el estudio presentado en (Porta y cols., 2013), los investigadores paralelizaron el bucle genético, es decir, los procesos para generar nuevos individuos a través de múltiples subprocesos o hebras. Cada subproceso genera un conjunto de individuos realizando selección de padres, cruzamiento, mutaciones y la operación de reemplazo. De esta manera, como las hebras comparten memoria, todos pueden leer la misma población y actualizar a una siguiente generación. No obstante, en el proceso de paralelización, los investigadores no contemplaron el cálculo de aptitud sobre la población inicial.

Para llevar a cabo la paralelización, los investigadores implementaron tres soluciones diferentes: una solución multinúcleo, una solución de clúster y una versión híbrida que combina ambos enfoques. En la implementación multinúcleo, utilizaron el paquete Java java.util.concurrent para

lanzar hilos en diferentes núcleos, permitiendo que cada hilo sea independiente y realice su tarea de generación de individuos en paralelo. Este enfoque no requiere de una mayor sincronización, solo requiere configurar un mutex para la escritura de la población. En la solución de clúster, emplearon la biblioteca de paso de mensajes de Java, MPJ Express, para que cada proceso se ejecute en un nodo diferente del clúster y se comuniquen entre sí, intercambiando información sobre los mejores individuos en cada sincronización. Para ello implementaron una metodología maestro-esclavo, donde el maestro recibe el mejor individuo de cada esclavo y realiza la sincronización, mientras que los esclavos realizan el algoritmo completo en cada uno de ellos con poblaciones diferentes. La versión híbrida combina las implementaciones multinúcleo y clúster para aprovechar ambos enfoques paralelos y mejorar los resultados.

El problema abordado fue el de la planificación de uso de la tierra, un problema de optimización donde cada parcela de tierra se asigna a una categoría según ciertos criterios y restricciones. Los investigadores utilizaron algoritmos genéticos con una representación basada en una cadena de genes para cada bloque de tierra y una función de evaluación que consideraba la idoneidad de cada parcela para la categoría asignada y la compacidad de los parches de uso de la tierra en dicha parcela. El primero se le denomina “aptitud” y al segundo “compacidad”.

Los resultados mostraron que la implementación multinúcleo logró una aceleración superlineal cuando el número de hilos utilizados por el algoritmo genético paralelo fue menor o igual al número de núcleos físicos de la computadora. Esto significa que, al utilizar más hilos, se obtuvo un aumento significativo en la velocidad de ejecución y en la cantidad de iteraciones del bucle genético realizadas en el mismo tiempo que la versión secuencial. Sin embargo, se destacó que la eficiencia disminuyó al utilizar más núcleos debido a la limitación del Hyperthreading, que no es tan eficiente como tener un núcleo físico diferente para cada hilo. Por otro lado, la solución de clúster buscó mejorar la aptitud en el menor tiempo posible, por lo que no se proporcionó una discusión específica sobre la aceleración en esta versión. Por último, los resultados de la versión híbrida que combinó las implementaciones multinúcleo y de clúster proporcionó mejores resultados que la versión multinúcleo. La combinación de ambos enfoques paralelos demostró ser la más efectiva en términos de eficiencia y rendimiento del algoritmo genético.

2.4.2. Topology and Evolutionary Migration Policy of Fine-grained Parallel Algorithms for Numerical Optimization

En la investigación realizada en (Lee, Park, y Kim, 2000), los autores se enfocaron en paralelizar un algoritmo evolutivo con población celular o Fine-grained Parallel Evolutionary Algorithms (FGPEAs), con el objetivo de utilizar un gran número de procesadores y evitar incompatibilidades entre individuos cruzados.

La paralelización se centró en la población celular y se propuso una topología de árbol binario completo, en lugar de una cuadricula 2D. Además, se propuso dos nuevos métodos de migración junto a esta nueva topología, para evitar el monopolio de un superindividuo y preservar la diversidad en las subpoblaciones. Aunque realizaron la paralelización de la población celular, es

decir, realizar un ciclo reproductivo en vecindarios de forma paralela, no consideraron paralelizar el cálculo de aptitudes de la población inicial.

Para su implementación, se utilizó un sistema CrayT3E (Cray T3E-900 LC128A-128) como computadora paralela. Es un tipo de computadora MPP (procesamiento paralelo masivo). Se utilizó un microprocesador DEC Alpha de 450 MHz para cada elemento de procesador y se disponían de 128 procesadores. En este trabajo, se utilizaron 64 procesadores para la simulación y se utilizó MPI (interfaz de paso de mensajes) como herramienta de programación paralela para la comunicación entre procesadores.

El problema en cuestión fue la optimización numérica con restricciones, para ello, se utilizó Evolian (Kim y Myung, 1997) para integrar el concepto de escalamiento de restricciones y optimización multifase, y se seleccionaron cinco casos de prueba de Michalewicz (Rocke, 2000) que representaban diversos tipos de funciones objetivo y restricciones. Los resultados mostraron que los FGPEAs con topología de árbol y los métodos de migración propuestos mejoraron el rendimiento del algoritmo en comparación con otros enfoques, como EAs secuenciales y FGPEAs de cuadrícula 2D. Aunque los FGPEAs con topología de árbol necesitaron más tiempo para obtener la solución óptima en problemas difíciles (como la optimización numérica altamente restringida), demostraron un buen rendimiento y evitaron la convergencia prematura gracias a los métodos de migración implementados. En general, la paralelización con FGPEAs mostró un speedup significativo en la reducción del tiempo de cálculo al utilizar múltiples procesadores, en comparación a su versión secuencial.

2.4.3. A coarse-grained Parallelization of Genetic Algorithms

En el artículo presentado en (Rathomi y Pulungan, 2018), se propuso paralelizar un algoritmo genético con una población distribuida en dos niveles. El primer nivel hace uso de paralelismo en computadores distribuidos, mientras que el segundo nivel se aplica un paralelismo a nivel de hardware.

En el primer nivel, la población se divide en subpoblaciones según el número de nodos de cómputo disponibles. Este nivel utiliza el paradigma de paso de mensajes con paralelismo maestro-esclavo: el nodo 0 (maestro) envía a todos los nodos esclavos el tamaño de la subpoblación. Cada nodo esclavo genera su propia subpoblación y ejecuta el algoritmo evolutivo en ella, luego de terminar su ejecución procede a intercambiar sus mejores individuos en la población “general”. Para ello, el nodo maestro recopila todas las subpoblaciones finales de todos los nodos esclavos y produce el resultado final.

En el segundo nivel, se realiza en el procesamiento de cada individuo, en este se paralelizan las partes del algoritmo genético que no tienen dependencia individual al hacer uso de la GPU: la selección, el cruzamiento, la mutación, la evaluación individual y la actualización de los individuos de la población. Es importante destacar que, en esta implementación, se realiza el bucle en cada generación y en cada nodo esclavo de forma secuencial al no existir una manera de abordar su

paralelización.

Para su implementación, en el primer nivel, la población se dividió en subpoblaciones utilizando MPI (Interfaz de paso de mensajes) para lograr una distribución equitativa entre los nodos de cómputo. Cada nodo esclavo generó su propia subpoblación y solo se comunicaron para intercambiar los mejores individuos. En el segundo nivel, que se llevó a cabo en la GPU, se trabajó con CUDA threads, que abordaron de manera independiente utilizando el modelo Single Instruction Multiple Threads (SIMT), similar al modelo SIMD explicado en la sección 2.3.1, pero utilizando la GPU. Esto implica que no exista una comunicación entre ellos (sincronización). El trabajo de cada hilo fue de manera independiente y aplicó todas las operaciones del algoritmo genético mencionadas anteriormente.

El problema abordado fue el Job Shop Scheduling Problem (JSSP), y los experimentos se realizaron utilizando diferentes conjuntos de datos relacionados a este problema. Los experimentos mostraron que la precisión de los resultados obtenidos por el algoritmo genético paralelo y el algoritmo genético secuencial fue relativamente similar, con una diferencia máxima de desviación estándar de aproximadamente 9.31. Aunque el tiempo de cálculo para encontrar una solución -utilizando el algoritmo genético paralelo- aún no era óptimo en comparación con la versión secuencial, la implementación paralela propuesta logró alcanzar más rápidamente el resultado de convergencia que la versión secuencial.

CAPÍTULO 3
PROPUESTA DE SOLUCIÓN

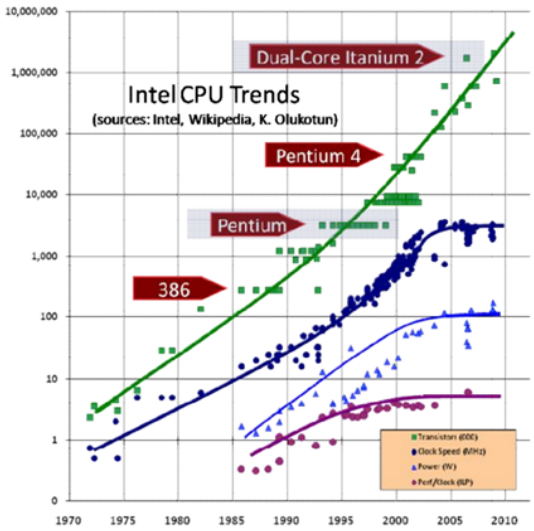


Figura 20: Tendencias de CPU Intel en los últimos años.
Fuente: (Saidu, Obiniyi, y Ogedebe, 2015).

En la figura 20 se puede apreciar que, cada año, se logra incorporar una mayor cantidad de transistores en la CPU. Sin embargo, el uso de un único procesador limita el incremento en su velocidad debido al riesgo de un sobrecalentamiento. Es por esto, y por lo discutido en capítulos anteriores, que los fabricantes adoptan un enfoque de arquitecturas “multinúcleo”, en las cuales múltiples procesadores (o núcleos) se comunican entre sí a través de memorias compartidas. Estos chips multiprocesadores permiten mejorar la eficiencia de la computación al emplear varios procesadores para trabajar en una misma tarea (Herlihy y Shavit, 2012).

Resolver problemas de optimización de la vida real, por lo general, considera tiempos extremadamente largos debido a su naturaleza NP-Difícil. Aún al hacer uso de metaheuristicas, pueden haber casos en que el tiempo para encontrar una solución sea muy extenso. Este viene a ser el caso de EVOCA, donde se implementa un algoritmo evolutivo para la sintonización de parámetros de algoritmos objetivos y debe considerar: (1) un espacio de búsqueda extremadamente grande al tener que considerar todos los posibles valores de los parámetros; y (2) múltiples ejecuciones del algoritmo objetivo debido al manejo de su estocasticidad. Computar la función de evaluación sobre un candidato, o en otras palabras, obtener la aptitud para cada individuo de la población, suele ser una de las operaciones de mayor costo en términos computacionales, al tener que realizar esta operación repetidamente. Debido a esto y a los objetivos planteados en un comienzo, es que se postula efectuar una paralelización del sintonizador EVOCA, en particular, haciendo uso de programación en multiprocesadores.

3.1. Análisis de EVOCA

El sintonizador de parámetros EVOCA se puede clasificar como un algoritmo evolutivo centralizado y de estado estacionario. Esta clasificación implica que: (1) el algoritmo evolutivo presenta una estructura centralizada en su población de soluciones y (2) en cada iteración se reemplazan como máximo dos individuos de la población.

Al considerar las propuestas de paralelización en la literatura, por un lado, se tiene el enfoque global de paralelización, que implica realizar el cálculo de aptitudes en forma paralela, mientras que se realiza la selección de padres y la generación de nuevos candidatos de manera secuencial. Por otro lado, existe el enfoque de descentralizar la población original en múltiples subpoblaciones de menor tamaño, para ejecutar de manera paralela el sintonizador EVOCA en cada una de ellas.

