
MPAI-Boost: Optimization of AI algorithms applying high-performance computing techniques

MPAI-Boost: Optimización de algoritmos de IA aplicando técnicas
enfocadas en cómputo de alto rendimiento



TRABAJO DE FIN DE GRADO

DANIEL PIZARRO GALLEGOS

Director:
Alberto Núñez Covarrubias

Facultad de Informática
Universidad Complutense de Madrid

19 de septiembre del 2024

Autorización de difusión

Autor

Daniel Pizarro Gallego

Fecha

Madrid, XX de Septiembre de 2024.

El abajo firmante, matriculado en el Grado de Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: Optimización de algoritmos de IA aplicando técnicas enfocadas en cómputo de alto rendimiento, realizado durante el curso académico 2023-2024 bajo la dirección de Alberto Núñez Covarrubias en el Departamento de Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

El trabajo que se presenta, se enfoca en la optimización de algoritmos de Inteligencia Artificial (IA), mediante el uso de MPI (Message Passing Interface), una biblioteca estándar desarrollada para el cómputo de alto rendimiento. El objetivo principal consiste en reducir el tiempo de ejecución de algoritmos explotando el paralelismo de los recursos de cómputo y la memoria distribuida. Esta tarea es especialmente relevante debido al alto coste computacional y de recursos que implica entrenar o ejecutar estos modelos.

Este proyecto incluye una descripción de los fundamentos teóricos de los algoritmos que se van a implementar, así como el funcionamiento de la biblioteca MPI. Una vez puesto en contexto, se desarrollan en profundidad las estrategias implementadas para mejorar los algoritmos. Y además, se ha realizado un estudio empírico para analizar las mejoras desarrolladas a lo largo del proyecto. Este estudio incluye la ejecución de las mejoras en un sistema distribuido que consta de 128 núcleos de CPU y 256 GB de RAM.

Palabras clave

IA, aprendizaje automático, MPI, speedup, memoria distribuida, redes neuronales, algoritmos evolutivos, clustering

Abstract

The work presented focuses on the optimization of Artificial Intelligence (AI) algorithms, using MPI (Message Passing Interface), a standard library developed for high-performance computing. The main objective consists of reducing the execution time of AI algorithms, exploiting the parallelism of computing resources and distributed memory. This task is especially relevant due to the high computational and resource cost involved in training or running these models.

This project includes a description of the theoretical foundations of the algorithms that will be implemented. Moreover, clarifying the functioning of the MPI library. Once put in context, the strategies employed to enhance the algorithms are thoroughly elaborated upon. And in addition, empirical study has been carried out to analyze the improvements developed throughout the project. This evaluation includes running the algorithms on a supercomputer with 128 cores.

Keywords

IA, machine learning, MPI, speedup, distributed memory, neural network, Evolutionary algorithm, clustering

Índice general

| | |
|--|------------|
| Índice | I |
| Dedicatoria | III |
| 1. Introducción | 1 |
| 1.1. Definición y alcance del proyecto | 1 |
| 1.2. Motivación | 3 |
| 1.3. Objetivo | 4 |
| 1.4. Estructura del documento | 5 |
| 2. Contextualización | 6 |
| 2.1. MPI | 6 |
| 2.2. Aprendizaje por Refuerzo | 9 |
| 2.2.1. Algoritmo Q-Learning | 9 |
| 2.2.2. Deep Q-Network (DQN) | 10 |
| 2.3. Aprendizaje No-Supervisado | 11 |
| 2.3.1. Clustering jerárquico aglomerativo | 12 |
| 2.3.2. Clustering Basados en particiones: K-Medias | 13 |
| 2.4. Aprendizaje Supervisado | 14 |
| 2.4.1. K-Vecinos más Cercanos - KNN | 15 |
| 2.4.2. Redes Neuronales | 16 |
| 2.5. Programación Evolutiva | 18 |
| 3. Diseño e Implementaciones | 19 |
| 3.1. Programas sencillos | 19 |
| 3.2. Algoritmos de Clustering | 25 |
| 3.2.1. Jerárquico Aglomerativo | 25 |
| 3.2.2. K-Medias | 27 |
| 3.2.3. K-Vecinos más cercanos (KNN) | 29 |
| 3.3. Aprendizaje por refuerzo | 32 |
| 3.4. Algoritmos Evolutivos | 36 |
| 3.5. Redes Neuronales | 39 |
| 4. Estudio empírico | 43 |
| 4.1. Programas sencillo | 44 |
| 4.1.1. Ordenaciones | 44 |
| 5. Conclusiones y trabajo futuro | 69 |

Dedicatoria

*A mis padres, por que gracias a ellos soy quien
soy hoy*

Capítulo 1

Introducción

En este capítulo se presenta una perspectiva general del contexto en el que se ha llevado a cabo el proyecto. Además de los desafíos enfrentados durante su desarrollo para alcanzar las contribuciones mencionadas, se detallan cada uno de los propósitos perseguidos en él.

1.1. Definición y alcance del proyecto

El desarrollo de las Inteligencias Artificiales en nuestra sociedad ha sido un fenómeno de gran relevancia, además de popular, en los últimos años. Estas tecnologías han llegado para quedarse. Están mejorando nuestra calidad de vida, desde la automatización de tareas hasta la asistencia virtual [20], estas IAs desempeñan un papel cada vez más importante en nuestro día a día. Con el advenimiento del Internet de alta velocidad y la proliferación de datos, las empresas tecnológicas se enfrentan a la necesidad creciente de desarrollar servicios de alta calidad en un mercado muy competitivo. Actualmente, se invierte mucho dinero y tiempo en mejorar y diseñar algoritmos, para implementar Inteligencias Artificiales para el acceso público[1].

El entrenamiento y ejecución de estos algoritmos para modelar inteligencias artificiales consumen mucha energía, además de provocar una cantidad excesiva de emisiones de CO₂. La empresa tecnológica *Hugging Face* estimó que el entrenamiento de *BLOOM*[16] emitió 25 toneladas de CO₂, cifra que se duplicó al contar el coste de producción del equipo informático usado[14]. Los investigadores se están enfocando en evaluar y reducir el impacto ambiental

de las tecnologías de IA. Una prueba de ello es el desarrollo de *CarbonTracker*[17] (CTI), un equipo de especialistas financieros que asumen el riesgo climático como realidad de los mercados financieros actuales. El objetivo de esta herramienta es predecir y reducir la huella de carbono de las etapas de entrenamiento de los modelos de IA[18].

El uso de la programación distribuida, más específicamente, aplicaciones basadas en MPI, permite tener varios procesos ejecutándose en paralelo, dividiendo la carga de trabajo y así reduciendo el tiempo de ejecución. Al contrario de la memoria compartida, en la cual se pueden dar problemas de memoria y sincronización, cada proceso generado tiene su propia memoria local, evadiendo estos problemas. Sin embargo, hay que diseñar implementaciones correctas y eficientes para no tener más carga espacial de la esperada. Las conexiones entre los procesos se pueden moldear para maximizar la eficiencia y reducir el tiempo de cómputo. Una de las más populares es el modelo master-worker. El proceso master se encarga de distribuir el trabajo a los workers, para, en paralelo, ejecutar la misma tarea pero con conjuntos mucho más reducidos. Al finalizar la tarea, el worker envía su salida, y si el proceso master no ha terminado, espera para recibir más datos. MPI permite la comunicación eficiente entre procesos, mejorando la escalabilidad y reduciendo el tiempo de procesamiento. Esta metodología, además de mejorar el rendimiento, también simplifica la gestión de recursos, mejorando la utilización del hardware disponible en el sistema.

Los algoritmos de IA suelen manejar un vasto número de datos para entrenar y evaluar los modelos deseados. Por eso es fundamental diseñar estrategias para distribuir los datos, y dividir las cargas de trabajo de manera equitativa, controlando el flujo de datos para evitar cuellos de botella, y equilibrar los recursos disponibles. Esto incluye, además de la optimización del tiempo de ejecución, una gestión efectiva de la memoria (complejidad espacial) y minimizar la latencia (tiempo de espera en transmitir los paquetes de información en una red) en la comunicación entre los procesos. En este proyecto se diseñarán e implementarán varias optimizaciones para diferentes algoritmos de IA, con el objetivo de reducir el tiempo de ejecución.

1.2. Motivación

Actualmente hay muchas implementaciones de algoritmos de IA. Scikit learn es una biblioteca de Python perfecta para probar cualquier técnica. Esta biblioteca, como la mayoría, ejecuta los algoritmos de manera secuencial, sin dividir la carga de trabajo. Las implementaciones están estudiadas y perfeccionadas para realizar los cálculos en un único proceso, pero se puede reducir el tiempo de ejecución aplicando paralelismo.

Las arquitecturas distribuidas junto con las técnicas de cómputo de alto rendimiento (HPC, por sus siglas en inglés), son una de las soluciones más apropiadas para los usuarios finales: científicos y empresas. La búsqueda de un rendimiento óptimo, bajo una perspectiva técnica en sistemas altamente distribuidos, demandan enfoques adaptables y escalables para implementar aplicaciones científicas de alto rendimiento. Sincronizar los procesos, distribuir los datos para aumentar el paralelismo y reducir el overhead, son los desafíos recurrentes en los sistemas distribuidos. La comunicación en el proceso tiene que ser controlada para el correcto funcionamiento, y en algunas ocasiones estas mejoras derivan en implementaciones más complejas.

En estas situaciones, es desafiante controlar las acciones de cada proceso, para obtener la especificación deseada. Cada algoritmo tiene su funcionamiento y su desempeño, al igual que implementación única.

Las empresas tecnológicas buscan mejorar el rendimiento y reducir costos de sus sistemas. Para ello tienen que hacer un uso eficiente de los recursos computacionales. Además de buscar reducir el consumo de energía y las emisiones de CO₂, al entrenar o procesar modelos de IA, puesto que el impacto ambiental está en auge hoy en día. Contando con una gran competitividad en el mercado tecnológico, cualquier mejora, aunque no sea tan significativa, es un avance.

1.3. Objetivo

El objetivo principal de este trabajo es **paralelizar algoritmos de IA, desarrollando varias implementaciones que reduzcan el tiempo de ejecución**. Entre los algoritmos a optimizar se encuentran técnicas como el agrupamiento de individuos, predicción de resultados y la optimización de funciones de evaluación. Todas estas implementaciones han sido desarrolladas en Python, el lenguaje de programación más popular en el ámbito de la inteligencia artificial [21]. Sin embargo, al ser un lenguaje interpretado (el código se traduce en la misma ejecución), aumenta la sobrecarga y hace que el programa sea más lento que la misma implementación en otros lenguajes. Por eso Python es el lenguaje de programación idóneo para aplicar técnicas de cómputo de alto rendimiento y reducir el tiempo de ejecución.

Asimismo, se requerirá alcanzar los siguientes objetivos secundarios:

1. **Diseño de implementaciones escalables y flexibles.** Al diseñar e implementar mejoras de cada algoritmo, se permite la varianza del número de procesos a ejecutar. Esto permite un estudio profundo de las implementaciones realizadas. Al variar el número de procesos se puede comprobar cuál es el número idóneo para cualquier implementación, así calculando el número óptimo de procesos. La flexibilidad en las mejoras permite variar los datos de entrada para que funcione correctamente con un tamaño de dataset variable.
2. **Correcto funcionamiento de los algoritmos.** Al realizar las mejoras, además de mejorar el rendimiento, tienen que tener una cohesión con el algoritmo original. Es decir, si queremos maximizar una función de evaluación, la implementación de la mejora tiene que dar resultados parecidos o mejores. No sirve implementar una mejora que reduzca el tiempo de ejecución pero no de buenos resultados.
3. **Estudio empírico.** Se realizará una evaluación de las mejoras para calcular el aumento del rendimiento. Primero se mide el tiempo que tarda cada algoritmo sin mejoras

y luego las diferentes implementaciones desarrolladas, variando el número de procesos ejecutados. Para finalizar se prueban las mejores implementaciones de cada algoritmo en el sistema distribuido con 128 núcleos.