En el primer caso, no es posible realizar un calculo en paralelo sobre las aptitudes de la población temporal debido a que EVOCA realiza la operación de mutación sobre el individuo generado por el operador de cruzamiento. Esto implica que se deba esperar la finalización del operador de cruzamiento para realizar la ejecución del operador de mutación, lo que no genera un gran beneficio al momento de paralelizar al convertir el cálculo de las aptitudes en procedimientos secuenciales. En cuanto al segundo caso, la opción de descentralizar la población conlleva considerar todos los parámetros adicionales que se deben definir para su ejecución, por ejemplo: el número y el tamaño de las subpoblaciones, la frecuencia en ocurren las migraciones, el número de candidatos que van a migrar, el destino de los migrantes, entre otros. Esto implica que se vea afectada la eficiencia en la calidad de las soluciones que se pueden alcanzar, ya que si es que no se escogen valores adecuados para cada uno de estos parámetros, la obtención de buenas configuraciones puede tener un impacto negativo. Esto genera una disyuntiva sobre el problema, ya que se deben encontrar valores adecuados para un sintonizador, que busca valores adecuados en los parámetros del algoritmo objetivo.

Es por ello que se propone adoptar un enfoque diferente, pero un poco similar al enfoque de paralelización global. Este nueva propuesta busca paralelizar el cálculo individual de las aptitudes en la población, al considerar las R semillas o instancias aleatorias que deben ser evaluadas para cada una de ellas. De esta manera, en vez de obtener las R aptitudes de manera secuencial, se propone paralelizar este proceso en el sintonizador al calcular todas las aptitudes de manera concurrente por medio de subprocesos. Así, en el mejor de los casos, se pretende calcular todas las aptitudes simultáneamente, en el tiempo requerido para realizar la evaluación más prolongada de ellas. En la figura 21 se puede observar que dicha operación esta presente en casi todos los pasos del algoritmo, se puede ver en el procedimiento de la población inicial, en el cruzamiento, en la mutación y en sus iteraciones para ejecutar el algoritmo de hill climbing. Es decir, esta operación únicamente no se considera en la selección de padres y en la actualización de la población.

La principal ventaja de esta nueva implementación reside, por un lado, en la potencial reducción de los tiempos globales del algoritmo, siempre y cuando el tiempo requerido para la ejecución de las R semillas sea mayor al tiempo requerido para la sincronización de los subprocesos, en cuyo

caso, los tiempos de espera globales no son muy altos. Por otro lado, esta implementación no afecta el funcionamiento ni la calidad de las soluciones obtenidas en EVOCA, al no considerar nuevos parámetros o operadores en el proceso de sintonización. En otras palabras, ambas implementaciones deben conducir a los resultados equivalentes al considerar el mismo escenario.

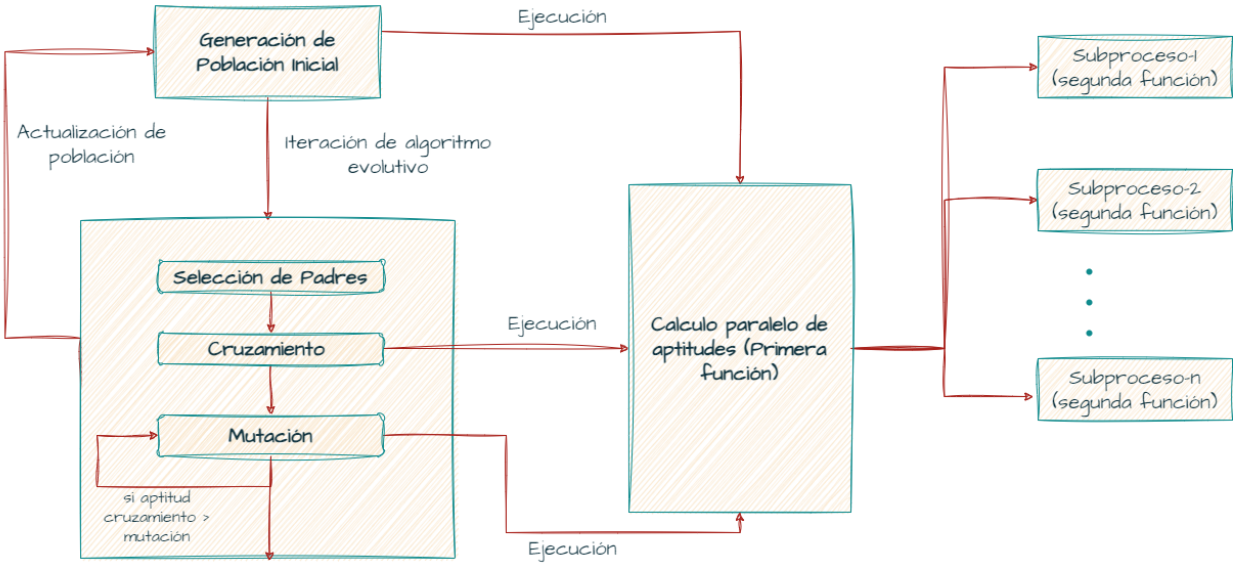


Figura 21: Propuesta global de PEVOCA.
Fuente: Elaboración propia.

3.2. Arquitectura de EVOCA en paralelo

Para lograr este nuevo enfoque, se propone utilizar una arquitectura MIMD en el proceso de obtención de aptitudes, ya que esta arquitectura permite la ejecución de diferentes tipos de datos en diferentes procesadores. De esta manera, si se observa la figura 22, cada instrucción corresponde a una instrucción asociada al cálculo de aptitud de una configuración y cada flujo de datos representa a la instancia/semilla en la que se debe efectuar dicho calculo.

Para emplear esta arquitectura, se decide utilizar un sistema con multinucleos en su procesador y hacer uso de la librería Pthreads, asociada a la programación en multiprocesadores. Con ello, se realiza la paralización del sintonizador EVOCA, a través de la ejecución de subprocesos POSIX.[8]

Pthreads es un modelo de ejecución independiente de un lenguaje de programación. Esta librería permite que un programa controle múltiples flujos de trabajo, cada flujo de trabajo se denomina subproceso o hebra, y tanto su creación como su control se logra mediante llamadas a la

[8]POSIX es el acrónimo de Portable Operating System Interface, y la 'X' proviene de UNIX como una señal de identidad de la API. Es una norma escrita por la IEEE que define una interfaz estándar del sistema operativo y el entorno, incluyendo un intérprete de comandos (o "shell")

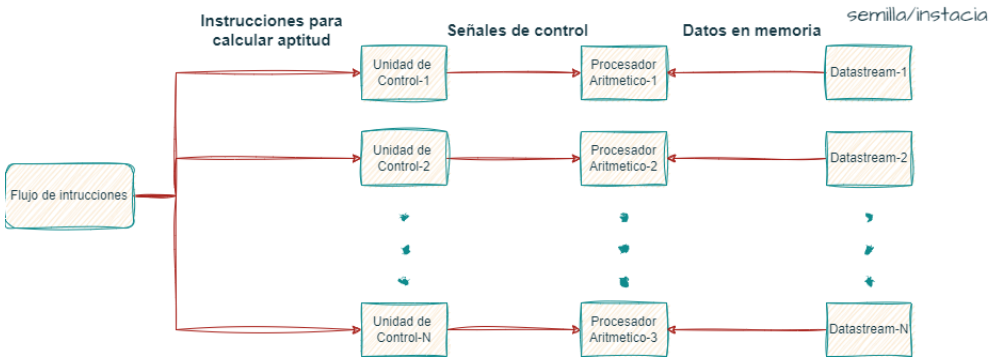


Figura 22: Diagrama de arquitectura MIMD a utilizar en EVOCA.
Fuente: Elaboración propia.

API de subprocesos POSIX. Cada subproceso POSIX o hebra hace uso de la programación en multiprocesadores y con ella se puede implementar la arquitectura MIMD expuesta anteriormente, al considerar que múltiples subprocesos POSIX pueden ejecutar múltiples flujos de instrucciones (algoritmo) con diferentes instancias de datos. El uso de Pthreads en este contexto permitirá aprovechar eficientemente los recursos de hardware disponibles al ejecutar los multiples subprocesos de manera concurrente, en diferentes núcleos de la CPU. Cabe destacar que, con cualquier librería asociada a la programación en multiprocesadores se podría implementar pero se prefiere esta, por un lado, debido a su extensa documentación y, por otro, dado a que el sistema en el que se realizará tanto el desarrollo, como las pruebas es un sistema Linux.

3.3. Parallel Evoltionary Calibrator

Para el desarrollo del algoritmo de EVOCA en paralelo, que llamaremos Parallel Evoltionary Calibrator (PEVOCA), se modificaron las funciones de EVOCA que estuvieran relacionadas al cálculo de la aptitud individual de las configuraciones de parámetros para emplear una metodología de cola de trabajos. Esta metodología consiste en dividir el par semilla e instancia que deben ser calculadas de manera paralela, en R diferentes tareas (o trabajos) y en la ejecución de $n-1$ subprocesos, o trabajadores, asociados a la cantidad de hebras que se desean ejecutar de manera concurrente, desde una hebra principal. De esta manera, los $n-1$ subprocesos se encargan de la ejecución concurrente de las R tareas, mientras que el hebra principal se encarga de encolar los R trabajos y de la sincronización de los subprocesos.

Entonces, se implementan dos nuevas funciones para la obtención de la aptitud de una configuración en paralelo. Estas funciones definen el comportamiento de la hebra principal y de sus subprocesos explicados anteriormente, al utilizar pthreads. Para su sincronización, se utiliza un mutex (pthread_mutex_t) y una variable de condición (pthread_cond_t) relacionados a la cola de trabajo, y se utiliza un mutex adicional para asegurar el acceso a secciones criticas por parte de los subprocesos.

En el algoritmo 1, se presenta la función asociada a la hebra principal. Esta comienza su proceso al inicializar las variables de sincronización, los mutexes y la variable de condición. A continuación, crea $n-1$ subprocesos concurrentes, los cuales ejecutan una segunda función que describe su comportamiento. Notar que estos subprocesos permanecerán en estado de espera mientras no haya tareas disponibles en la cola de trabajos.

Luego, se realiza una iteración para obtener las R semillas/instancias que deben ejecutarse en paralelo. En cada paso de la iteración, se obtienen los datos correspondientes de la semilla e instancia aleatoria y se encolan en forma de tarea en la cola de trabajos. Para lograr esto: (1) se bloquea el acceso a la cola de trabajo, (2) se encola la tarea y (3) una vez que la tarea ha sido encolada, se notifica a los subprocesos su disponibilidad a través de la variable de condición. Después de esto, se desbloquea el mutex para permitir el acceso compartido a la cola de trabajo por parte de los otros subprocesos y se continúa con la siguiente iteración.

Una vez que todas las tareas han sido encoladas, se marcan como procesadas y se espera a que todos los subprocesos finalicen su ejecución. Al terminar, se destruyen las variables de sincronización y se calcula la aptitud promedio de la configuración al dividir su valor actual por las R tareas procesadas.

Algorithm 1 Procedimiento en hebra principal

1: $n \leftarrow$ Número de hebras
2: $R \leftarrow$ Número de semillas/instancias
3: **procedure** calcular_aptitud_calibracion_paralelo(cal_{temp}, n, R)
4: Crear $n-1$ subprocesos que ejecutan hebra_calcular_aptitud_semilla_instancia(cal_{temp})
5: Inicializar mutexes y variable_condicional
6: **for** i = 0 hasta R-1 **do**
7: Obtener semilla e instancia aleatoria
8: Inicializar tarea
9: Bloquear mutex de la cola de trabajo
10: Agregar tarea a la cola de trabajo
11: Notificar a los subprocesos que hay una tarea disponible (variable de condición)
12: Desbloquear mutex de la cola de trabajo
13: **end for**
14: Marcar todas las tareas como procesadas
15: Esperar a que los subrpocesos terminen
16: Destruir mutexes y variable condicional
17: $cal_{temp} \rightarrow$ aptitud_promedio $= \frac{cal_{temp} \longrightarrow \text{aptitud\_promedio}}{k}$
18: **end procedure**

En el algoritmo 2, se presenta la función asociada a los subprocesos. En esta, se inicia en un bucle while, el cual se detiene únicamente bajo dos condiciones: (1) cuando no hay más tareas por procesar y todos los subprocesos han finalizado su ejecución; y (2) cuando se alcanza el límite máximo de evaluaciones (b_{max}) permitido.