4. Cálculo del mejor número de procesos para cada mejora, dado un dataset.

1.4. Estructura del documento

El resto de este documento está organizado en los siguientes capítulos:

- Capítulo 2: Contextualización. En este capítulo se proporciona información de cada algoritmo estudiado, para la correcta lectura del trabajo.
- Capítulo 3: Diseño e implementaciones, Comienza describiendo unos ejemplos básicos fuera del ámbito de la inteligencia artificial, seguido de las implementaciones desarrolladas para las diferentes técnicas abordadas.
- Capítulo 4: Estudio empírico, presenta el estudio empírico realizado, el cual consiste en medir las mejoras, con los algoritmos secuenciales, variando el número de procesos y el tamaño de los datos. Para poder medir el speed-up y realizar comparaciones significativas.
- Capítulo 5: Conclusiones y trabajo a futuro.

Capítulo 2

Contextualización

En este capítulo se presenta una breve descripción de los algoritmos de Inteligencia Artificial que se van a profundizar a lo largo del proyecto. Así como los usos y características.

El objetivo de este capítulo es explicar rápidamente los algoritmos, y facilitar la lectura de los capítulos posteriores. Que desarrollos las mejoras realizadas y resultados obtenidos.

2.1. MPI

Message Passing Interface² (MPI) es un estándar para una biblioteca de paso de mensajes, diseñado para funcionar en una amplia variedad de arquitecturas informáticas paralelas. Permite la comunicación entre procesos, mandando y recibiendo mensajes de todo tipo. Comúnmente usado en informática de alto rendimiento⁷ (HPC) y entornos informáticos paralelos, para desarrollar aplicaciones paralelas escalables y eficientes.

Al crear el entorno MPI en una aplicación se ejecutan en paralelo varios procesos, cada uno con su correspondiente id, también llamado rank. El programador elige cuál va a ser el desempeño de los procesos. Por ejemplo, en el modelo *Master-Worker*, el primer proceso (id=0) generalmente, es llamado *Master*, se encarga de distribuir los mensajes entre los demás procesos, llamados *Workers*, para que haya una comunicación eficiente y paralelizar los programas. La Figura 2.1, muestra la comunicación entre los procesos en este modelo. El proceso master reparte los datos, los workers los procesan y los devuelven.

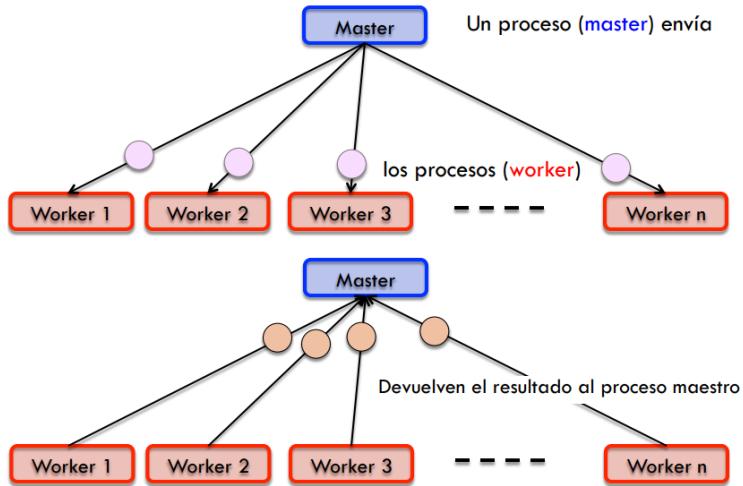


Figura 2.1: Comunicación Master-Worker

Esta técnica para paralelizar utiliza memoria distribuida, es decir, cada proceso tiene su propia memoria local. No tienen que preocuparse por los problemas de la memoria compartida, como la sincronización para el acceso de variables compartidas, condiciones de carrera o deadlocks. Asimismo la memoria compartida no es fácilmente escalable a un gran número de procesadores².

- Single Program Multiple Data (SPMD). Un programa ejecutado en paralelo, donde múltiples procesos se ejecutan en el mismo programa de manera independiente, pero trabajan con diferentes conjuntos de datos. Es un modelo comúnmente utilizado en computación de alto rendimiento y entornos de procesamiento paralelo. La escalabilidad, y eficiencia de este modelo son sus principales ventajas.

Un proceso (el Master) contiene todos los datos del programa, y se encarga de gestionar todos los datos y repartirlos de manera eficiente Figura 2.2. Los mensajes pueden ser:

- Síncronos: al ejecutar la función `recv()` el proceso receptor se queda bloqueado.
- Asíncrono: el receptor no se bloquea, por lo que puede adelantar código mientras espera a recibir el mensaje.
- Broadcast: un mensaje se envía a todos los procesos ejecutados. Los emisores tienen

que llamar a la misma función para recibir el mensaje.

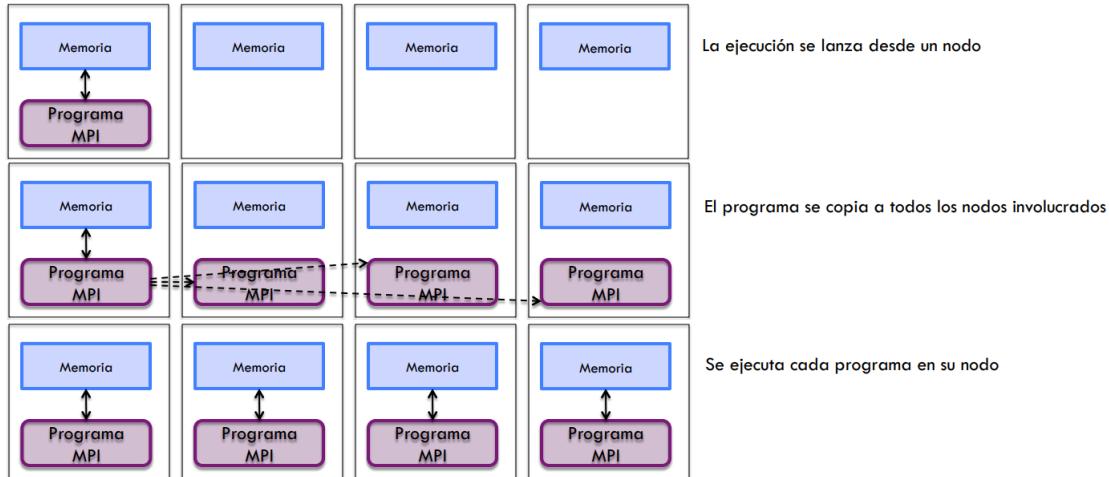


Figura 2.2: Ejecución MPI

Un programa MPI, comparte el mismo código para todos los procesos ejecutados. Un proceso lee el conjunto de datos y los carga en su memoria, para luego dividirlos y enviarlos a los procesos disponibles. Una vez repartido el dataset se ejecutan en paralelo y procesan los datos recibidos. Cuando un proceso finaliza el procesado, envía devuelta los datos procesados.

```

1  from mpi4py import MPI
2  # Al importar la biblioteca en Python se genera el entorno.
3
4  comm = MPI.COMM_WORLD      # Comunicador
5  status = MPI.Status()      # Status
6  myrank = comm.Get_rank()    # id de cada proceso
7  numProc = comm.Get_size()   # Numero de procesadores
8
9  if myrank==0:              # Master
10     # Carga el conjunto de datos. Los divide y envia.
11     # Recibe todos los datos procesados.
12 else:                      # Workers
13     # Recibe el subconjunto de datos que le asigna el Master.
14     # Procesa los datos.
15     # Envia los datos procesados.

```

Figura 2.3: Esquema básico para ejecutar un programa MPI en Python

2.2. Aprendizaje por Refuerzo

Reinforcement Learning (RL), en español, Aprendizaje por Refuerzo, es un tipo de aprendizaje automático donde el agente aprende en base a las decisiones tomadas al interactuar con el entorno. El agente aprende a llegar a una meta o maximizar un cúmulo de recompensas obtenidas al realizar un determinado número de acciones consecutivas, y observar las posibles recompensas al realizar cada acción en los estados del entorno.

El agente aprende a través de prueba y error, explorando las diferentes acciones y aprendiendo cuáles conducen a mejores resultados. No requiere entradas etiquetadas como en el aprendizaje supervisado, aprende con una serie de feedbacks (recompensas o castigos).

Los componentes esenciales del algoritmo son los siguientes:

- Agente que interactúa con el entorno y aprende de él.
- El entorno, con el cual el agente interactúa. Responde a las acciones tomadas por el agente y provee el feedback al agente.
- El conjunto de acciones o decisiones que el agente puede realizar.
- Estados. Situaciones de configuraciones en el entorno.
- Recompensas. Se suele representar como una matriz $R(S,A)$, estados-acción. Almacena el feedback del entorno al realizar una acción en un estado.
- Q-Table $Q(S,A)$. Guarda los valores-Q de las acciones en los estados.
- Condición de finalización. Puede ser desde encontrar la función-Q óptima, hasta realizar un número de épocas (Ejecución por el entorno, hasta finalizar una partida)

2.2.1. Algoritmo Q-Learning

Mezcla entre programación dinámica y Monte Carlo⁹.

Es el más básico de entre los algoritmos de aprendizaje por refuerzo. Se usa para encontrar la mejor política de selección de acciones para un proceso de Decisión de Markov Determinado (MDP en inglés)⁵.

El procedimiento se realiza actualizando iterativamente las estimaciones de calidad de realizar dicha acción en el estado actual, conocido como valor-Q. Este valor representa el feedback que recibirá el agente al realizar la acción desde un estado. Se usa la siguiente fórmula para actualizar los valores.

El agente toma las decisiones de ejecutar una acción dependiendo del hiper parámetro ϵ con valores entre [0,1]. Con un número aleatorio (en el mismo intervalo) calcula la probabilidad de ejecutar la mejor acción aprendida hasta el momento, o una acción aleatoria entre las disponibles. Si el valor es alto, casi siempre se ejecutará la “mejor” y es posible que no aprenda otras formas de alcanzar el objetivo.

$$Q(S, A) = (1 - \alpha) \cdot Q(S, A) + \alpha \cdot \left(R(S, A) + \gamma \cdot \max_i Q(S', A_i) \right)$$

$Q(S, A) \leftarrow$ Es el valor-Q de ejecutar la acción A en el estado S.
 $R(S, A) \leftarrow$ Es la recompensa obtenida al ejecutar la acción A en el estado S.
 $\alpha \leftarrow$ Tasa de aprendizaje. Controla cuánta importancia le da a la nueva información frente a la antigua.
 $\gamma \leftarrow$ Factor de descuento. Determina la importancia de futuras recompensas comparadas con las recompensas inmediatas.
 $\max(Q(S', A_i))$: Es el valor máximo obtenible de realizar las posibles acciones en el estado siguiente.

Este algoritmo se ha aplicado en muchos dominios, como puede ser videojuegos de Atarí⁶, robótica o problemas de optimización. Sin embargo sufre cuando el entorno tiene muchos estados, ya que la complejidad espacial aumenta muchísimo y no es práctico tener las dos matrices.

2.2.2. Deep Q-Network (DQN)

Por los problemas de escalabilidad mencionados anteriormente, se desarrolló el algoritmo de Redes Neuronales Profundas. Combina redes neuronales, con la base de aprendizaje por refuerzo, así eliminando la Q-Table. [TODO...](#)

2.3. Aprendizaje No-Supervisado

Los métodos no supervisados (unsupervised methods) son algoritmos de aprendizaje automático que basan su proceso en un entrenamiento con datos sin etiquetar. Es decir, a priori no se conoce ningún valor objetivo, ya sea categórico o numérico.