El bucle while cumple con dos propósitos, en primer lugar, permite que los subprocesos no tengan que esperar a los demás para volver a ejecutarse. Una vez que han completado su tarea, tienen la capacidad de desencolar directamente otra tarea para su ejecución. Esto evita retrasos innecesarios y optimiza el uso de los recursos disponibles. En segundo lugar, cuando los subprocesos no tienen una tarea asignada, permite que permanezcan en un estado ocioso. En este estado, están a la espera de que la variable de condición cambie y existan tareas disponibles en la cola de trabajos. Bajo este mecanismo, se realizan todos las tareas exactamente una vez y lo más rápido que pueden ser procesadas.

Cabe destacar que, cuando hay tareas disponibles en la cola, lo adquiere el primer subproceso que logra acceder a la cola de trabajos. Para ello, se repite un procedimiento de acceso exclusivo, en donde: (1) se bloquea la cola de trabajo, (2) se desencola la tarea y (3) se desbloquea el acceso a la cola de trabajo. Luego de obtener la tarea, se procede a su ejecución, es decir, se realiza el calculo de la aptitud de la configuración bajo la semilla/instancia adquirida de la tarea. Al finalizar, se vuelve nuevamente al bucle while hasta que se cumpla una de las dos condiciones mencionadas anteriormente.

Algorithm 2 Procedimiento en subprocesos

```
1:  procedure hebra_calcular_aptitud_semilla_instancia(cal_temp)
2:      while true do
3:          Bloquear mutex de la cola de trabajo
4:          if Todas las tareas han sido procesadas y la cola de trabajo está vacía then
5:              Desbloquear mutex y salir del bucle
6:          end if
7:          if Hay tareas en la cola then
8:              Desencolar tarea de la cola
9:              Desbloquear mutex
10:             realizar_tarea(tarea, cal_temp)
11:             if Se alcanza el límite de evaluaciones then
12:                 Salir del programa
13:             end if
14:         else
15:             Esperar a que haya tareas en la cola por medio de la variable de condición
16:             Desbloquear mutex
17:         end if
18:     end while
19: end procedure
```

El calculo de la aptitud en cada subproceso se describe en el algoritmo 3. En ese proceso, primero se crea un archivo para guardar los resultados de una semilla en particular. Para evitar problemas de acceso simultáneo a un mismo archivo por parte de diferentes subprocesos, se determina previamente, que el nombre del archivo estará asociado al numero de la tarea que se realizará. Así, cada subproceso tiene asociado su propio archivo. Notar que en caso de existir una mayor

cantidad de hebras (R<$n-1$), al solo poder adquirir las tareas una vez, existirán hebras que simplemente estarán en estado ocioso.

A continuación, se realiza la ejecución del algoritmo objetivo con la configuración dada en la semilla/instancia obtenida de la tarea. Su aptitud se almacena en el archivo creado anteriormente y, posteriormente, se actualiza la aptitud promedio de la configuración sumando dicho valor al valor actual almacenado en la aptitud promedio. Para garantizar un acceso seguro a este recurso compartido por todas las hebras, se utiliza nuevamente un acceso exclusivo al realizar un bloqueo y desbloqueo del mutex en lo que denominamos como "sección crítica". En otras palabras, se permite que solo una hebra acceda a la escritura en la variable compartida (aptitud promedio) a la vez.

Algorithm 3 Procedimiento para realizar tareas

```
1: procedure realizar_tarea(tarea, cal_temp)
2:     Construir el nombre del archivo de resultados
3:     Crear el archivo de resultados
4:     Ejecutar algoritmo objetivo con parámetros de la tarea a realizar
5:     Guardar la aptitud obtenida de dicha ejecución en el archivo de resultados
6:     Bloquear mutex de sección critica
7:     Actualizar la aptitud promedio al sumar aptitud obtenida
8:     Desbloquear mutex de sección critica
9: end procedure
```

Es importante destacar una diferencia fundamental con respecto al proceso secuencial, en lugar de calcular el promedio de la aptitud en cada paso de la iteración de las R semillas, este cálculo se realiza únicamente al finalizar la ejecución de todos los subprocesos. Para ello, durante el proceso, cada hebra suma el valor de aptitud obtenida durante su ejecución a la aptitud promedio de la configuración y, al finalizar la ejecución de todos los subprocesos, se divide su total por la cantidad de R semillas procesadas.

3.4. Conclusiones

La implementación propuesta logra una paralelización transversal en el cálculo de aptitudes en el sintonizador EVOCA, lo cual se refleja en el procesamiento de la población inicial, el operador de cruzamiento y el operador de mutación. Sin embargo, la selección de padres y el mecanismo de reemplazo no se paralelizan, y cada generación ocurre de manera secuencial. Además, esta implementación presenta tres secciones críticas que requieren un manejo especializado: (1) el acceso a la cola de trabajo, (2) el acceso a la variable cal_{temp} y (3) el acceso a los archivos de resultados.

En primer lugar, el acceso a la cola de trabajo se controla mediante el uso de un mutex y una variable de condición. Esto permite que las hebras esclavas sean notificadas cuando hay una tarea

disponible y, al mismo tiempo, garantiza que solo una hebra tenga acceso exclusivo a la cola de trabajo en un momento dado.

En segundo lugar, el acceso a la variable compartida cal_{temp} se maneja mediante otro mutex asociado a las secciones críticas. Este enfoque garantiza que el acceso a la aptitud promedio almacenada en dicha variable se realice de manera exclusiva para evitar posibles conflictos de datos entre las hebras.

Por último, el manejo de los archivos de resultados se lleva a cabo mediante la creación de múltiples archivos con nombres diferentes, los cuales están asociados a la ejecución de una semilla en particular. Dado que la cola de trabajos asegura que cada hebra solo pueda ejecutar una tarea específica y que esta se realice exactamente una vez, se evita cualquier posible conflicto de acceso entre las hebras cuando intentan acceder a un mismo archivo simultáneamente durante su ejecución, ya que nunca tendrán asociado un archivo en común.

En conjunto, estas medidas de control garantizan que la implementación funcione de manera efectiva y segura, evitando problemas de concurrencia y asegurando la coherencia en el manejo de recursos compartidos por las hebras.

CAPÍTULO 4
VALIDACIÓN DE LA SOLUCIÓN

Con el propósito de validar la propuesta desarrollada en este trabajo, se ha optado por realizar una comparación entre la versión secuencial del sintonizador EVOCA y su implementación en paralelo PEVOCA. Para ello, se han llevado a cabo pruebas en dos escenarios distintos.

Estas pruebas fueron diseñadas para abarcar casos límites en donde se puede ejecutar EVOCA. Al considerar estas situaciones, se busca determinar los factores que podrían influir en el desempeño y funcionamiento de esta nueva versión del sintonizador. Entre dichos factores se incluyen el número de núcleos utilizados, la cantidad de semillas aleatorias que deben ejecutarse, la complejidad en el procedimiento de los algoritmos objetivos (tiempo de ejecución) y el número máximo de iteraciones o recursos que pueden ser empleados.

Los resultados obtenidos en estas pruebas son de suma importancia para establecer la eficacia y eficiencia de la implementación en paralelo en comparación con su versión secuencial. Se han recopilado datos cuantitativos que abarcan métricas de rendimiento, tales como los tiempos de respuesta y la calidad de la solución obtenida. La comparación entre ambas permitirá identificar en qué escenarios es posible lograr un mejor desempeño al aplicar la implementación en paralelo y en cuales no es posible. Además, de evaluar el impacto de la paralelización en términos de velocidad de procesamiento y capacidad de respuesta, en comparación con la versión secuencial.

Es importante destacar que en ambos escenarios se ha decidido solo utilizar el conjunto de instancias de entrenamiento (I_{train}). La razón de esta elección radica en que el objetivo de esta implementación no se centra en mejorar las configuraciones alcanzadas por Evoca. En consecuencia, carece de sentido evaluar dichas configuraciones en el conjunto de testeo, siendo la única preocupación el que se obtengan las mismas poblaciones finales en ambas implementaciones pero reduciendo su tiempo de ejecución con la versión paralela. Además, para evaluar su desempeño se considerara I_{train} como un conjunto de instancias de referencia o "benchmark", cuyo óptimo global se conoce previamente.

Por último, para la obtención de los tiempos de ejecución de ambas implementaciones de EVOCA en los diferentes escenarios, se considera el uso del comando /usr/bin/time, el cual está disponible en sistemas UNIX o Linux. Mediante este comando, se obtuvieron tres métricas temporales:

1. **Tiempo Real (*real*):** Representa el tiempo total transcurrido desde que el programa se inicia hasta que finaliza su ejecución. Esta métrica engloba el tiempo de ejecución del programa y cualquier tiempo de espera asociado, como bloqueos de E/S o la programación de otros procesos en el sistema. El tiempo real refleja el tiempo global empleado en completar la tarea, incluyendo periodos de inactividad.
2. **Tiempo de Usuario (*user*):** Indica el tiempo de CPU utilizado por el programa en el modo

de usuario. Es el tiempo que la CPU ha dedicado a ejecutar directamente el código del programa, excluyendo cualquier tiempo asignado por el sistema operativo u otros procesos del sistema. El tiempo de usuario cuantifica la cantidad de tiempo de CPU consumido al ejecutar las instrucciones propias del programa.

3. **Tiempo del Sistema (*sys*):** Corresponde al tiempo de CPU utilizado por el sistema operativo en nombre del programa. Esto incluye el tiempo en el que el sistema operativo ha llevado a cabo llamadas al sistema en representación del programa, como operaciones de entrada/salida (E/S) o la gestión de señales. El tiempo del sistema proporciona una medida del tiempo que el programa ha pasado en actividades relacionadas con el sistema operativo.

4.1. Algoritmo Objetivo: Ant Knapsack

En el primer escenario, se emplea un algoritmo objetivo (A) que busca resolver el problema de la mochila multidimensional mediante una implementación del algoritmo de optimización por colonia de hormigas, denominado Ant Knapsack (AK).

El problema de la mochila multidimensional (MKP, por sus siglas en inglés) se aborda en diferentes estudios (Khuri, Bäck, y Heitkotter, 1994; Leguizamon y Michalewicz, 1999; Osorio y Cuaya, 2005), representa una variante del clásico problema de la mochila mencionado en la sección 1.1. En este problema, en lugar de considerar una única dimensión (como peso o volumen) para la mochila y los objetos, se deben contemplar múltiples dimensiones que influyen en el problema, tales como peso, volumen, capacidad de carga, entre otros. En otras palabras, la capacidad de la mochila y el impacto de los objetos en ella ahora se manifiestan en múltiples dimensiones. De esta manera, se contempla la existencia de M objetos con distintas ganancias y capacidades relacionadas a D dimensiones diferentes y una mochila con capacidades limitadas en las D dimensiones.

Con el propósito de adaptar la optimización basada en colonias de hormigas a este problema, el enfoque implementado en AK contempla un grafo de M nodos interconectados por aristas que reflejan la ganancia asociada a cada objeto. El proceso restante guarda similitudes con la implementación de ACO mencionada en la sección 2.1.3, sin embargo, a diferencia del problema TSP, el objetivo en este caso es encontrar la ruta de costo máximo, con el propósito de maximizar la ganancia total dentro de la mochila. Además, al considerar los posibles destinos de las hormigas artificiales, se deben tener en consideración las restricciones presentes en las D dimensiones al agregar un objeto a la mochila. En otras palabras, al incluir un objeto, su capacidad se ve reducida en todas las dimensiones de la mochila, y si se excede su límite, dicho objeto no se considera como un destino viable para la hormiga.

Para su ejecución, se modela la ganancia de cada objeto como una matriz de $1xM$ donde cada columna indica la ganancia asociada a un objeto. También, se modelan las capacidades de cada objeto como otra matriz de MxD, donde su fila i representa al objeto y su columna j, su capacidad en la dimensión j de D. Por ultimo, se genera una matriz de feromonas de dimensiones MxM donde sus valores se asocian a la cantidad de feromona que existe al momento desplazarse de un

objeto i a otro objeto j. De esta manera, se configura en su diagonal valores igual a -1, al no ser un destino valido.

Además, se define la siguiente heurística local para el calculo de la atractividad de cada componente:

$$\eta_{ij} = \frac{\text{Profits}[j]}{\sum_{d \in D} \frac{\text{Weights}[j][d]}{\text{CapRem}[d]}}$$

donde Profits$[j]$ representa la ganancia del objeto j, Weights$[j][d]$ representa el peso del objeto j en la dimensión d, CapRem$[d]$ representa la capacidad restante en la dimensión d en la mochila.

De esta manera, la probabilidad de que una hormiga k, considere ir de un nodo i a un nodo j, en una iteración $t \geq 0$ esta dada por:

$$p_{ij}^{k}(t) = \frac{\tau_{ij}(t)^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{j \in \text{vecinos}_k} \tau_{ij}(t)^{\alpha} \cdot \eta_{ij}^{\beta}}$$

donde $\tau_{ij}(t)$ se puede observar en la sección 2.1.3.

Por último, se actualiza la matriz de feromonas al finalizar la iteración de todas las hormigas en un ciclo t y solo la hormiga con la mejor solución obtenida en dicho ciclo deposita una cantidad Q de feromona en su recorrido. Esta cantdidad viene dada por formula:

$$Q = \frac{1}{1 + \text{bestGlobal} - \text{bestFound}}$$

donde bestGlobal es la calidad mejor solución encontrada hasta el momento en toda la ejcución y bestFound es la calidad de la mejor solución encontrada en el ciclo.

4.1.1. Escenario de Sintonización

Para este escenario, se define la métrica de costo C como el error porcentual entre la solución generada por el algoritmo objetivo y el óptimo global en las instancias de entrenamiento (Opt):

$$C = \frac{\text{Mejor Aptitud observada} - Opt}{Opt} \cdot 100$$

La métrica C proporciona una evaluación precisa de la calidad de cada configuración, ya que ofrece información sobre la proximidad de la solución obtenida respecto al óptimo global del problema. En otras palabras, cuanto menor sea el valor de C, mayor será la cercanía de la solución

obtenida con el óptimo global, lo que indica una mayor calidad de la configuración utilizada en el algoritmo objetivo.

Para la ejecución del algoritmo objetivo en un par instancia/semilla dadas, se requiere definir previamente los siguientes parámetros P:

- **TotalEvaluations**: Número máximo de iteraciones.
- **TotalAnts**: Número de hormigas en la colonia.
- **ph_max**: Nivel máximo de feromona que puede haber en un nodo.
- **ph_min**: Nivel mínimo de feromona que puede haber en un nodo.
- **alpha**: Importancia relativa de la huella de feromona en la probabilidad escoger un nodo de destino (atractividad).
- **beta**: Importancia relativa al conocimiento heurístico en la probabilidad de escoger un nodo de destino (atractividad).
- **rho**: Tasa de evaporación de la feromona en los nodos del grafo.

De estos parámetros, se consideran valores fijos para **TotalEvaluations** (1000) y **ph_min** (0.01). Por lo tanto, no se incluyen en el proceso de sintonización. De esta forma, el conjunto final de parámetros P que se requiere configurar queda expresado como:

$$P = \{\text{TotalAnts}, \text{ph\_max}, \text{alpha}, \text{beta}, \text{rho}\}$$

Además, se definen los siguientes dominios iniciales (ID) para cada uno de estos parámetros:

- $ID_{\text{TotalAnts}} = \{2, 3, 4, ..., 50\}$ Con rango $\{d_l = 2, d_u = 50\}$ y precisión igual a 1
- $ID_{\text{ph\_max}} = \{0.1, 0.2, 0.3, ..., 20\}$ Con rango $\{d_l = 0.1, d_u = 20\}$ y precisión igual a 0.1
- $ID_{\text{alpha}} = \{0.1, 0.2, 0.3, ..., 20\}$ Con rango $\{d_l = 0.1, d_u = 20\}$ y precisión igual a 0.1
- $ID_{\text{beta}} = \{0.1, 0.2, 0.3, ..., 30\}$ Con rango $\{d_l = 0.1, d_u = 30\}$ y precisión igual a 0.1
- $ID_{\text{rho}} = \{0.001, 0.002, 0.003, ..., 1\}$ Con rango $\{d_l = 0.001, d_u = 1\}$ y precisión igual a 0.001

Luego, se consideran los siguientes hiperparámetros para llevar a cabo la ejecución del sintonizador:

Instancias

- **MaxEval = 10000**: Representa el número máximo de evaluaciones disponibles (b_{max}) por el sintonizador para encontrar una configuración adecuada.
- **MaxM = 10**: Es el número máximo de candidatos en la población.
- **semilla = 123**: Es una semilla fija que se utiliza para la generación de pseudo números aleatorios en EVOCA, permitiendo la replicación de los resultados.
- **R = 3**: Indica la cantidad de veces que se debe aplicar el algoritmo objetivo bajo una misma configuración en diferentes instancias.
- **n = 4**: Representa la cantidad de hebras que se ejecutarán concurrentemente para la obtención de la aptitud de una configuración. Cabe mencionar que este valor no influye en la implementación secuencial del algoritmo, siendo útil principalmente en el contexto de la implementación en paralelo. Se busca considerar el número máximo de núcleos en el sistema, para evitar posibles demoras de sincronización.

Finalmente, se utiliza un total de 27 instancias benchmark en I_{train}, para llevar a cabo las pruebas de aptitud de las configuraciones del problema de la mochila de hormigas (AK). Estas instancias se representan en forma de archivos.

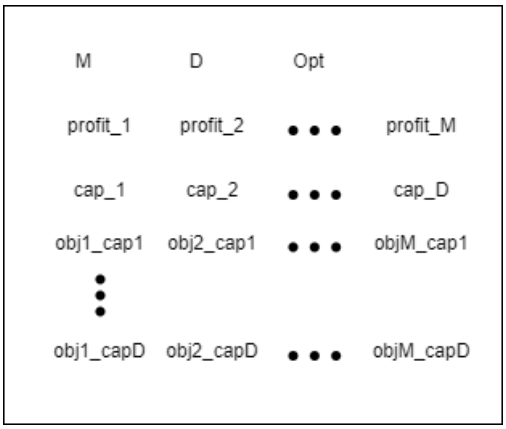


Figura 23: Diagrama de instancias benchmark utilizadas en AK.
Fuente: Elaboración propia.

En la figura 23 se muestra la estructura de cada uno de estos archivos, en donde:

1. La primera línea de cada archivo se encuentran tres valores: la cantidad de objetos disponibles (M), la cantidad de dimensiones a considerar (D) y el valor óptimo global para dicha instancia (Opt).
2. A continuación, se presentan M valores que representan la ganancia asociada a cada objeto.

3. Luego, se presentan D valores que indican la capacidad de la mochila en cada una de las dimensiones.

4. Por último, se presentan M líneas, cada una con D valores, para indicar las capacidades de los objetos en cada dimensión.

Este conjunto de instancias benchmark se ha obtenido a partir de un estudio previo referenciado como (Drake, 2015). Dichas instancias presentan una diversidad de objetos (M) y dimensiones (D). Concretamente, las instancias pertenecientes a la OR-Library siguen una nomenclatura con el siguiente formato: OR5x500_0.25.1. Donde 5 indica las D dimensiones, 500 los M objetos, 0.25 indica el tightness ratio y por último 1 indica su identificador. El parámetro "tightness ratio" representa el cociente entre la suma de los coeficientes de peso de cada objeto en una dimensión y la capacidad de dicha dimensión. Por otro lado, el "identificador", nos indica que pueden haber archivos con la misma nomenclatura pero con valores diferentes para sus objetos, es decir, sons instancias diferente pero bajo las mismas características principales.

En la tabla 2, se encuentran listadas las instancias específicas utilizadas en este escenario, siguiendo las convenciones mencionadas anteriormente.

Nombre del archivo	D	M	Tightness Ratio	Version
OR5x500-0.25_4.dat	5	500	0.25	4
OR5x500-0.25_7.dat	5	500	0.25	7
OR5x500-0.25_8.dat	5	500	0.25	8
OR5x500-0.25_9.dat	5	500	0.25	9
OR5x500-0.50_2.dat	5	500	0.50	2
OR5x500-0.50_6.dat	5	500	0.50	6
OR5x500-0.75_5.dat	5	500	0.75	5
OR10x500-0.25_1.dat	10	500	0.25	1
OR10x500-0.25_6.dat	10	500	0.25	6
OR10x500-0.25_7.dat	10	500	0.25	7
OR10x500-0.25_8.dat	10	500	0.25	8
OR10x500-0.25_9.dat	10	500	0.25	9
OR10x500-0.50_5.dat	10	500	0.50	5
OR10x500-0.50_6.dat	10	500	0.50	6
OR10x500-0.75_1.dat	10	500	0.75	1
OR10x500-0.75_2.dat	10	500	0.75	2
OR10x500-0.75_3.dat	10	500	0.75	3
OR10x500-0.75_4.dat	10	500	0.75	4
OR10x500-0.75_5.dat	10	500	0.75	5
OR10x500-0.75_10.dat	10	500	0.75	10
OR30x500-0.25_2.dat	30	500	0.25	2
OR30x500-0.25_6.dat	30	500	0.25	6
OR30x500-0.25_7.dat	30	500	0.25	7
OR30x500-0.50_2.dat	30	500	0.50	2
OR30x500-0.50_4.dat	30	500	0.50	4
OR30x500-0.75_3.dat	30	500	0.75	3
OR30x500-0.75_4.dat	30	500	0.75	4
OR30x500-0.75_8.dat	30	500	0.75	8
OR30x500-0.75_9.dat	30	500	0.75	9

Tabla 2: Nombre de archivos de instancias y sus características
Fuente: Elaboración propia.

4.1.2. Resultados

En la Tabla 3, se presentan los resultados obtenidos de la población inicial y final al utilizar tanto EVOCA secuencial como su implementación en paralelo en el primer escenario. Esta tabla muestra las mejoras en la calidad de las soluciones candidatas al completar el número máximo de evaluaciones (b_{max}) predefinido.

$P_{\text{TotalAnts}}$	P_{alpha}	P_{beta}	$P_{\text{ph\_max}}$	P_{rho}	C
22	8.1000	2.1000	6.1000	0.7010	2.16911
6	0.1000	6.1000	15.1000	0.5010	3.00524
50	10.1000	4.1000	30.0000	1.0000	6.51051
10	4.1000	10.1000	0.1000	0.8010	6.77907
18	2.1000	0.1000	3.1000	0.3010	6.89863
26	14.1000	14.1000	24.1000	0.4010	8.76812
14	16.1000	20.0000	12.1000	0.2010	13.1614
2	20.0000	16.1000	9.1000	0.1010	14.0909
34	12.1000	8.1000	21.1000	0.0010	16.3622
30	6.1000	12.1000	18.1000	0.6010	17.009

$\longrightarrow$

$P_{\text{TotalAnts}}$	P_{alpha}	P_{beta}	$P_{\text{ph\_max}}$	P_{rho}	C
6	1.9820	4.7400	15.3950	0.1030	0.16551
22	3.9140	4.7400	15.3950	0.1090	0.18104
28	3.5510	4.7400	10.3840	0.1030	0.183267
22	3.9140	4.7400	28.5530	0.1030	0.185055
22	3.9140	4.7400	28.5530	0.1090	0.189535
22	3.9140	4.7400	15.3950	0.1090	0.201439
22	3.9140	4.7400	15.3950	0.1030	0.204356
22	1.9820	4.7400	6.1000	0.1090	0.213224
8	3.9140	4.7400	28.5530	0.1090	0.393273
22	3.9140	4.7400	28.5530	0.1090	0.774867

Tabla 3: Valores de la población inicial y la población final obtenidos en ambas implementaciones en el primer escenario.