La meta de este aprendizaje es encontrar patrones o estructura en los datos proporcionados. Estos algoritmos son útiles en escenarios en los cuales hay escasez de datos etiquetados o no están disponibles.

Hay muchos tipos de técnicas de aprendizaje no supervisado. Como puede ser detección de anomalías, reducción de dimensionalidad o clustering. En este proyecto vamos a reducir el tiempo de ejecución de las técnicas de Clustering que se encargan de agrupar individuos, basándose en alguna medida de similitud. Como no es aprendizaje supervisado, no disponemos de información categorizada previamente, por lo que hay que calcular el número óptimo de clusters, para el cual hay medidas ya estudiadas como el coeficiente de Davies-Bouldin cuyo valor mínimo indica el número óptimo de clusters, Silhouette es parecido al anterior pero con el valor máximo, o Diagramas de codo. Siendo este último el menos preciso debido a que se calcula visualmente, cuando empieza a crearse un codo (la diferencia con el número anterior no es tan pronunciada como en puntos anteriores). Se pueden apreciar los diferentes coeficientes en la Figura 2.4.

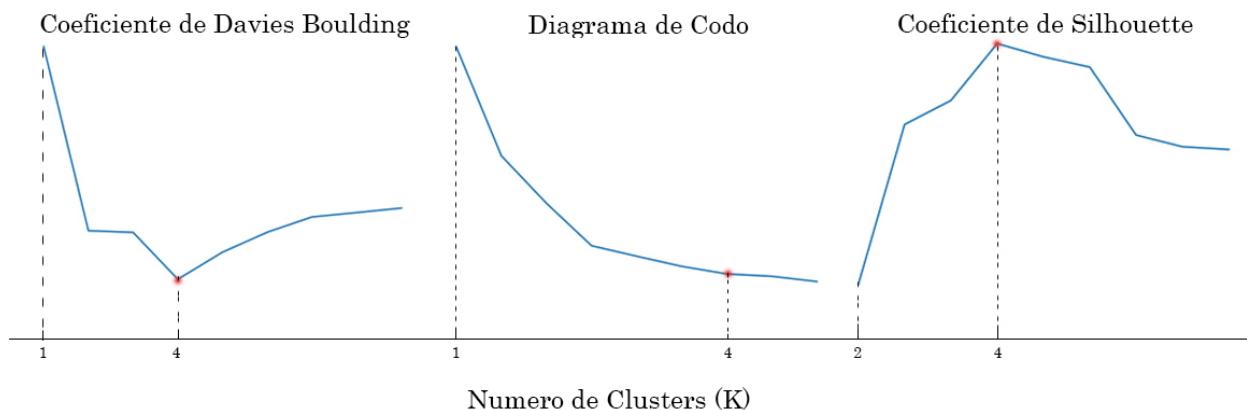


Figura 2.4: Coeficientes

Los llamados métodos jerárquicos¹ tienen por objetivo agrupar clusters para formar uno nuevo o bien separar alguno ya existente para dar origen a otros dos, de tal forma que, si sucesivamente se va efectuando este proceso de aglomeración, se minimice alguna distancia o bien se maximice alguna medida de similitud.

2.3.1. Clustering jerárquico aglomerativo

Este algoritmo usa una matriz para realizar la agrupación de los individuos. Comienza teniendo N cluster, uno por cada individuo de la población. La matriz se representa por las filas, es decir, la fila i-ésima representa el cluster i-ésimo. La matriz se rellena con las distancias entre los clusters, por lo que la celda (i,j) representa la distancia entre el cluster i y el j.

En cada iteración, se busca en la matriz la distancia mínima, y se juntan los clusters que representan la fila i con la columna j. La matriz se actualiza, eliminando la fila y la columna con mayor índice (entre i,j), y actualizando la fila y columna de menor índice. Este proceso se repite hasta que solo haya un cluster. Las distancia entre clusters pueden ser:

- Centroides, cada cluster tiene un centro.
- Enlace simple o compuesto. La distancia entre cluster viene dada por la menor o mayor distancia, respectivamente, entre los individuos que representan cada cluster.

Algorithm 1: Jerárquico Aglomerativo

```

Data: poblacion, C // Número de clusters deseados
Result: agrupacion // Clusters para cada individuo de la población
D := init() // Inicializar la matriz de distancias
while number of rows in matrix > C do
    // Recorrer la matriz en búsqueda del menor valor (i, j)
    // Agrupar los cluster (i, j) y eliminar la fila y columna de mayor índice
    // del agrupacion[max(i,j)];
    // Calcular nuevas distancias al cluster agrupado

```

La complejidad en los enlaces simple y completo tienen un coste cúbico $O(N^3)$, al tener que comparar todos los individuos uno a uno entre dos cluster.

Al finalizar la ejecución se puede representar la agrupación mediante un dendrograma⁴, y comprobar el número óptimo de clusters para la población calculada. Aunque no es igual de preciso como los coeficientes mencionados anteriormente. Ejemplo: Figura 2.5

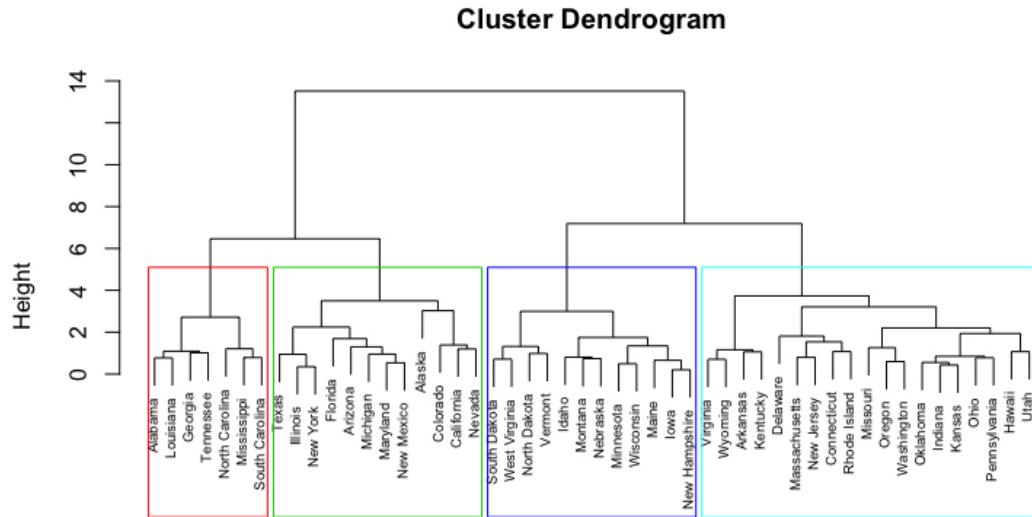


Figura 2.5: Dendograma

2.3.2. Clustering Basados en particiones: K-Medias

La meta de este algoritmo es particionar la población inicial en K cluster, cada individuo se agrupa con el cluster más próximo. Para ello se busca minimizar el sumatorio de distancias entre los individuos y el centroide de su cluster.

Algorithm 2: K-Medias

```

Data: poblacion, K // Número de clusters
Result: agrupacion // Clusters para cada individuo de la población
centrosNuevos := init(K); // Inicializar los centros de manera aleatoria
centros := centrosNuevos;
repeat
|   asignar(poblacion);
|   centrosNuevos := calculaCentros(poblacion, asignar);
until centros != centrosNuevos;
return agrupacion;
```

Sin embargo, hay que tener en cuenta que la inicialización de los centros es estocástica, por lo que el algoritmo puede converger en un óptimo local. Por eso es importante repetir el algoritmo varias veces para encontrar el óptimo general. Idea representada en la figura 2.6. Se ejecuta varias veces el algoritmo, para ver las posibles asignaciones. El intento con menor valor de sumatorio de distancias de individuos y su cluster asignado, es la mejor asignación entre los intentos ejecutados.

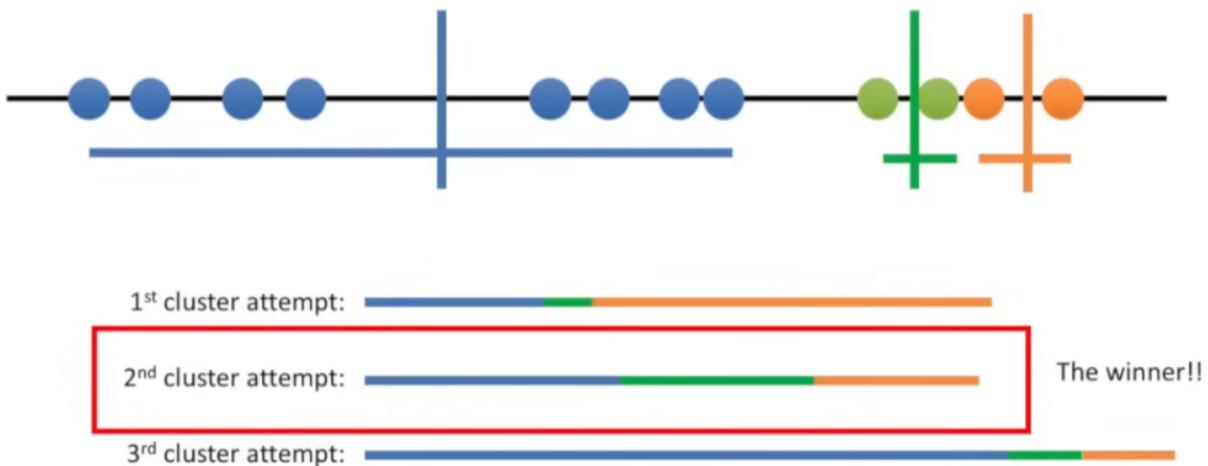


Figura 2.6: KMedias - Busqueda

2.4. Aprendizaje Supervisado

Al contrario que el apartado anterior, este tipo de aprendizaje automático, es entrenado con un dataset categorizado con su salida correcta. El algoritmo aprende de este conjunto, para hacer predicciones sobre unos datos desconocidos.

El objetivo de este algoritmo es aprender la función que mapea las variables de entrada en las categorías correctas de salida. Ajusta los parámetros con técnicas de optimización iterativas para minimizar el error en sus predicciones.

Los ejemplos más comunes son la clasificación, para dividir la población en categorías según unos parámetros. Y regresión, que encuentra las correlaciones entre las variables dependientes e independientes.

2.4.1. K-Vecinos más Cercanos - KNN

Simple pero potente, es muy efectivo para tareas de clasificación y regresión. Se basa en la idea de que los puntos de datos similares tienden a agruparse en el espacio de características. Pertenece al paradigma de aprendizaje perezoso o basado en instancias.

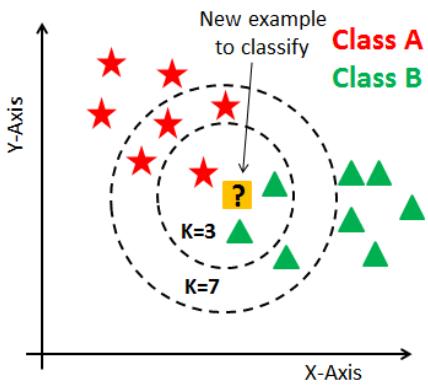
- Perezoso: no calcula ningún modelo y demora todos los cálculos hasta el momento en que se le presenta un ejemplo nuevo.
- Basado en instancias: usa todos los individuos disponibles y ante un ejemplo nuevo recupera los más relevantes para componer la solución.

No hay una forma de determinar el mejor valor para K, hay que probar con varias ejecuciones. Valores pequeños de K crea sonido y valores grandes con pocos datos hará que siempre sea la misma categoría. Un valor diferente de K puede cambiar la categoría de un individuo. En la Figura 2.7a se puede apreciar el proceso de asignación de un cluster a un nuevo individuo. Como se puede ver con los círculos que delimitan los K vecinos más cercanos, si variamos K, la asignación del nuevo individuo puede cambiar.