Por otro lado, la Tabla 4 muestra una comparación entre los tiempos de ejecución obtenidos en EVOCA secuencial y EVOCA paralelo. Los tiempos están representados en el formato “horas:minutos:segundos” y se dividen bajo las 3 métricas descritas anteriormente.

Programa	Tiempo Real	Tiempo de Usuario	Tiempo del Sistema
EVOCA secuencial	187:38:52	187:36:36	00:00:07
EVOCA paralelo	74:46:50	174:26:27	00:00:27

Tabla 4: Comparación de tiempos de ejecución entre implementación paralela y secuencial en el primer escenario.

En el caso de EVOCA secuencial, se requirieron aproximadamente 7 días, 19 horas y 38 minutos en tiempo real para completar su ejecución. Un tiempo similar con una diferencia de dos minutos fue utilizado en el modo de usuario, es decir, la CPU se dedicó ejecutar el código del programa casi en su totalidad. Además, el sistema operativo invirtió aproximadamente 7 segundos en actividades relacionadas con el programa. Por otro lado, PEVOCA logró reducir significativamente el tiempo de ejecución. En este caso, se completó su ejecución en aproximadamente 3 días, 2 horas y 45 minutos en tiempo real. En cuanto a el tiempo de usuario realziado fue de 7 días, 6 horas y 26 minutos, dado que el programa hizo un uso extensivo de la CPU en modo de usuario al considerar los tiempos de ejecución de todas las hebras concurrentes en su cálculo. Por ultimo, el tiempo

del sistema permaneció en aproximadamente 27 segundos. Este aumento se explica debido a la sincronización requerida en las hebras.

Por otro lado, al aplicar la ley de Amdahl en los resultados se tiene que al utilizar la ecuación (1) la aceleración obtenida por esta implementación esta dada por:

$$Speedup_4 = \frac{T_1}{T_{N4}} = \frac{675532}{269210} \approx 2.51 \tag{5}$$

Luego al considerar el uso de $n = 4$ núcleos físicos para la ejecución paralela, al utilizar la ecuación (2) se tiene que la eficiencia obtenida fue de:

$$Efficiency_4 = \frac{2.51}{4} = 0.6275 \approx 63\,\% \tag{6}$$

Con estos datos, al utilizar la ecuación (3) se tiene la fracción paralela que se ejecuto en este escenario fue de:

$$F_{parallel} = \frac{n}{n-1} \cdot \frac{T_1 - T_n}{T_1} = \frac{4}{3} \cdot \frac{675532 - 269210}{675532} \approx 0.82 \tag{7}$$

Por último, al utilizar la ecuación (4) se puede determinar la aceleración máxima que puede alcanzar esta ejecución:

$$maxSpeedup_4 = \frac{1}{1 - F_{parallel}} \approx 5.05 \tag{8}$$

Los resultados obtenidos demuestran, por un lado, que en este escenario se lograron satisfacer exitosamente todos los objetivos establecidos para la implementación paralela. Se pudo reducir significativamente los tiempos de ejecución sin comprometer la calidad de las soluciones obtenidas. Se validó este hecho al alcanzar la misma población final de soluciones y obtener los mismos costos, pero en un tiempo de ejecución considerablemente menor, con una diferencia aproximada de 4 días, 16 horas, 52 minutos y 53 segundos.

Por otro lado, se logro obtener una eficiencia igual al 62.75 % con la nueva versión, lo que implica que la implementación reduce un poco más de la mitad el tiempo requerido en la versión secuencial, sin embargo, puede seguir aumentando.

4.2. Algoritmo Objetivo: GA-NK

En el segundo escenario, se presenta un algoritmo objetivo (A) con un tiempo de ejecución reducido. Dicho algoritmo está orientado a resolver el problema de los paisajes de aptitud NK mediante la implementación de un algoritmo genético (GA-NK, por sus siglas en inglés).

En el ámbito de la biología evolutiva, los paisajes de aptitud representan una valiosa herramienta de análisis que permite visualizar y comprender la relación existente entre genotipos y su éxito reproductivo, es decir, su capacidad para propagarse eficazmente en una población. Esta medida de éxito se expresa simbólicamente como la "altura" dentro del paisaje, mientras que la adyacencia entre genotipos está determinada por su similitud. Es decir, aquellos genotipos que comparten similitudes se consideran adyacentes, mientras que aquellos más divergentes se encuentran distanciados. En consecuencia, el conjunto que abarca todos los genotipos posibles, con su grado de similitud y los valores correspondientes de aptitud, conforman un paisaje de aptitud.

Los paisajes de aptitud NK fueron presentados en (Kauffman y Weinberger, 1989) y son modelos que permiten ajustar la rugosidad de los paisajes de aptitud al modificar el tamaño general del paisaje y la cantidad de características locales, denominadas "colinas y valles", que lo componen. Estos paisajes utilizan cadenas de bits de longitud fija y se caracterizan por dos parámetros fundamentales: N, que indica el número total de bits en la cadena, y K, que determina el tamaño del vecindario para cada bit, es decir, cuántos valores puede cambiar cada bit. Para cada bit, se define una función que establece la aptitud propia y de sus vecinos. Por lo general, esta función se representa mediante una tabla de búsqueda de tamaño $2^k + 1$, con un valor para cada combinación posible del bit y sus vecinos. En la figura 24 se presenta una representación gráfica de este problema. El objetivo consiste en encontrar una combinación de valores que maximice la aptitud total de la cadena de bits, es decir, alcanzar la mayor altura en el paisaje de aptitud.

Este problema se clasifica como NP-completo cuando k es mayor que 1. Para afrontarlo, se emplea un enfoque basado en algoritmos genéticos, que sigue la lógica de los algoritmos evolutivos previamente explicados. En consecuencia, se establece lo siguiente:

1. Representación de las soluciones por medio de cadenas binarias de longitud fija: Cada solución se codifica como una cadena binaria con una longitud predefinida. Cada bit en la cadena representa un valor especifico de la cadena de bits de NK, y su valor puede ser 0 o 1, indicando la presencia o ausencia de dicho valor.

2. Función de evaluación para calcular las aptitudes de las cadenas binarias: Se define una función de aptitud que evalúa la calidad de cada solución representada por la cadena binaria. Esta función mide qué tan adecuada es la solución al hacer uso de las tablas de búsqueda.

3. Población inicial generada de manera aleatoria: Se crea una población inicial de tamaño ps de soluciones candidatas al azar. Cada solución se representa mediante una cadena binaria generada aleatoriamente con las características apropiadas para el problema.

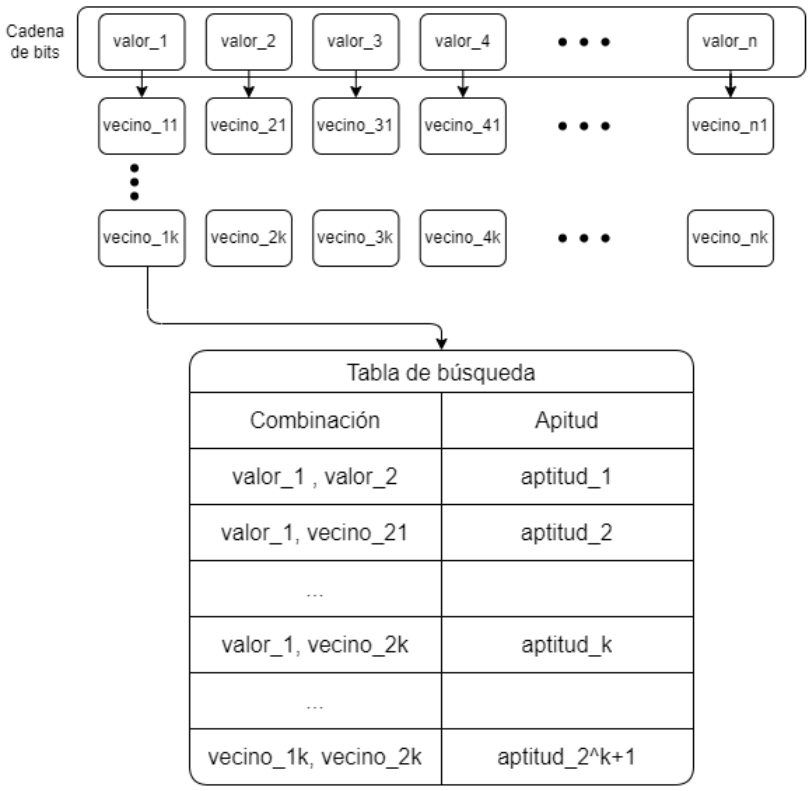


Figura 24: Diagrama de paisajes de aptitud NK.
Fuente: Elaboración propia.

4. Selección de padres por medio del método “selección por torneo binario’: se seleccionan ps soluciones candidatas de la población actual mediante un proceso de competencia llamado torneo binario. Se seleccionan aleatoriamente dos soluciones diferentes y se compara su aptitud. La solución con mayor aptitud se elige como uno de los padres para la reproducción, y el proceso se repite hasta completar las ps soluciones seleccionadas.

5. Para la generación de nuevos individuos se utilizaron 3 operadores: a) Cruzamiento uniforme: Se intercambian bits y segmentos entre dos soluciones padres seleccionadas para crear una nueva solución descendiente. b) Mutación de cambio de bit: Se aplica a una solución descendiente con una cierta probabilidad. En este proceso, uno o varios bits en la solución se cambian de valor (0 a 1 o viceversa). c) Un algoritmo de búsqueda local: Se aplica un procedimiento de búsqueda local determinista basado en inversiones de un solo bit (single-bit flips) sobre una solución candidata. En este operador, se realiza un número N de iteraciones sobre la solución candidata. En cada iteración, se evalúa si realizar un cambio en un solo bit (inversión de un solo bit) mejora la solución actual. Si se encuentra una solución vecina con una aptitud mejor, se acepta el cambio de bit y se continúa con el proceso.

6. Mecanismo de selección de sobrevivientes: Para el reemplazo o mecanismo de selección

de sobrevivientes, las nuevas soluciones candidatas se incorporan a la población original mediante un proceso denominado “reemplazo de torneo restringido” (RTR). En este proceso se eligen aleatoriamente dos soluciones de la población actual y una nueva solución descendiente. Luego, se selecciona la solución con mayor aptitud entre las tres como el sobreviviente.

7. El criterio de termino viene dado por un determinado número de evaluaciones máximo que puede realizar el algoritmo objetivo.

Es importante señalar que el operador de búsqueda local se aplica en cada iteración sobre el primer candidato de la población actual. Esto ocurre después de generar la población inicial y antes de aplicar el operador de selección. Por otro lado, los operadores de mutación y cruzamiento se aplican dada ciertas probabilidades en la población de soluciones seleccionadas durante el proceso de selección de padres.

4.2.1. Escenario de Sintonización

En este caso, la métrica de costo C se define como una medida que indica el número de evaluaciones realizadas por el algoritmo objetivo para encontrar el óptimo global en las instancias de entrenamiento. Esta métrica C también proporciona una evaluación precisa de la calidad de cada configuración, al ofrecer información sobre el tiempo que el algoritmo objetivo requiere para alcanzar el óptimo bajo diferentes configuraciones de parámetros. En otras palabras, cuanto menor sea el valor de C, menor será el tiempo necesario para encontrar el óptimo utilizando una configuración particular de parámetros. Es importante destacar que, en el peor de los casos, cuando no se logra encontrar el óptimo global, el valor de C será igual al número máximo de evaluaciones permitidas. Por lo tanto, se descartan aquellas configuraciones que obtienen el peor rendimiento y/o no llegan a encontrar el óptimo, al ser menos eficientes en la búsqueda de configuraciones.

Para la ejecución de este algoritmo objetivo en un par instancia/semilla dadas, se requiere definir previamente los siguientes parámetros P:

- **me**: Número máximo de evaluaciones para alcanzar el óptimo.
- **cr**: Tasa en la que se aplica el operador de cruzamiento.
- **mr**: Tasa en la que se aplica el operador de mutación.
- **ps**: Tamaño de la población.

De los cuales se considera un valor fijo para **me** (100000), por lo que no se incluye en el proceso de sintonización. De esta manera, el conjunto final de parámetros P que se requiere configurar en este escenario queda expresado como:

$$P = \{\text{cr}, \text{mr}, \text{ps}\}$$

Por otro lado, se definen los siguientes dominios iniciales (ID) para cada uno de los parámetros P a configurar por EVOCA:

- $ID_{\text{cr}} = \{0.00, 0.01, 0.02, ..., 1\}$ Con rango $\{d_l = 0.00, d_u = 1\}$ y precisión igual a 0.01
- $ID_{\text{mr}} = \{0.00, 0.01, 0.02, ..., 1\}$ Con rango $\{d_l = 0.00, d_u = 1\}$ y precisión igual a 0.01
- $ID_{\text{ps}} = \{2, 3, 4, ..., 50\}$ Con rango $\{d_l = 2, d_u = 50\}$ y precisión igual a 1

Luego, para llevar a cabo la ejecución del sintonizador se consideran los mismos hiperparámetros, pero considerando diferentes valores en el número de núcleos y en la cantidad de semillas aleatorias que se deben ejecutar:

- **MaxEval = 10000**
- **MaxM = 10**
- **semilla = 123**
- **R = 10**
- **n = {2, 4, 6, 8}**

Se considera un total de 50 instancias de benchmark pertenecientes al conjunto I_{train}. Estas instancias están representadas mediante archivos que siguen una estructura definida.

En la Figura 25 se representa la estructura que tiene cada archivo, de esta manera se tiene que:

1. La primera línea del archivo contiene dos valores: la cantidad de bits en la cadena (N) y la cantidad de vecinos para cada bit (K).
2. A continuación, se presentan los valores de cada bit y de sus vecinos. Por ejemplo, el primer número “Valor_1” representa el primer bit, y sus vecinos se describen como “vecino_1_1” hasta “vecino_1_k”. Este patrón se repite hasta completar los N bits.
3. Luego, se presentan $N \cdot 2^k + 1$ valores que indican las aptitudes de cada combinación en la tabla de búsqueda.
4. La última línea contiene el valor del óptimo global para la instancia (Opt).

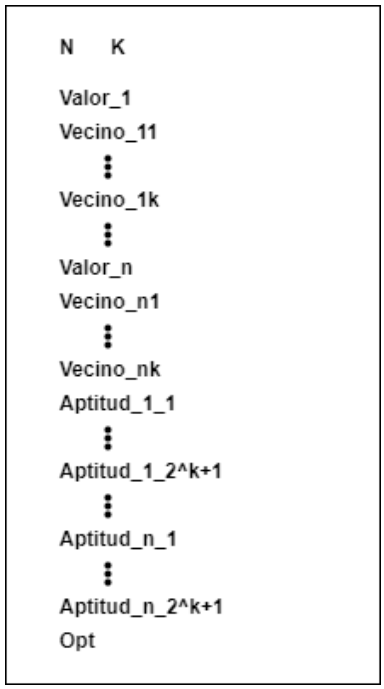


Figura 25: Diagrama de instancias benchmark utilizadas en AK.
Fuente: Elaboración propia.

Este conjunto de instancias benchmark se ha generado de manera aleatoria a partir del estudio presentado en (Pelikan, Sastry, Goldberg, Butz, y Hauschild, 2009). Dichas instancias presentan un $N = 20$ y un K variable entre 2 y 6. Estas instancias siguen una nomenclatura con el formato nk_N_K.version. Al igual que en el escenario anterior,"versión", nos indica que pueden haber archivos con la misma nomenclatura pero su valores son diferentes, es decir, son instancias con las mismas características principales pero diferentes valores.

En la Tabla 5, se encuentra un extracto del listadas las instancias específicas utilizadas para este escenario, se adjunta en anexo la tabla completa:

Nombre del archivo	N	K	Versión
nk_20_2.1	20	2	1
nk_20_2.2	20	2	2
nk_20_2.3	20	2	3
nk_20_2.4	20	2	4
nk_20_2.5	20	2	5
nk_20_2.6	20	2	6
nk_20_2.7	20	2	7
nk_20_2.8	20	2	8
nk_20_2.9	20	2	9
nk_20_2.10	20	2	10

Tabla 5: Nombre de archivos de instancias y sus características (extracto).
Fuente: Elaboración propia.

4.2.2. Resultados

En cuanto al segundo escenario, la Tabla 6 presenta la población inicial y final obtenida para las implementaciones de EVOCA secuencial y paralela, al considerar todos los casos de n descritos anteriormente. En esta tabla, se evidencian las mejoras en la calidad de las soluciones candidatas al alcanzar el número máximo de evaluaciones (b_{max}) predefinido.

P_{cr}	P_{mr}	P_{ps}	C
0.5000	0.7000	10	356
1.0000	0.3000	6	396
0.6000	0.4000	22	748
0.1000	0.6000	18	1067.4
0.7000	0.5000	34	1254.6
0.2000	0.1000	14	1864.8
0.4000	0.8000	50	1925
0.3000	0.2000	26	2158
0.8000	1.0000	2	50005.6
0.0000	0.0000	30	60030

$\rightarrow$

P_{cr}	P_{mr}	P_{ps}	C
0.6900	0.4000	2	17.6
0.7100	0.4000	2	20
0.7100	0.4000	3	20.4
0.3400	0.4000	2	20.8
0.7100	0.3500	2	21.4
0.8200	0.3300	2	23.4
0.8200	0.5200	2	23.8
0.6900	0.5200	2	24.4
0.7100	0.8300	2	38.2
0.7100	0.5200	2	43.2

Tabla 6: Valores de la población inicial y la población final obtenidos obtenidos en el segundo escenario.

Con esta tabla, se demuestra que la implementación paralela se puede ejecutar bajo una gran cantidad de hebras concurrentes y llegar a los mismos resultados en todos los casos. Sin embargo, se debe considerar la sincronización requerida para ello.

En la Figura 26 se presentan los diferentes tiempos de ejecución en segundos, mencionados en un comienzo, en función del número de núcleos utilizados en el sistema para las diferentes versiones implementadas de EVOCA. Cada punto en el gráfico representa el tiempo que tardo una configuración específica bajo las diferentes métricas, en el eje x se muestra la cantidad de núcleos que fueron utilizados y en el eje y se muestra el tiempo en segundos que se demora en completar su ejecución. Además, se agregó una línea de tendencia que se ajusta a los puntos, lo que permite visualizar la tendencia general de los resultados. En la Tabla 7 se presenta con mayor detalle cuales fueron los tiempos tomados para cada una de las ejecuciones bajos las 3 métricas.

Por otro lado, en las figuras 27 y 28 se muestra cual fue la aceleración y la eficiencia de las diferentes implementaciones al aplicar la ley de Amdahl sobre los resultados obtenidos anteriormente. En la Tablas 8, se presenta en detalle cuales fueron los valores obtenidos para cada uno de los casos de $n > 1$. Notar que no existe aceleración y la eficiencia es máxima al considerar un único núcleo (versión secuencial).

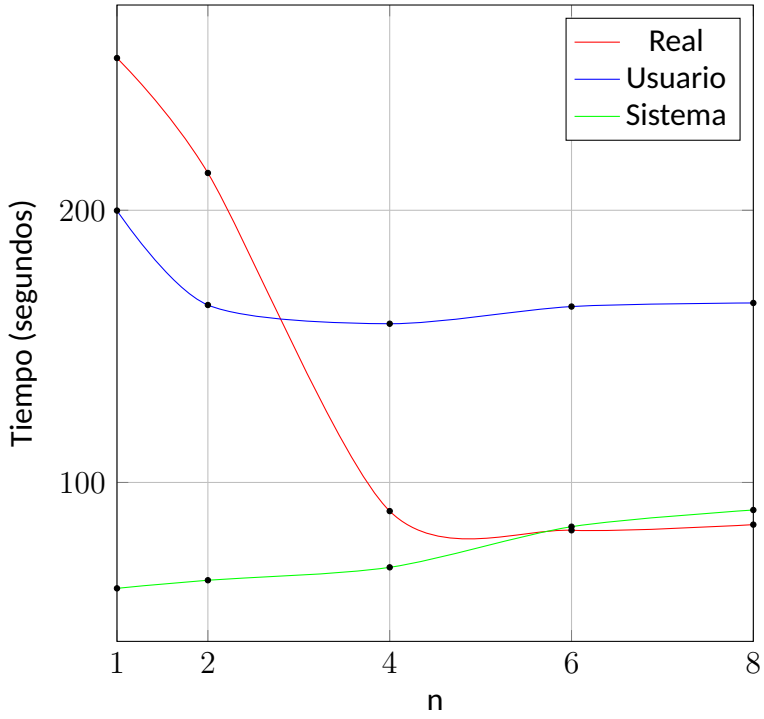


Figura 26: Tiempo de Ejecución en función de n.

Programa	Tiempo Real	Tiempo de Usuario	Tiempo del Sistema
EVOCA secuencial	00:04:16	00:03:20	00:01:01
EVOCA paralelo (2)	00:03:34	00:02:45	00:01:04
EVOCA paralelo (4)	00:01:29	00:02:38	00:01:09
EVOCA paralelo (6)	00:01:22	00:02:45	00:01:24
EVOCA paralelo (8)	00:01:24	00:02:46	00:01:30

Tabla 7: Comparación de tiempos de ejecución entre implementación paralela y secuencial en el segundo escenario.

n	**Speedup**	**Eficiencia**	**F_parallel**	**maxSpeedup**
2	1.12	0.60	0.33	1.49
6	3.12	0.52	0.82	5.42
8	3.05	0.38	0.77	4.31

Tabla 8: Tabla de Speedup, Eficiencia, F_parallel y maxSpeedup en función de n

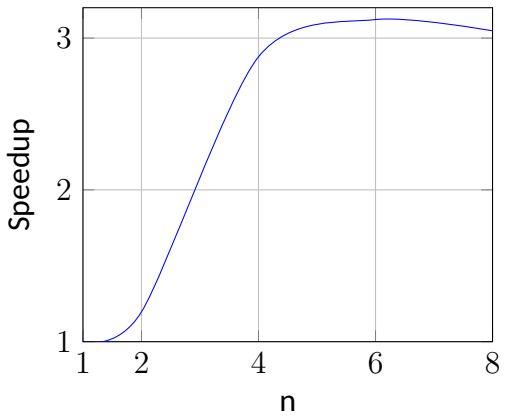


Figura 27: Speedup en función de n.