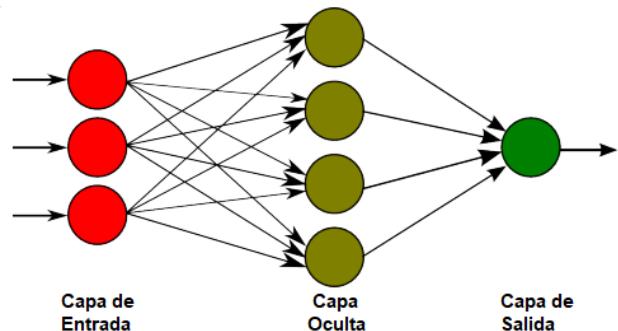
Algorithm 3: KNN

```
Data: poblacion, etiquetas, poblacionPred  
Result: agrupacion // Clusters para cada individuo de la población  
agrupacion :=  $\emptyset$   
for each individuo ind in poblacionPred do  
    // Recorrer toda la poblacion categorizada hasta el momento y clasificar ind  
    // con los K individuos más cercanos.  
    agrupacion.append(cluster);  
return agrupacion;
```

La distancia entre individuos más usada es la Euclídea, pero demora más tiempo al aplicar potencias y raíces cuadradas en su cálculo. La distancia Manhattan es más rápida, pero un poco menos precisa.



(a) KNN



(b) Red Neuronal

Figura 2.7: Aprendizaje Supervisado

2.4.2. Redes Neuronales

Modelo computacional inspirado en el funcionamiento y estructura de las neuronas del cerebro humano. Consiste en capas de nodos interconectados, llamadas neuronas artificiales.

Estructura del modelo (Figura 2.7b):

- Capa de entrada, en la cual, habrá tantas neuronas como variables de entrada tenga el modelo de predicción.
- Capa oculta, representada con una o más capas internas. Cada una con su número de neuronas.
- Capa de salida, como en la entrada. Tendrá un número de neuronas relacionadas con las variables de salida.

Se ha demostrado que tiene un rendimiento notable en muchas tareas, como el reconocimiento de imágenes o procesamiento de lenguaje natural. Aprenden patrones complejos al someterse a un entrenamiento específico con un amplio dataset categorizado.

En el proceso de entrenamiento aprende a realizar una tarea específica ajustando los parámetros internos (pesos en las conexiones), gracias al dataset proporcionado. Normalmente

ajustando con algoritmos de optimización como descenso de gradiente, donde se comparan las predicciones del modelo con las categoría correcta, y actualizando los parámetros del modelo con un método de propagación hacia atrás, backpropagation en inglés. Estos valores se actualizan dependiendo del error cometido y la tasa de aprendizaje proporcionada al modelo.

Algorithm 4: Red Neuronal

Data: entrenamiento, etiquetas, evaluacion // Individuos sin categorizar
repeticiones, capas // Tam. entrada, oculta, salida

Result: pesos // Opcionalmente, devolver los pesos de la red
pesos := init(); // Inicializar los pesos de manera aleatoria

for rep ← 0 **to** *repeticiones* **do**

for each individuo *ind* in *entrenamiento* **do**

// Suma el valor recibido de la capa anterior multiplicada por los pesos de la capa actual con la siguiente. Así se determina la importancia de conexión entre las neuronas.

// Con el valor calculado se aplica a una función de activación y se pasa a la siguiente capa hasta llegar a la salida.

forward();

// El valor predicho calculado en la salida es comparado con la etiqueta, y se calcula el error. Este error se manda para atrás actualizando los pesos. Se suma la multiplicación del valor predicho en cada capa con la tasa de aprendizaje y el error.

backpropagation();

return agrupacion;

2.5. Programación Evolutiva

La programación evolutiva es una técnica de optimización inspirada en la teoría de la evolución biológica. Se basa en el concepto de selección natural y evolución de las poblaciones para encontrar soluciones a problemas complejos.

La población está compuesta por individuos, que pueden ser representados con arrays de números reales, binarios o un árbol, en programación genética. Los individuos tienen un cromosoma, que a su vez tiene uno o varios genes, con uno o más alelos. Esta población es sometida a métodos de selección, mutación y evaluación, para, con el paso de las generaciones maximizar o minimizar un valor fitness.

Esta técnica es muy útil para problemas de optimización donde los métodos tradicionales son bastante lentos. Se han aplicado a varios dominios, como por ejemplo la bioinformática o robótica³.

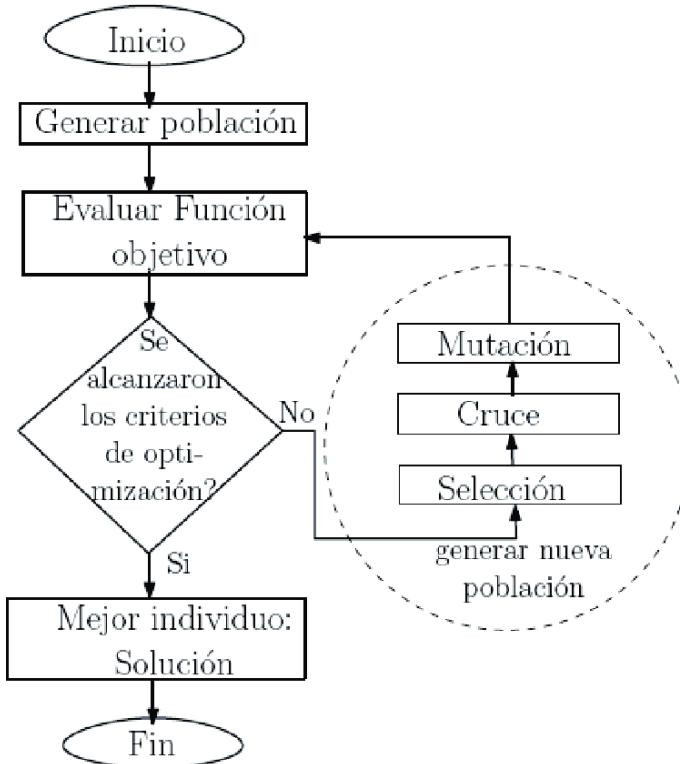


Figura 2.8: Algoritmo Evolutivo

Capítulo 3

Diseño e Implementaciones

En este capítulo se presentan los diseños e implementaciones desarrolladas a lo largo del desarrollo del mismo. Empezando con programas sencillos para introducir MPI, y avanzando por nivel de dificultad en los algoritmos de IA. Los algoritmos de clustering son sencillos de entender y de paralelizar. El aprendizaje por refuerzo y los algoritmos evolutivos aumentan la complejidad en las implementaciones y para finalizar las redes neuronales, que de por si tienen un complejidad más elevada.

3.1. Programas sencillos

Para introducir MPI en el proyecto se implementan y comentan varios programas sencillos. La multiplicación de matrices, que tiene un coste cúbico, al tener que recorrer, para cada elemento de la matriz, una fila y columna entera. Y algoritmos de ordenación, que para simplificar, solo se realizan un estudio de las ordenaciones que más tiempo de ejecución consumen, $O(N^2)$ y MergeSort con coste $O(N\log N)$.

Las matrices son un concepto matemático muy relevante en el mundo de los videojuegos y en el ámbito de la inteligencia artificial. Hay muchas técnicas de IA que llevan la gestión de imágenes, como en el algoritmo DQN cuya red neuronal tiene como entrada varios fotogramas para poder aprender. Estas imágenes, pueden en mayor o menor medida ser implementadas con matrices.

El cálculo de la multiplicación de dos matrices es cúbico $O(N^3)$. Recorre toda la matriz,

y para cada elemento, hace el sumatorio de las multiplicaciones fila, columna.

Es un buen primer ejemplo para iniciar con la arquitectura MPI, debido a la lentitud del cálculo. Para ello hay que plantear cómo dividir el trabajo entre los procesos.

Se puede pensar que es mejor ir enviando los datos conforme se finaliza una operación, pero en esta operación se necesitan las filas de una matriz y columnas de otra, por lo que conviene que cada proceso tenga una matriz entera en su memoria local, para agilizar el proceso y poder enviar más datos al mismo tiempo.

Cada worker se va a encargar de un determinado número de filas, así paralelizando el cálculo. El master se encarga de dividir la matriz entre los procesos, y se puede abordar con dos enfoques distintos.

- Enviar la división completa de filas a cada worker, y esperar a que terminen todos los cálculos, y el master agrupa los mensajes.
- Ir enviando filas de la matriz conforme se va resolviendo. Así logramos tener un flujo constante de mensajes y reducimos la complejidad espacial.

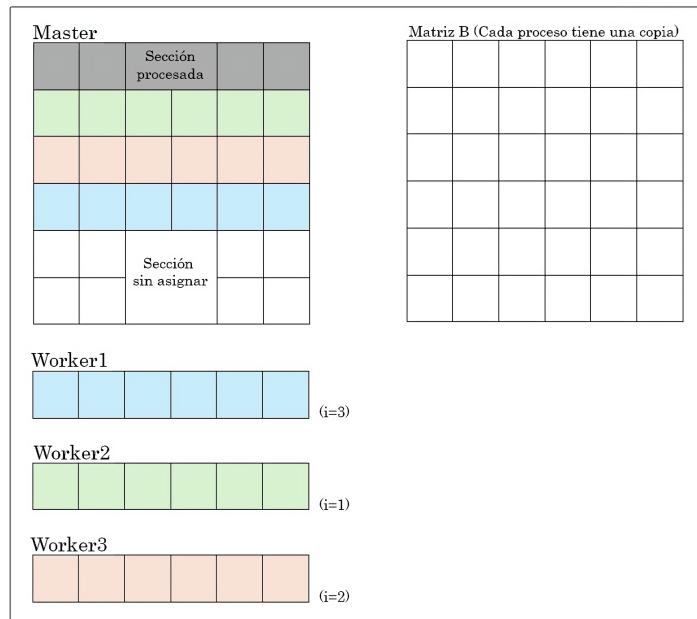


Figura 3.1: MPI - Matriz

En el primer enfoque dejamos al master dormido hasta que recibe los resultados, y al recibir los resultados, se genera un cuello de botella. Además de aumentar la complejidad espacial al tener siempre las filas asignadas. Sin embargo en la segunda idea, el flujo de mensajes hace que el master vaya colocando los resultados recibidos, y se reduce el espacio en memoria, al manejar en cada iteración una fila. Figura 3.1

Los algoritmos de ordenación tienen que iterar varias veces hasta que esté completamente ordenado. Y los métodos pueden variar bastante el tiempo de ejecución.

Para las ordenaciones cuadráticas, los métodos populares son BubbleSort, InsertionSort y SelectionSort, han sido estudiados y optimizados para que aunque tengan un coste cuadrático en el caso peor, tengan un buen rendimiento. Basándome en estos algoritmos he diseñado otro al cual he llamado SequentialSort. Consiste en recorrer todas las posiciones del array, y buscar en qué posición debería de colocarse para que el array esté ordenado. Cada elemento se compara con todos los demás. Se suma uno al contador por cada elemento menor comparado con el actual, y al finalizar una iteración, el contador es la posición del elemento en el array ordenado (si hay repetidos su posición es la primera sin ocupar) Figura 3.5a.

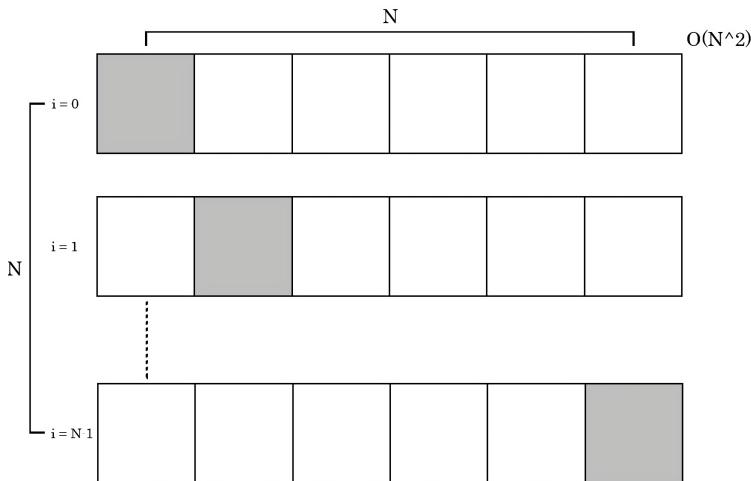


Figura 3.2: MPI - SequentialSort

Este método siempre tendrá coste cuadrático. No es como los anteriores que van redu-

ciendo el espacio conforme aumentan las iteraciones. Pero es fácilmente paralelizable.

Para lograr esta idea, el master envía a todos los workers el array entero, y además asigna a cada uno qué elementos tiene que buscar su posición en el array ordenado.

Los algoritmos de ordenación logarítmicos son muy útiles y eficientes. QuickSort tiene varios problemas como la profundidad de recursión y en el caso peor es cuadrático. Los algoritmos de RadixSort y HeapSort son de los mejores si no se aplican mejoras, y MergeSort es muy popular, tanto que se aplica en Python para el método de ordenación por defecto, TimSort[19]. Este último mezcla una ordenación cuadrática optimizada al máximo, para luego usar las mitades ordenadas con MergeSort. Sin embargo, el algoritmo básico de MergeSort no es tan eficiente.

Aplicando la misma idea que TimSort, se puede mejorar el tiempo de ejecución de MergeSort, aplicando combinaciones de los métodos básicos con complejidad cuadrática y comprobar la eficiencia.

Aplicando MPI se crean varios procesos (para mayor eficacia y simpleza el número de procesos tiene que ser potencia de dos), y se divide el array entre los procesos.

- Primera fase de ordenación: cada proceso ordena su sub-array con el método de ordenación más eficiente. Una vez desarrollado el estudio, el método básico SelectionSort es el que mejores resultados obtiene.
- Segunda fase de reagrupación y ordenación: esta fase se repite hasta solo tener un proceso activo, es decir, el array esté completamente ordenado.
- Proceso de comunicación entre procesos. Cada proceso se conecta con el más cercano (activo). El proceso de mayor id manda su array ordenado y finaliza su ejecución. El que recibe se encarga de ordenar ambas mitades en una sola.

Se usa barrier MPI, para garantizar que terminen todos al mismo tiempo. Esta mejora aplica la idea de sincronización con barrera simétrica mariposa (Figura 3.3). Se espera a los procesos en potencias de dos para sincronizar.

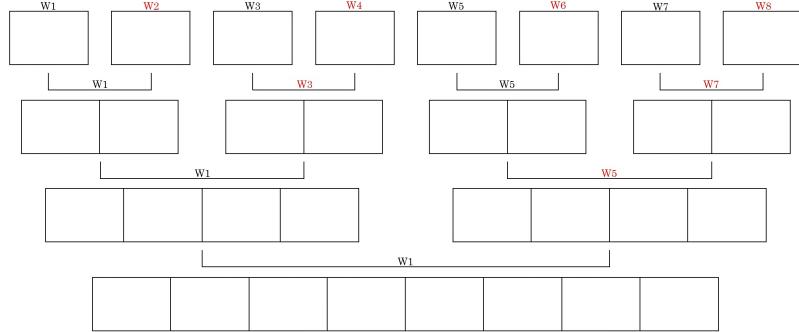


Figura 3.3: MPI - MergeSort

Para aplicar MPI y paralelizar programas hay que tener en cuenta que la comunicación entre procesos tarda un tiempo. Si queremos reducir el tiempo de ejecución de un programa tenemos que asegurarnos que el programa es viable para paralelizar. Si ejecutamos, por ejemplo una búsqueda lineal en un array, a primera vista, reducir el espacio de búsqueda puede ser beneficioso. Dividiendo el espacio de búsqueda entre los workers reduce el tiempo de $O(N)$ a $O(N/\text{numWorkers})$. Pero ¿se puede reducir el tiempo de ejecución al dividir el espacio entre los workers?

Hay que tener en cuenta el tiempo de paso de mensajes. Si no se tuviese en cuenta, se podría garantizar la reducción, pero la comunicación entre procesos tiene un coste, y con un tiempo lineal no se pueden lograr mejoras, más bien aumenta el tiempo de búsqueda.

Por este motivo, hay que tener en cuenta la complejidad temporal de los algoritmos que queremos optimizar, porque no siempre es eficiente aplicar paralelismo.

3.2. Algoritmos de Clustering

Una vez introducido MPI con programas básicos, podemos pasar a ver las implementaciones de algoritmos relacionados con la inteligencia artificial. La clusterización toma una población, y dependiendo del conjunto de datos categoriza los individuos. Puede ser supervisado, si además de la población a categorizar, tenemos un población categorizada previamente. O no-supervisado, si no contamos con esta población etiquetada.

3.2.1. Jerárquico Aglomerativo

Este algoritmo usa una matriz para calcular las agrupaciones. Como es una **matriz simétrica**, podemos reducir la complejidad espacial usando solo el triángulo superior.

La distancia entre clusters es muy importante. Además de calcular agrupaciones distintas, también varía la complejidad temporal. La más eficaz y rápida es la de centroides, para calcular la distancia entre dos cluster solo necesita el cálculo entre dos puntos (los centros de los clusters). Los demás métodos, enlaces simples y completos, comparan las distancias entre todos los individuos de ambos clusters, provocando un coste mayor. Conviene explotar más el paralelismo en estas distancias.

Una vez implementadas las mejoras en el cálculo de multiplicación de matrices, podemos usar estas para mejorar este algoritmo. La primera idea de enviar las filas conforme se realizan los cálculos no se puede aplicar. El algoritmo es más complejo que unos simples sumatorios de multiplicaciones y varía conforme a la población. Dividir el todo espacio entre los workers esta vez si es óptimo.

Como es una matriz simétrica y se representa con el triángulo superior, hay que dividir la carga de trabajo equitativamente. No podemos implementar una mejora sin dividir el espacio de forma óptima entre los workers. Si dividimos las filas de forma secuencial, el primer worker tendrá muchos más elementos que el último (Figura ??).

Dividiendo las filas por pares (parte superior, parte inferior) repartimos de forma óptima las filas para cada worker. Figura 3.5a.

$$\sum_{i=1}^{\text{filas}} (N - i) \gg \sum_{i=\text{filas}(M-1)}^{\text{filas}(M-1)+\text{filas}} (N - i)$$

N individuos de la población, M procesadores. N/M filas para cada worker.

Con 100 individuos de población y 4 workers, cada uno tendrá 25 filas. Por lo que:

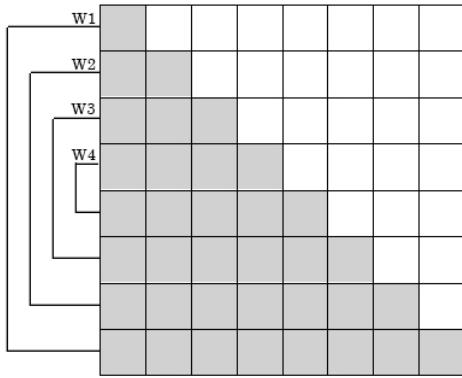
Worker1 tiene las filas de 1-25, con 2175 elementos.

Worker4, las filas de 76-100, con solo 300 elementos.

El Worker1 tiene 7.25 veces más elementos, no reducirá el tiempo de ejecución.

Figura 3.4: Jerarquico Aglomerativo - Cálculo de la distribución optima

Así cada worker tiene el mismo número de elementos que calcular y analizar, inicialmente.



(a) División óptima

El master, que se encarga de:

- Dividir las filas con el método óptimo comentado.
- Establecer la comunicación en el bucle principal, para el desarrollo del algoritmo.

En cada iteración:

- Los workers envían su distancia mínima. El master se queda con la más pequeña, y le pide la fila (c_1) y columna (c_2) al worker con distancia mínima. Así, con un diccionario, optimizar la búsqueda del worker con la fila a eliminar y el que tiene la fila a actualizar.
- Los workers reciben los índices (c_1 y c_2), y los ids eliminan y actualiza.
 - Todos eliminan los elementos de la columna c_2 . El worker con el id elimina, borra la fila c_2 de su memoria.
 - Todos actualizan los elementos de la columna c_1 . El worker con el id actualiza, re-calcula las distancias de la fila c_1 .

(b) Tareas de los proceso

Figura 3.5: MPI - Jerarquico Aglomerativo

Una vez descrito el reparto de espacio entre los procesos, cada uno tiene que ejecutar el algoritmo en paralelo, sincronizándose cada cierto tiempo para actualizar valores.

Refrescando la memoria, este algoritmo en cada iteración agrupa los dos individuos más cercanos. Eliminando una fila y columna de la matriz.

El cálculo de las distancias de la nueva fila usando distancias de centroides es lineal, no se puede reducir el tiempo de ejecución. Pero para los enlaces simples y completos, que tienen un coste cuadrático, se debe intentar reducir el tiempo de cómputo. Para lograrlo se puede dividir el cálculo entre todos los workers.

Hay que encontrar una manera óptima de realizar el cálculo.

- Cuando hay pocos individuos por cluster, es más probable que haya muchas columnas que actualizar, y conviene dividir el cálculo de distancias de todas las celdas.
- Sin embargo, cuando aumentan los individuos, reducen las columnas y esta idea ya no conviene. Cada proceso estaría mucho tiempo realizando los cálculos. Por eso es mejor ir calculando las distancias de forma escalonada usando todos los worker, dividiendo los individuos del cluster para encontrar la distancia mínima (simple) o máxima (completa).

3.2.2. K-Medias

Perteneciente al aprendizaje no supervisado, es una técnica de clustering en la cual tenemos una población inicial de individuos sin clasificar, y un valor K sujeto a una asignación flexible según nuestros criterios. Al contrario al algoritmo anterior no se usa una matriz, y solo se usa distancia por centroides. Una mejora MPI se puede implementar de las siguientes formas:

1. Dividir la población entre los workers. Figura 3.6
2. Ir repartiendo partes de la población para que se vayan procesando y enviando.

La primera idea es la más simple y la que mejor suena en un primer momento. El master se encarga de generar los centroides de manera aleatoria, eligiendo K individuos al azar, sin repeticiones. Mediante broadcast los workers reciben estos centros, y con conexiones punto-a-punto se recibe la población dividida sin intersecciones. Cada worker se encarga de una subpoblación.

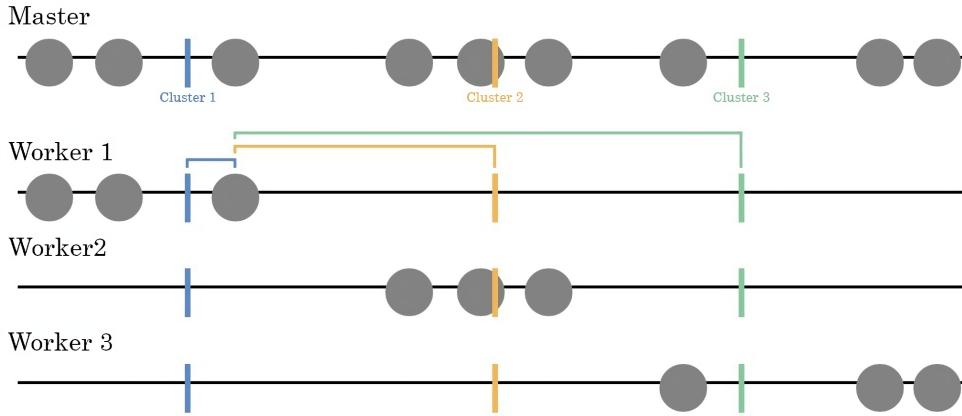


Figura 3.6: KMedias - División de poblaciones

El siguiente proceso se repite hasta que el master envíe un mensaje de finalización, es decir, no cambien los centros:

- Los workers calculan la asignación de sus individuos. Además calculan la suma de distancias de los individuos a sus centros y el número de individuos asociado a cada cluster. Estos valores los envían al master.
- El master recibe estos valores y calcula los nuevos centroides. Manda un mensaje a todos los workers.
 - CentroidesNuevos, si los centros cambian. Se actualizan los centros.
 - Finalización, en caso contrario.