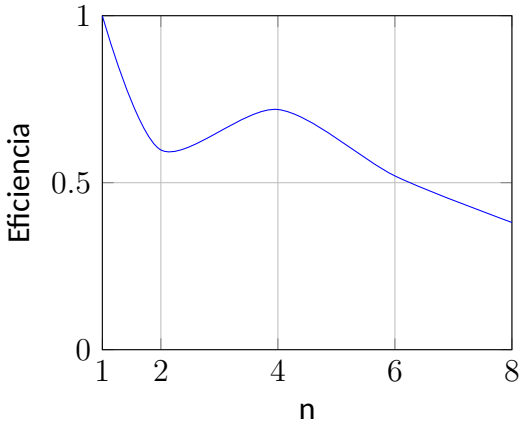


Figura 28: Eficiencia en función de n.

Los resultados que se obtuvieron fueron, en primer lugar, que las ejecuciones de PEVOCA presentan una aceleración casi lineal, mientras que el número de núcleos (n) utilizados sea menor o igual al número de núcleos físicos de la computadora, en este caso 4 núcleos físicos con hyperthreading. Por otro lado, también se logra observar un comportamiento dinámico en cuanto a la eficiencia, con un patrón que inicialmente disminuye, luego aumenta y finalmente vuelve a disminuir. Este patrón nos indica dos aspectos importantes. En primer lugar, al utilizar un valor de $n = 2$, el cálculo de aptitudes se lleva a cabo secuencialmente, con una única hebra encargada de realizar todos los cálculos. Como resultado, la eficiencia no alcanza su punto óptimo, lo que se hace evidente al mostrar un aumento al pasar n de 2 a 4. Sin embargo, al aumentar el número de núcleos, la eficiencia comienza a disminuir debido al uso de hyperthreading, que no resulta tan eficiente como asignar un núcleo físico diferente para cada hilo. Aunque el hyperthreading permite ejecutar varios hilos simultáneamente, el hecho de que estos compartan algunos recursos del núcleo afecta el rendimiento real al pasar de $n = 4$ a $n = 8$. Esto se refleja en una menor aceleración de los cálculos y una disminución de la eficiencia general. Es más, al aumentar de $n = 6$ a $n = 8$, la aceleración disminuye en lugar de aumentar.

En segundo lugar, también se logra observar que al considerar un número muy grande de hebras, se genera un aumento proporcional en el tiempo del sistema. Este comportamiento indica que la sincronización y la comunicación entre las hebras se vuelve cada vez más costoso. Además, cabe destacar que si bien el tiempo requerido es pequeño (1-2 minutos aproximadamente), si existe un tiempo asociado al costo de la sincronización de las hebras concurrentes. Por lo tanto, si la ejecución de las R semillas en algoritmo objetivo es menor que el tiempo de sincronización, el tiempo requerido en la versión secuencial será menor que el de la implementación en paralelo.

Por ultimo, se logra advertir que el tiempo de ejecución en modo usuario en un comienzo disminuye, pero luego vuelve aumentar, lo que sugiere que no existe un comportamiento predeterminado, ya que al estar asociado al tiempo total utilizado por la CPU, se considera el tiempo de las hebras individuales como si fueran secuenciales. En rigor, este tiempo debiese permanecer constate, pero sus cambios o oscilaciones se pueden deber a factores externos, como por ejemplo:

el estado del sistema cuando se ejecuto el programa (muchos procesos en cada núcleo) o en la demora que existe sobre el manejo de archivos. Sin embargo, se puede apreciar que al considerar un número de hebras mayores que $n = 6$, la distancia entre los puntos asociados al tiempo real y el tiempo de usuario, si se mantiene constante. Por lo que se puede establecer un limite máximo en cuanto al tiempo de uso de la cpu.

4.3. AK v/s GA-NK

Al comparar ambas implementaciones, por un lado, se puede apreciar, a medida que se incrementa el número de núcleos utilizados, se observa una notable disminución en el tiempo real requerido para completar su ejecución. Sin embargo, en GA-NK, también se identifica un comportamiento interesante: a medida que se continúa aumentando el número de núcleos, el beneficio en términos de reducción del tiempo de ejecución empieza a disminuir hasta alcanzar cierto umbral. Este comportamiento sugiere la existencia de un limite máximo de mejora al utilizar la implementación paralela, lo que significa que no se obtenga una mejora proporcional al incrementar indefinidamente el número de núcleos, es más, en este caso se logra alcanzar el limite al considerar la ejecución de 6 hebras concurrentes. Esto sugiere que dado un $R > n$, al implementar hyperthreading seguira aumentando la aceleración del algoritmo, siempre que la reducción de rendimiento sea menor que el aumento de utilizar la implementación.

Por otro lado, al considerar $n = 4$, se obtiene que ambas implementaciones logran alcanzan su mayor eficiencia. Para GA-NK estuvo cerca del 72 %, mientras que para AK se obtuvo un 63 %. Lo que se traduce directamente en una menor aceleración, donde se obtuvo que la aceleración de GA-NK fue igual a 2.88 y AK fue igual a 2.51. Sin embargo, cabe destacar que en el escenario de AK, la reducción en el tiempo de ejecución es mucho mayor que GA-NK, aún cuando su aceleración es menor. Este comportamiento indica que la reducción de tiempo es proporcional a la aceleración y al tiempo empleado en la ejecución. Esto implica que para proceso más largos, la disminución de tiempo sea mayor y su uso sea más efectivo.

Por último, en base a lo descrito anteriormente, se puede determinar que solo es posible hacer uso de la concurrencia cuando se tiene exactamente una hebra por núcleo, dado que se requiere que cada hebra tenga su propia instancia/semilla en la memoria de dicho núcleo. Sin embargo, en el caso de contar con hypertreading, se debe considerar además las limitaciones que esto implica. De esta manera, en el mejor de los casos, se debe tener exactamente R hebras ejecutando R semillas en $n = R + 1$ núcleos físicos diferentes (considerando la hebra maestra).

4.4. Discusión

La evaluación de la implementación paralela en diferentes algoritmos objetivo arrojó resultados altamente positivos. En general, se puede concluir que la implementación paralela, haciendo uso de múltiples núcleos, ofrece una mejora significativa en el tiempo de ejecución en comparación

con la implementación secuencial.

No obstante, solo cuando la ejecución de R semillas del algoritmo objetivo es más rápida que la sincronización de las n hebras concurrentes, se presenta un obstáculo en la implementación paralela, en la cual la versión secuencial obtiene mejores tiempos. Sin embargo, en esta circunstancia particular, la necesidad de una menor duración de ejecución resulta irrelevante al considerar tiempos globales considerablemente menores.

Por otro lado, si no se considera este caso, los resultados demuestran que al aumentar el número de hebras utilizadas, se logra una disminución considerable en el tiempo requerido para completar la tarea, lo que indica que el paralelismo es una estrategia efectiva para mejorar el rendimiento del programa. Sin embargo, es relevante considerar que la relación entre el número de hebras y la reducción del tiempo de ejecución alcanza un límite máximo. A medida que se incrementa su cantidad, el beneficio en términos de reducción del tiempo disminuye debido a los puntos críticos a los que se enfrenta, por un lado, derivados de la sincronización entre las hebras y, por otro lado, de la restricción impuesta por el número máximo de núcleos físicos presentes en el sistema.

CAPÍTULO 5
CONCLUSIONES

En esta memoria se ha llevado a cabo el diseño e implementación de una estrategia para paralelizar el sintonizador de parámetros EVOCA, con el propósito de reducir el tiempo necesario para encontrar configuraciones adecuadas en los algoritmos objetivo, sin comprometer la calidad de sus resultados.

Se realizó una exhaustiva revisión de la literatura relacionada con sintonizadores, paralelismo y algoritmos evolutivos. Se han apreciado diferentes métodos para resolver el problema de sintonización, diversas arquitecturas y tipos de paralelismo, así como un análisis más profundo sobre los algoritmos evolutivos, sus clasificaciones y posibles implementaciones en paralelo. Esta revisión permitió establecer una base teórica y una comprensión de los enfoques y metodologías existentes en el campo de sintonización de parámetros.

La revisión bibliográfica fue esencial para fundamentar el diseño de la implementación paralela del sintonizador EVOCA. El diseño de la arquitectura en paralelo para el sintonizador EVOCA fue un paso crucial en el logro de los objetivos propuestos. La estrategia de paralelización se basó en dividir el proceso de búsqueda en múltiples subprocesos, los cuales se ejecutan de forma concurrente al hacer uso de: (1) una arquitectura MIMD, (2) múltiples núcleos en el procesador y (3) al utilizar la programación en multiprocesadores (pthreads). Esta distribución de carga de trabajo permitió explorar diferentes combinaciones de parámetros de manera más eficiente, reduciendo el tiempo total requerido para encontrar configuraciones adecuadas. Asimismo, la implementación paralela se adaptó adecuadamente para su compatibilidad con diversos algoritmos objetivo, lo que aseguró su aplicabilidad en una amplia gama de contextos.

En consecuencia, al seleccionar un número adecuado de hebras para la implementación paralela, es necesario considerar cuidadosamente el equilibrio entre el rendimiento deseado y los recursos disponibles en el sistema. Es importante realizar un análisis detallado de las características del programa y del hardware en el que se ejecutará para determinar la configuración más adecuada que permita obtener el máximo beneficio del paralelismo y una mejora significativa en el tiempo de ejecución. No obstante, se destaca que en todas las ejecuciones realizadas se obtienen las mismas poblaciones finales que su versión secuencial, lo que indica que la paralelización no afectó la calidad de las soluciones obtenidas.

En resumen, la implementación paralela del sintonizador de parámetros EVOCA representa una solución eficiente y prometedora para abordar los desafíos asociados a la búsqueda de configuraciones adecuadas en algoritmos objetivos complejos y con un gran número de parámetros, se observó una reducción significativa en el tiempo de ejecución sin comprometer la calidad de las configuraciones obtenidas. Esta aceleración del proceso de búsqueda representa un avance significativo en la optimización de algoritmos, ya que los diseñadores pueden obtener resultados de alta calidad en un menor tiempo, lo que les permite dedicar recursos a otros proyectos de

investigación y el desarrollo de nuevas investigaciones de manera más expedita.

Finalmente, es de suma importancia destacar la relevancia de seguir investigando y mejorando la implementación del sintonizador EVOCA. Explorar y desarrollar nuevas implementaciones permitirá continuar el avance en la investigación de sintonizadores y mejorar su rendimiento bajo diversos contextos. Además permitirá que EVOCA siga siendo una herramienta valiosa y relevante para la optimización de algoritmos.

Como trabajo futuro, en primer lugar, se sugiere implementar una modificación en los operadores de selección, cruzamiento y mutación. Al explorar diferentes estrategias para estos operadores en el proceso de búsqueda, se podría mejorar aún más el rendimiento del sintonizador EVOCA. La investigación y aplicación de operadores más avanzados y adaptativos, ajustados a las características específicas de cada algoritmo objetivo y su espacio de búsqueda, podrían conducir a una mayor convergencia y exploración del espacio de soluciones.

Otra dirección que se puede tomar es la descentralización de la población y analizar su comportamiento en base a los nuevos parámetros que se deben configurar en el sintonizador. Esto quiere decir que, en lugar de mantener una única población centralizada para la búsqueda, se podría implementar una estrategia de descentralización con múltiples subpoblaciones trabajando de forma independiente en diferentes regiones del espacio de búsqueda. La implementación de un mecanismo para intercambiar información entre estas subpoblaciones periódicamente permitiría una exploración más exhaustiva del espacio de soluciones. Esto ayudaría a evitar el estancamiento en óptimos locales y fomentaría una búsqueda global más efectiva. Se recomienda, en particular, utilizar valores predefinidos en la literatura en los nuevos parámetros a configurar previamente en el sintonizador, para asegurar un mejor desempeño del algoritmo en este enfoque.