Después de ver e implementar la primera opción, no es una buena idea tener un flujo constante de mensajes, pidiendo nuevos datos. La población no cambia, solo cambia la asignación de los individuos y los centros de los clusters. Solo es necesario un mensaje para recibir la población.

Búsqueda del óptimo general, y mejor valor para K.

Si queremos realizar este proceso, vamos a necesitar ejecutar la búsqueda del óptimo general para ciertos valores de K. Por lo que se van a necesitar muchas ejecuciones del algoritmo.

1. Se puede aplicar la técnica mejorada MPI, con dos bucles. El más externo varía los valores para K, y el interno ejecuta varias veces el algoritmo para conseguir la mejor asignación.

$$O((T \times (N \times K)) \times \text{Rep} \times K_{\max}) \approx O\left(\frac{(T \times (N \times K)) \times \text{Rep} \times K_{\max}}{M}\right)$$

T = número de iteraciones en el algoritmo de K-Medias

N = número de individuos en la población

M = número de proceso

Rep = repeticiones para buscar el óptimo general

$KMax$ = valor máximo de K en la búsqueda.

Figura 3.7: KMedias - Comparación temporal de las mejoras

2. Cada proceso ejecuta la búsqueda del óptimo general para valores de K distintos. El master se encarga de recibir las mejores asignaciones y calcular el mejor valor para K, aplicando coeficientes de optimalidad. Esta idea tiene un coste mucho mayor, puesto que cada proceso tendrá la población entera.

Ambas ideas tienen el mismo coste temporal, sin contar los mensajes en MPI. Pero la segunda opción tiene un coste espacial mayor, por lo que reduce la eficiencia al crear muchos procesos.

3.2.3. K-Vecinos más cercanos (KNN)

Tenemos un valor K asignado de manera arbitraria como en el algoritmo de K-Medias. Esta técnica de clustering pertenece al aprendizaje supervisado, tenemos una población de individuos categorizados con las etiquetas de asignación de cluster. Y una población a predecir.

Aplicando una cola de prioridad de máximos para el cálculo de los K vecinos más cercanos, reducimos la complejidad del algoritmo. Al recorrer la población categorizada, se compara con la cima de la cola. Si la distancia a comparar es menor que la cima, se elimina la cima y se introduce la nueva distancia. Los valores de la cola se mueven con la restricción de prioridad. Y al finalizar la búsqueda en la población se cuentan los elementos de la cola, para saber qué cluster se repite más.

Es importante actualizar la población conforme se van prediciendo los valores, para tener más puntos de referencia. Si no actualizamos la población, la agrupación de los individuos puede variar mucho. Aunque es menos precisa a la hora de predecir, el algoritmo es más veloz, al no aumentar la población categorizada. Al tener una población extra (la categorizada) hay dos formas de implementación posible:

1. Dividir la población categorizada entre los workers. (Figura 3.8a)
2. Dividir la población a predecir entre los workers. (Figura 3.8b)

Si dividimos la población categorizada (primera idea), los workers trabajan menos en cada individuo. Comparan el individuo a predecir con su subpoblación, y el master trabaja más, al recibir los K vecinos de cada worker. Mientras que el master comprueba las distancias recibidas, los workers trabajan con el siguiente valor a predecir. Para la actualización de individuos, el master reparte de forma equitativa. En cada iteración envía el individuo categorizado a uno distinto.

Con la división de la población a predecir (segunda idea), cada worker trabaja lo mismo, pero predice menos individuos. El máster no trabaja tanto como en la mejora anterior, solo recibe la categorización de los workers. Actualizar la población se puede realizar de varias formas, ya que aquí todos los workers comparten la población categorizada, no como en el anterior. Se puede hacer enviando M nuevos individuos a los M workers ejecutados, en cada iteración, es decir, al recibir un individuo predicho de cada worker. O cada X iteraciones, enviar los nuevos individuos categorizados.

El coste espacial depende de los tamaños de las poblaciones.

- La primera idea, al dividir la población inicial, es más eficiente cuando esta población inicial es mayor a la de predicción.
- En su contraparte, la segunda implementación, al dividir la población a predecir, tiene un mejor rendimiento con poblaciones de predicción mayores.

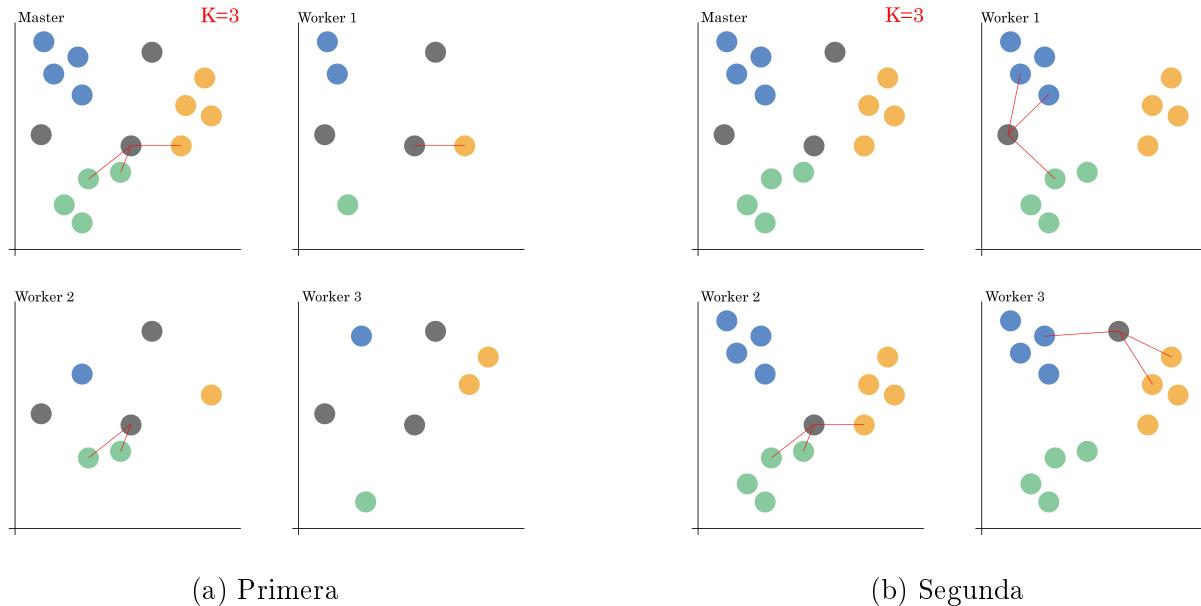


Figura 3.8: MPI - KNN implemetaciones

El proceso Master (el de la esquina superior izquierda) tiene el dataset completo. En la primera implementación (gráfico de la izquierda) divide la población categorizada entre los procesos. Y la segunda implementación (gráfico de la derecha) tiene una copia entera de la población categorizada en todos los procesos.

Para realizar una búsqueda del mejor valor para K, al contrario que en K-Medias, no hace falta repetir varias veces el mismo algoritmo, el proceso es determinista. Es decir, con la misma población, siempre sale la misma predicción. A no ser que se cambie el orden de predicción y se actualice la población a predecir. No es necesario ejecutar varias veces el mismo algoritmo, pero sí puede ser útil variar el número de vecinos (K). Las mejoras de esta búsqueda son las mismas que en el algoritmo anterior. Aplicar la mejora MPI e iterar cambiando los valores de K, o ejecutar de manera secuencial en varios procesos.

3.3. Aprendizaje por refuerzo

El algoritmo de Q-Learning actualiza iterativamente las estimaciones de calidad de las acciones permitidas en el entorno de desarrollo. Estos valores se almacenan en la Q-Table, representado como una matriz en la que cada fila es un estado, y las columnas son las acciones disponibles. Nos centramos en la técnica de **minimizar las acciones** de un agente.

El agente se encuentra en un laberinto, el tamaño y la semilla es variable por parámetros de inicialización. El agente tiene que recorrerlo desde un punto inicial hasta una celda destino, en el menor número de acciones posibles. Las acciones disponibles para el agente son solo de movimiento: norte, oeste, este y sur. No puede atravesar ni situarse en un muro del laberinto. Hay que fijar unas recompensas con las acciones tomadas.

- Si se choca con un muro castigamos al agente con valores altos para que no haga esas acciones.
- Al moverse, el agente recibe un castigo pequeño para que aprenda a minimizar las operaciones.
- Al llegar a la meta le damos una recompensa alta.

Con estas recompensas aprende a llegar a la meta minimizando las acciones ejecutadas. El código que genera los laberintos ha sido implementado por @ChlouisPy en github. [12]

Se puede optimizar el algoritmo haciendo algún **preprocesado**. Modificar la Q-Table y convertirla en un array bidimensional, en el cual no se almacenen las acciones que no deseamos que realice el agente, como puede ser chocarse con un muro. Asimismo, no añadir estados inaccesibles. Esto reduce el número de estados, pero añade un nuevo array bidimensional de acciones para cada estado. Este preprocesado tiene complejidad cuadrática $O(4^*N^2) \equiv O(N^2)$. Recorre toda la matriz, comprueba para cada celda si no es un muro, y en caso afirmativo comprobar las 4 las acciones permitidas y almacenar las disponibles en los

arrays bidimensionales de valores-Q y acciones. Con tamaños de laberintos pequeños no hace falta paralelizar el preprocesado, porque no se consigue reducir el tiempo significativamente.

Los hiper parámetros (α , γ , ϵ) son muy importantes para el desarrollo del agente en el entorno. Una mala configuración de estos hace que sobre aprenda o no aprenda correctamente, generando bucles infinitos. Por este motivo es importante comprobar las diferentes combinaciones de hiper parámetros, y ver cuales funcionan correctamente en el entorno.

La búsqueda en laberintos grandes es muy lenta. Hay que comprobar muchas combinaciones entre los hiper parámetros y los episodios del entrenamiento. Por eso es más útil desarrollar el algoritmo Deep Q-Learning que no tiene problemas con los estados, al usar una red neuronal. Pero si queremos usar el algoritmo básico de Q-Learning, hay que realizar una búsqueda exhaustiva en el entorno ejecutando muchas combinaciones de hiper parámetros.

El algoritmo de Deep Q-Learning no lo implemento ya que las mejoras son parecidas a las que se implementan en el apartado 3.4

Búsqueda de hiper parámetros ideales:

Con MPI se puede procesar ejecutando de manera secuencial en diferentes procesos. Idea mencionada en otros algoritmos.

El master envía configuraciones diferentes a los workers. Si aplicamos una precisión de 0.1, y reduciendo el número de mensajes, cada worker ejecuta 9 veces el algoritmo, aumentando un 10 % en cada iteración el hiper parámetro ϵ (el que controla la toma de decisiones).

El master envía configuraciones diferentes a los workers. Si aplicamos una precisión de 0.1, y reduciendo el número de mensajes, cada worker ejecuta 9 veces el algoritmo, aumentando un 10 % en cada iteración el hiper parámetro ϵ (el que controla la toma de decisiones). Al terminar una ejecución el worker escribe en un fichero de texto:

- La id del proceso.
- Los hiper parámetros ejecutados.v - El tiempo de ejecución
- Como termina:
 - Bucle en entrenamiento.
 - Bucle en evaluación.
 - Movimientos. Si termina correctamente.

Figura 3.9: RL - Búsqueda de hiperparámetros

Los bucles se detectan de diferentes formas. En el entrenamiento, si un episodio tarda más de x segundos, es porque está en un bucle y estos hiper parámetros no funcionan. En la evaluación se comprueba llevando una cuenta de los últimos 4 estados visitados.

Estados[0]==Estados[2] and Estados[1]==Estados[3] and Estados[0]!=Estados[1]:

Las posibles mejoras de este algoritmo son más complejas, se puede:

1. Dividir el entorno entre los procesos.

2. Ejecutar el algoritmo en los workers y juntar las experiencias.

Al dividir el laberinto entre los procesos, cada proceso controla una zona, y se genera un flujo constante de episodios. Cuando un agente sale del dominio de un proceso, este le manda un mensaje al proceso que controla esa parte del laberinto, con la posición en la que entra. Figura 3.10. El master se encarga de iniciar a los agentes y cuando sale su dominio genera otro agente. Para garantizar que funcione el master no puede recibir agentes de los workers, porque si no no se podría garantizar el flujo de nuevos episodios. Como cada proceso tiene su propio dominio, la Q-Table se divide entre estos. Aplicando la mejora con el preprocesado, es la misma idea pero con los dos arrays bidimensionales (acciones y Q-valores).

Ejecutar en varios procesos el algoritmo, funciona. El master recolecta las experiencias de los workers, haciendo la media de los valor-Q obtenidos de los procesos, así calculando la

mejores acciones para cada estado. Es mucho mejor inicializar el algoritmo desde distintos puntos, para así explorar todo el laberinto y encontrar el mejor camino en menos tiempo. Al menos un worker inicia desde el punto de inicio que se desea evaluar, porque por el contrario no se podría garantizar que visiten la zona del inicio.

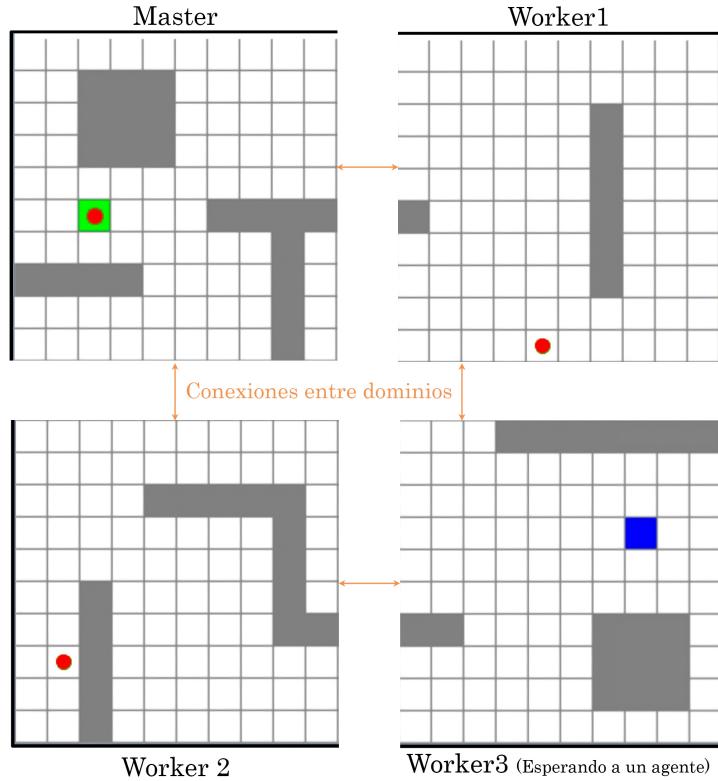


Figura 3.10: RL - División del entorno

Hay que tener en cuenta los límites de cada proceso, para cuando el agente salga fuera de estos, envíe un mensaje con la posición en la que entra al proceso correcto. Es un flujo constante de agentes nuevos, por eso es mejor que una vez salga del primer proceso (el que genera los agentes), no pueda volver a entrar, así simplificando la lógica de programación.

3.4. Algoritmos Evolutivos

Los algoritmos evolutivos son sencillos de paralelizar, debido a que son procesos que se repiten muchas veces, y se ejecutan en muchos individuos.

1. **Inicialización.** Dados los parámetros iniciales se crea la población, con los individuos deseados. Hay diferentes tipos, con sus respectivas características.

- Binarios. Estos individuos son fáciles de inicializar, pero ralentizan la comunicación entre procesos, debido a tener que enviar muchos bits.
- Reales. Parecidos a los binarios pero estos son más portables.
- Árboles. Más lentos para inicializar y difíciles de tratar.

2. **Evaluación.** Este es el método que más tiempo de ejecución puede llegar a consumir. Varía dependiendo del tipo de individuo del problema.

3. **Selección.** Se seleccionan a los individuos. El tiempo depende de los métodos de selección aplicados, pero suelen ser lineales.

4. **Cruce.** Con una cierta probabilidad, se cruzan 2 individuos del conjunto seleccionado. Tienen un mayor coste que selección.

5. **Mutación.** Igual que el cruce tiene una probabilidad para mutar. Parecido a cruce, pero un poco más rápido.

Para cada tipo de individuos se desarrollan unos problemas.

1. Binario, con un intervalo dado y una precisión, queremos calcular el valor máximo o mínimo para ciertas funciones. Los valores fitness se calculan con la representación real del cromosoma, que varía dependiendo de la precisión que se le asigna al ejecutar el algoritmo. Por lo que hay que convertir de binario a real. Este problema se ejecuta bastante rápido. Reducir su tiempo de ejecución es desafiante.

2. Real. Aeropuertos con un número variable de vuelos y pistas. Queremos calcular el sumatorio de tiempos mínimos de retraso para que los aviones aterricen en el aeropuerto. Cada avión tiene asignada la hora de aterrizaje para cada pista, y hay un tiempo mínimo de separación entre vuelos que aterrizan en cada pista. Se puede resolver con vuelta atrás pero tiene un coste exponencial (inviable). El valor fitness tiene coste $O(NumAviones * NumPistas) \equiv O(N^2)$. Se calcula:

```
fitness=0
for avion in aviones:
    for pista in range(pistas): # Calculamos TLA para cada pista
        TLA = maximo(TLA(vuelo_anterior) + SEP[vuelo_anterior][vuelo_actual], TEL)
        # Se asigna el vuelo actual a la pista con minimo TLA calculado
        fitness+=(menor_TLA-menor_TEL)^2
    # menor_TEL: menor TEL de ese vuelo con todas las pistas
```

El tiempo de ejecución para este problema depende de la función de evaluación, que varía dependiendo del número de aviones y pistas.

3. Árbol, tenemos una matriz de enteros que representa un jardín. 1: césped y 0: césped podado. Queremos maximizar el área podada. Para ello el agente tiene unas acciones que puede ejecutar. El coste temporal de este algoritmo viene dado principalmente por la función de evaluación. Ejecuta la simulación en la matriz hasta cumplir un determinado número de ticks (acciones realizadas), que es proporcional al número de filas y columnas de la matriz. Aunque las funciones de cruce y mutación también tardan en ejecutarse, debido al control de punteros.

Cada una de las siguientes mejoras MPI se pueden configurar para cada tipo de individuo.

1. Dividir la población en subpoblaciones.
2. Modelo de islas.
3. PipeLine. (Figura 3.11)

La primera implementación, de dividir la población entre procesos, se puede parallelizar fácilmente. El master recibe de los Workers las subpoblaciones inicializadas y evaluadas, y comienza el bucle principal del algoritmo, en el cual:

- El master se encarga de hacer la selección, y enviarla dividida a los workers. Mientras los workers trabajan, el master almacena el progreso de los mejores individuos de cada generación.
- Los workers reciben la selección, la cruzan, mutan y evalúan. Al finalizar estos procesos mandan la subpoblación al master para empezar la siguiente iteración.

Para el **modelo de islas**, se aplica la idea de mejoras pasadas, en cada proceso se ejecuta el algoritmo. Todos tienen el mismo tamaño, pero distintas poblaciones. Hay varios tipos de comunicaciones en esta mejora. Cada cierto tiempo se reinicia la población con los mejores individuos generales (de todos los procesos).

- Estrella. Solo hay comunicaciones master-worker. El master recibe los mejores individuos.
- En red. No hay proceso master, todos los procesos ejecutan el algoritmo. Todos los procesos están en comunicación constante, mandando los mejores individuos para el reinicio de la población.
- En anillo. Tampoco hay proceso master, los procesos se comunican como una lista enlazada.

Segmentar el algoritmo entre los procesos, provoca un flujo constante de generaciones. Cada proceso se encarga de un método. El master se encarga de generar cuatro poblaciones distintas y las evalúa para mandarlas al worker que se encarga de la selección. El master no genera más poblaciones, y pasa a un estado de recepción mejores individuos. Las poblaciones van evolucionando conforme avanzan por el pipeline. El último worker se encarga de la evaluación, que envía al worker de selección, y al master.

Se puede reducir el tiempo de ejecución si comprobamos que métodos tardan más, añadiendo más procesos para reducir la carga de trabajo.

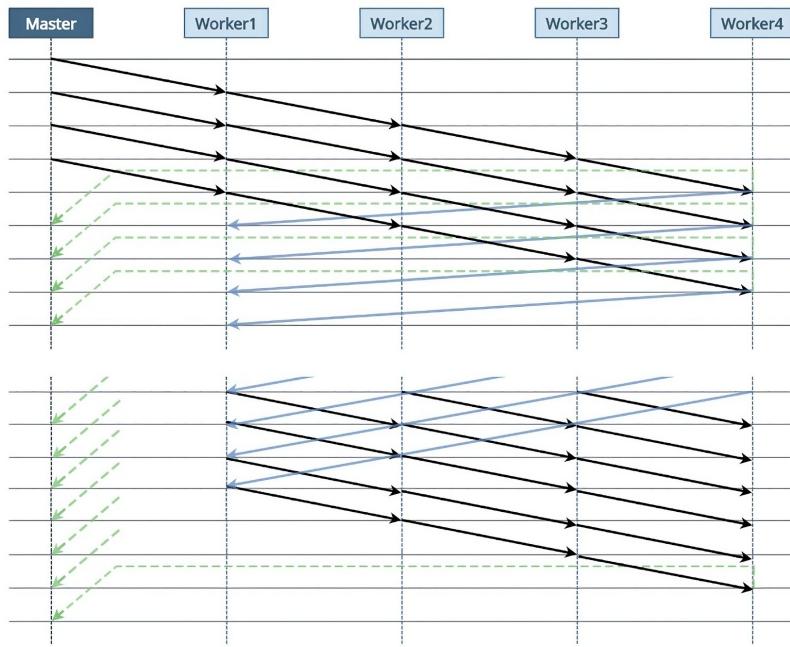


Figura 3.11: MPI - PEV PipeLine

3.5. Redes Neuronales

Esta poderosa herramienta de aprendizaje supervisado, está diseñada para reconocer patrones complejos y realizar diversas tareas. Aprende con un proceso iterativo de entrenamiento, ajustando las conexiones entre neuronas. Este proceso secuencial es complejo de paralelizar. Al finalizar una predicción el modelo se tiene que actualizar propagando hacia atrás.

Nos centramos en la técnica de **predicción**. La capa de entrada está formada por dos neuronas, cada individuo son variables numéricas que representan, la altura y el peso de una persona, y aprende a predecir el Índice de Masa Corporal (IMC). La capa de entrada y salida no varían, pero la capa oculta se puede modificar libremente, aumentando el tiempo en la fase de entrenamiento.