Además, se sugiere considerar la ejecución paralela en un ambiente de computación distribuida en la nube. Es decir, en lugar de limitar la paralelización a un enfoque de hebras en una máquina local, se puede aprovechar de una infraestructura de computación distribuida en la nube para ejecutar múltiples aplicaciones del sintonizador EVOCA en diferentes nodos de manera simultánea. Esta estrategia aumentaría la capacidad de procesamiento y reduciría aún más el tiempo de ejecución necesario para encontrar configuraciones óptimas. Es más, esto abre más posibilidades, ya que se puede plantear una optimización de los recursos en la nube. Esto implica que, dado el sintonizador en la nube, se presenta la oportunidad para optimizar la asignación de recursos, la cantidad de nodos, el tamaño de las instancias y el balanceo de carga entre ellos.

Por último, se sugiere considerar una implementación híbrida que combine los enfoques mencionados anteriormente. Al combinar estos enfoques, se podría obtener una sinergia que permita una búsqueda más rápida y efectiva de configuraciones óptimas para los algoritmos objetivo. Por ejemplo, si se considera la descentralización de la población, que facilitaría una exploración exhaustiva del espacio de soluciones, evitando el estancamiento en óptimos locales y fomentando una búsqueda global más eficiente. Con la ejecución paralela en la nube, que proporcionaría un mayor poder de procesamiento, reduciendo aún más el tiempo necesario para encontrar las mejores configuraciones, se podría beneficiar del uso de ambas.

REFERENCIAS BIBLIOGRÁFICAS

Aarti, E. (2019, 08). Role of genetic algorithm in soft computing. *Think India Journal*. (Research journal for publication)

Aguilar, J., Leiss, E., y cols. (2004). Introducción a la computación paralela. *Venezuela: Gráficas Quinteto*.

Akl, S. G. (1989). *The design and analysis of parallel algorithms*. Prentice-Hall, Inc.

Alba, E., y Tomassini, M. (2002). Parallelism and evolutionary algorithms. *IEEE transactions on evolutionary computation*, *6*(5), 443–462.

Bailey, M. (2017). *Parallel programming: Speedups and amdahl's law*.

A beginner's guide to tuning methods. (2014). *Applied Soft Computing*, *17*, 39-51.

Birattari, M., Stützle, T., Paquete, L., Varrentrapp, K., y cols. (2002). A racing algorithm for configuring metaheuristics. En *Gecco* (Vol. 2).

Bouneffouf, D., y Claeys, E. (2021). Online hyper-parameter tuning for the contextual bandit. En *ICASSP 2021-2021 IEEE international conference on acoustics, speech and signal processing (ICASSP)* (pp. 3445–3449).

Butenhof, D. R. (1997). *Programming with posix threads*. Addison-Wesley Professional.

Cantú-Paz, E. (2000). A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, *12*(2), 141–171.

Chau, D. P., Thonnat, M., Bremond, F., y Corvee, E. (2014). Online parameter tuning for object tracking algorithms. *Image and Vision Computing*, *32*(4), 287–302.

Chiang, W.-C., Russell, R., Xu, X., y Zepeda, D. (2009). A simulation/metaheuristic approach to newspaper production and distribution supply chain problems. *International Journal of Production Economics*, *121*(2), 752–767.

Dorigo, M., Maniezzo, V., y Colorni, A. (1996). Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, *26*(1), 29–41.

Downey, A. B. (2016). *The little book of semaphores* (2nd ed.). Needham, MA: Green Tea Press.

Drake, J. (2015, 01). *Benchmark instances for the multidimensional knapsack problem*. doi: 10.13140/2.1.3578.9122

Eiben, A., y Smith, J. (2015). *Introduction to evolutionary computing* (2nd ed.). Berlin, Germany: Springer.

Fan, L., y Mumford, C. L. (2010). A metaheuristic approach to the urban transit routing problem. *Journal of Heuristics*, *16*(3), 353–372.

Feo, T. A., y Resende, M. G. (1989). A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, *8*(2), 67-71.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE transactions on computers*, *100*(9), 948–960.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, *13*(5), 533-549. (Applications of Integer Programming)

Hennessy, J. L., y Patterson, D. A. (2011). *Computer architecture: A quantitative approach* (5th ed.). San Francisco, CA: Morgan Kaufmann.

Herlihy, M., y Shavit, N. (2012). *The art of multiprocessor programming*. Morgan Kaufmann.

Holland, J. H. (1992). Genetic algorithms. *Scientific american*, *267*(1), 66–73.

Hutter, F., Hoos, H. H., y Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. En *International conference on learning and intelligent optimization* (pp. 507–523).

Hutter, F., Hoos, H. H., Leyton-Brown, K., y Stützle, T. (2009). Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, *36*, 267–306.

Hutter, F., Hoos, H. H., y Stützle, T. (2007). Automatic algorithm configuration based on local search. En *Aaai* (Vol. 7, pp. 1152–1157).

Karp, R. M. (1972). Reducibility among combinatorial problems. En R. E. Miller y J. W. Thatcher (Eds.), *Complexity of computer computations* (pp. 85–103). New York: Plenum.

Kauffman, S. A., y Weinberger, E. D. (1989). The nk model of rugged fitness landscapes and its application to maturation of the immune response. *Journal of Theoretical Biology*, *141*(2), 211-245. doi: https://doi.org/10.1016/S0022-5193(89)80019-0

Kennedy, J., y Eberhart, R. (1995). Particle swarm optimization. En *Proceedings of the ieee international conference on neural networks* (pp. 1942–1948).

Khuri, S., Bäck, T., y Heitkotter, J. (1994, 10). The zero/one multiple knapsack problem and genetic algorithms.

doi: 10.1145/326619.326694

Kim, J.-H., y Myung, H. (1997, 08). Evolutionary programming techniques for constrained optimization problems. *Evolutionary Computation, IEEE Transactions on*, *1*, 129 - 140. doi: 10.1109/4235.687880

Kirkpatrick, S., Gelatt, C. D., y Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671-680.

Laurenso, F., Martin, O., y Stutzle, T. (1994). Iterated local search: A framework for intensification and diversification. *Frontiers in Heuristics*, *56*, 147–161.

Lee, C.-H., Park, S.-H., y Kim, J.-H. (2000). Topology and migration policy of fine-grained parallel evolutionary algorithms for numerical optimization. En *Proceedings of the 2000 congress on evolutionary computation. cec00 (cat. no.00th8512)* (Vol. 1, p. 70-76 vol.1). doi: 10.1109/CEC.2000.870277

Leguizamon, G., y Michalewicz, Z. (1999). A new version of ant system for subset problems. En *Proceedings of the 1999 congress on evolutionary computation-cec99 (cat. no. 99th8406)* (Vol. 2, p. 1459-1464 Vol. 2). doi: 10.1109/CEC.1999.782655

López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L. P., Birattari, M., y Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, *3*, 43–58.

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., y Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, *3*, 43-58.

Marinelli, M., Dell'Orco, M., y Sassanelli, D. (2015). A metaheuristic approach to solve the flight gate assignment problem. *Transportation Research Procedia*, *5*, 211-220. (SIDT Scientific Seminar 2013)

Nannen, V., y Eiben, A. (2007b). Relevance estimation and value calibration of evolutionary algorithm parameters. *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, 1034-1039.

Nannen, V., y Eiben, A. E. (2007a). Efficient relevance estimation and value calibration of evolutionary algorithm parameters. En *2007 ieee congress on evolutionary computation* (pp. 103–110).

Osorio, M., y Cuaya, G. (2005). Hard problem generation for mkp. En *Sixth mexican international conference on computer science (enc'05)* (p. 290-295). doi: 10.1109/ENC.2005.22

Pelikan, M. (2005). Hierarchical bayesian optimization algorithm: Toward a new generation of evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, *9*(3), 281–295.

Pelikan, M., Sastry, K., Goldberg, D., Butz, M., y Hauschild, M. (2009, 07). Performance of evolutionary algorithms on nk landscapes with nearest neighbor interactions and tunable overlap. En (p. 851-858). doi: 10.1145/1569901.1570018

Petrovic, D., Morshed, M., y Petrovic, S. (2011). Multi-objective genetic algorithms for scheduling of radiotherapy treatments for categorised cancer patients. *Expert Systems with Applications*, *38*(6), 6994–7002.

Porta, J., Parapar, J., Doallo, R., Rivera, F. F., Santé, I., y Crecente, R. (2013). High performance genetic algorithm for land use planning. *Computers, Environment and Urban Systems*, *37*, 45-58. doi: https://doi.org/10.1016/j.compenvurbsys.2012.05.003

Rathomi, M., y Pulungan, R. (2018, 04). A coarse-grained parallelization of genetic algorithms. *International Journal of Advances in Intelligent Informatics*, *4*, 1. doi: 10.26555/ijain.v4i1.137

Riff, M.-C., y Montero, E. (2013). A new algorithm for reducing metaheuristic design effort. En *2013 ieee congress on evolutionary computation* (pp. 3283–3290).

Rocke, D. (2000, 03). Genetic algorithms + data structures = evolution programs by z. michalewicz. *Journal of the American Statistical Association*, *95*, 347-348. doi: 10.2307/2669583

Roosta, S. H. (2012). *Parallel processing and parallel algorithms: theory and computation*. Springer Science & Business Media.

Saidu, C. I., Obiniyi, A., y Ogedebe, P. O. (2015). Overview of trends leading to parallel computing and parallel programming. *British Journal of Mathematics & Computer Science*, *7*(1), 40.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, *27*(3), 379-423. doi: 10.1002/j.1538-7305.1948.tb01338.x

Sun Microsystems, Inc. (2008). *Multithreaded programming guide*. USA: Prentice-Hall, Inc.

Syswerda, G. (1991). A study of reproduction in generational and steady-state genetic algorithms. En *Foundations of genetic algorithms* (Vol. 1, pp. 94–101). Elsevier.

Talbi, E.-G. (2009). *Metaheuristics: from design to implementation*. John Wiley & Sons.

von Neumann, J. (1993). First draft of a report on the edvac. *IEEE Annals of the History of Computing*, *15*(4), 27-75. doi: 10.1109/85.238389

ANEXO

Nombre del archivo	N	K	Versión
nk_20_2.1	20	2	1
nk_20_2.2	20	2	2
nk_20_2.3	20	2	3
nk_20_2.4	20	2	4
nk_20_2.5	20	2	5
nk_20_2.6	20	2	6
nk_20_2.7	20	2	7
nk_20_2.8	20	2	8
nk_20_2.9	20	2	9
nk_20_2.10	20	2	10
nk_20_3.1	20	3	1
nk_20_3.2	20	3	2
nk_20_3.3	20	3	3
nk_20_3.4	20	3	4
nk_20_3.5	20	3	5
nk_20_3.6	20	3	6
nk_20_3.7	20	3	7
nk_20_3.8	20	3	8
nk_20_3.9	20	3	9
nk_20_3.10	20	3	10
nk_20_4.1	20	4	1
nk_20_4.2	20	4	2
nk_20_4.3	20	4	3
nk_20_4.4	20	4	4
nk_20_4.5	20	4	5
nk_20_4.6	20	4	6
nk_20_4.7	20	4	7
nk_20_4.8	20	4	8
nk_20_4.9	20	4	9
nk_20_4.10	20	4	10
nk_20_5.1	20	5	1
nk_20_5.2	20	5	2
nk_20_5.3	20	5	3
nk_20_5.4	20	5	4
nk_20_5.5	20	5	5
nk_20_5.6	20	5	6
nk_20_5.7	20	5	7
nk_20_5.8	20	5	8
nk_20_5.9	20	5	9
nk_20_5.10	20	5	10
nk_20_6.1	20	6	1
nk_20_6.2	20	6	2
nk_20_6.3	20	6	3
nk_20_6.4	20	6	4
nk_20_6.5	20	6	5
nk_20_6.6	20	6	6
nk_20_6.7	20	6	7
nk_20_6.8	20	6	8
nk_20_6.9	20	6	9
nk_20_6.10	20	6	10

Tabla 9: Nombre de archivos de instancias y sus características (completo)