Como en algoritmos pasados, necesitamos encontrar la mejor tasa de aprendizaje, para que aprenda lo mejor posible. Por lo que diseñamos una programa MPI, en el cual se ejecutan en varios procesos el mismo algoritmo con diferentes tasas de aprendizaje y repeticiones para

el entrenamiento. El speedup es proporcional al número de procesos.

Como en el algoritmo anterior, se puede aplicar un pipeline para que haya un flujo de mensajes, pero esta vez en vez de ser unidireccional es bidireccional, al tener que actualizar los pesos de las neuronas al predecir un individuo.

1. PipeLine
2. Dividir el trabajo en procesos

Segmentar el proceso de entrenamiento puede llegar a ser beneficioso. Cada proceso se encarga de una capa de la red neuronal, siendo el master el encargado de enviar individuos de la población categorizada. El último worker controla la capa de salida, con las etiquetas calcula el error y lo propaga hacia atrás. Para el correcto funcionamiento, hay que crear un buen diseño para tener un flujo constante de mensajes. Figura 3.12.

1. El master envía un número proporcional de individuos al número de procesos ejecutándose. Luego entra en un bucle en el cual recibe el error, actualiza, y envía otro individuo. Para finalizar recibe el mismo número de individuos que envió al principio y actualiza.
2. El último worker solo recibe las predicciones y calcula el error.
3. Los workers de la capa oculta tienen un proceso más complejo. Primero reciben un número de individuos proporcional a su id, los procesan y envían. Después entran en un bucle en el cual:
 - Reciben, de la capa siguiente, los errores, actualizan sus pesos y envían a la capa anterior.
 - Reciben, de la capa anterior, los nuevos individuos, procesan y propagan hacia adelante.

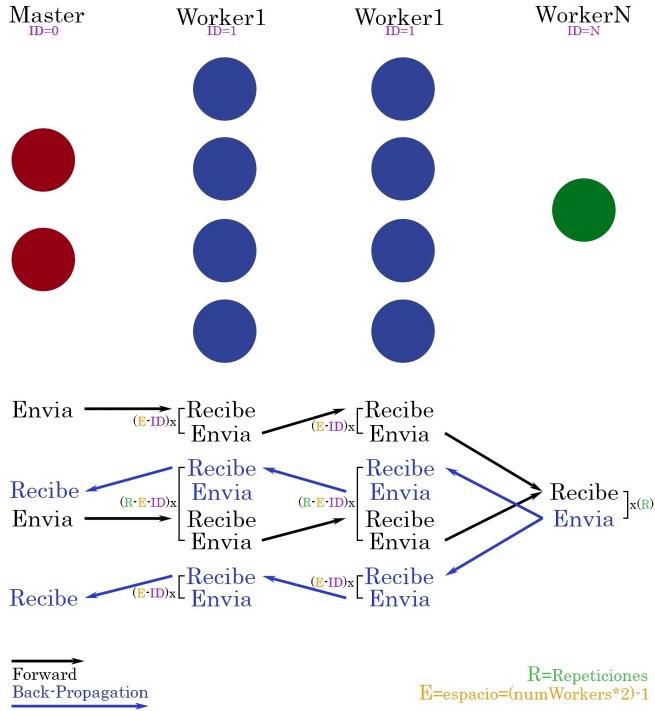


Figura 3.12: MPI - PipeLine Rede Neuronal

Al ser un proceso iterativo, en el cual el modelo va aprendiendo en la fase de entrenamiento, a primera vista, dividir la población entre procesos no parece ser beneficioso para el correcto aprendizaje de la red. Sin embargo, en redes neuronales hay un proceso llamado fine tuning[13] que consiste en entrenar una red neuronal, con unos pesos ya calculados. Basándonos en esta técnica, podemos implementar una mejora en la cual dividimos la población inicial entre procesos, y en paralelo ejecutamos la fase de entrenamiento. Una vez finalizadas el master recibe los pesos de cada worker y hace la media.

Cuanto más grande sea la red neuronal mejor, tanto en rendimiento como en evaluación. En las redes neuronales grandes, un nodo se especializa en unos ciertos parámetros, por lo que no hay demasiadas intersecciones entre los entrenamientos de los procesos. Pero hay que dividir correctamente la población inicial entre los procesos.

1. Con una población distinta, depende del tamaño de la red.
 - Si es una red pequeña, esta técnica no es muy efectiva. La diferencia de pesos

entre procesos puede llegar a ser grande, y al hacer la media dar evaluaciones incorrectas.

- Sin embargo con muchas capas se mejora la evaluación. Cada proceso especializa unas neuronas y al juntarlas en el master no intersecan.
2. Si enviamos una población parecida, depende de la inicialización de los pesos, pero muy seguramente no surta efecto. Es como ejecutar varias veces el algoritmo en diferentes ejecuciones.

Capítulo 4

Estudio empírico

Después de implementar y diseñar los algoritmos y mejoras utilizando MPI, llevamos a cabo un análisis exhaustivo para evaluar los tiempos de ejecución, realizar pruebas, contrastar resultados y extraer conclusiones.

Primero se ejecutan distintas pruebas en un ordenador normal, para luego ejecutar las mejores implementaciones en un cluster con varios ordenadores y muchos procesos. El cluster se representa en la siguiente figura:

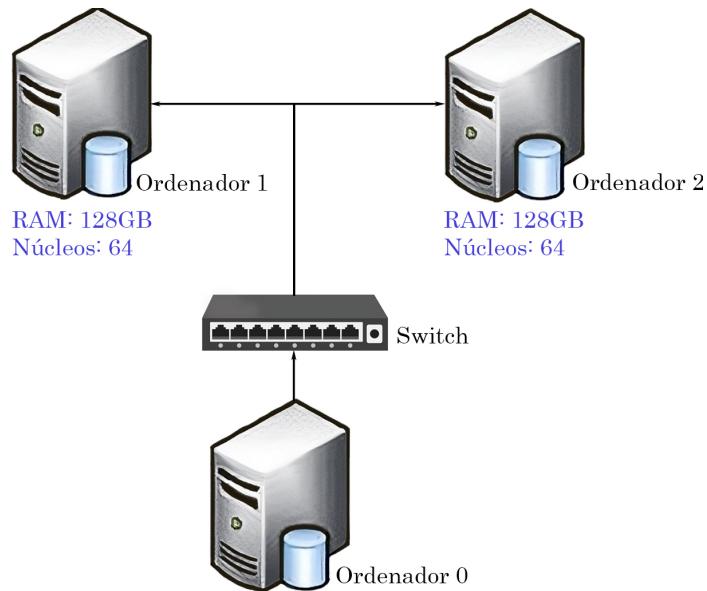


Figura 4.1: Esquema del cluster utilizado

El ordenador 0, realiza la conexión remota con los otros dos ordenadores, situados en la facultad. Lee los datos de entrada y dependiendo del numero de procesos ejecutados, reparte los datos para realizar el cómputo en paralelo (como máximo 128 procesos, que son los núcleos totales entre los dos ordenadores).

Para realizar las pruebas se usa las funciones *open()* y *write()* de Python para almacenar los tiempos de ejecución en ficheros de texto. Los tiempos se miden con las funciones de tiempo de MPI, *MPI.Wtime()*

4.1. Programas sencillo

Primero realizamos el estudio de los programas más básicos, multiplicación de matrices y ordenación de arrays.

4.1.1. Ordenaciones

Pruebas

Arrays de enteros, siempre en el caso peor, es decir, ordenados de forma **decreciente**, por lo que tiene que ejecutar el mayor número de comparaciones para ordenarlo de forma creciente.
Las pruebas se ejecutan sobre el mismo array de enteros, y se van aumentando el tamaño para medir los tiempos de ejecución. En cada prueba se añaden *x* elementos más a ordenar.

Algoritmos cuadráticas sin mejoras

Para las ordenaciones cuadráticas, debido a su coste, las pruebas se incrementan de la siguiente forma:

- [20-1.000) → 20 elementos.
- [1.000-10.000) → 250 elementos.
- [10.000-100.000) → 1.000 elementos.

Algoritmos cuadráticas con mejoras

Algoritmo logarítmico con mejora

Cluster

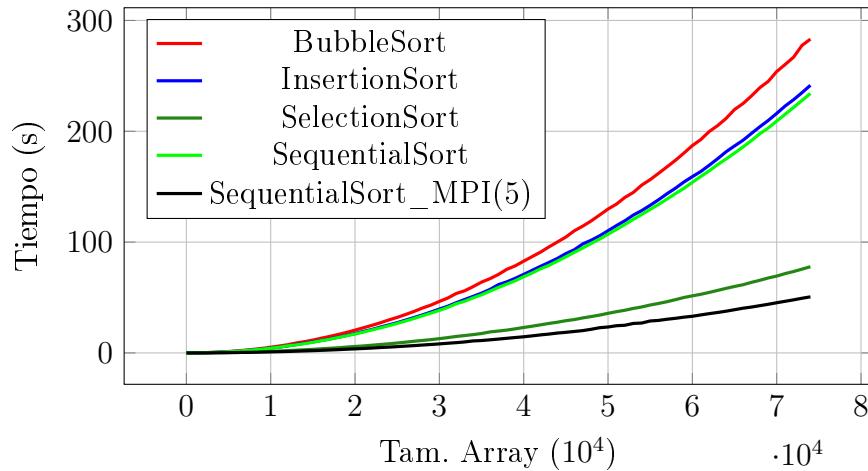


Figura 4.2: Tiempo de ejecución de los algoritmos de ordenación cuadráticos

La implementación aplicando MPI sobre SequentialSort tiene un speed-up proporcional al número de procesos ejecutándose al llegar a un cierto número de elementos a ordenar. Al paralelizar el trabajo de SequentialSort, se pueden obtener mejores resultados que SelectionSort al usar cinco o más procesos workers.

De las ordenaciones básicas con coste cuadrático ($O(N^2)$) comentadas anteriormente, SelectionSort es la que mejores resultados obtiene, y BubbleSort la peor, siendo aproximadamente 3.5 veces más lenta con 70.000 elementos. La ordenación diseñada SequentialSort es incluso más rápida que dos de las más famosas, debido a la simpleza de las operaciones aplicadas en la ordenación. Esta ordenación hace N^2 comparaciones, pero al no modificar elementos del array sigue siendo más rápida que las otras.

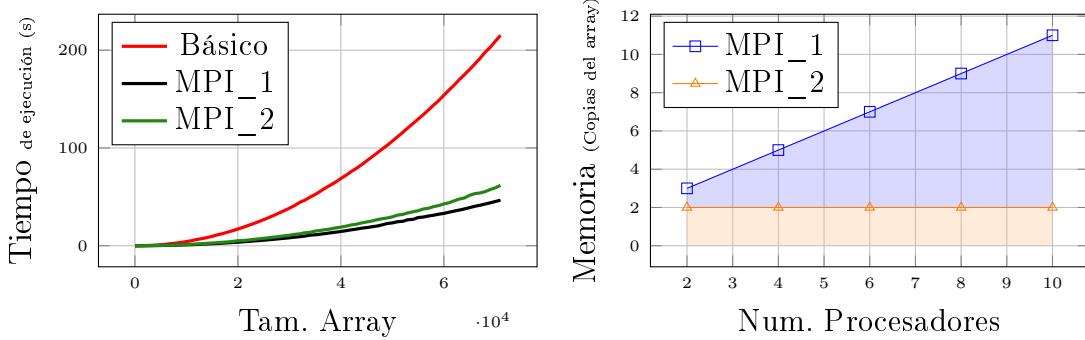


Figura 4.3: Mejoras MPI SequentialSort + Memoria

La primera mejora es un poco más eficiente pero consume mucho más memoria que la segunda. En cuestión de complejidad temporal es un poco mejor repartir el array entero entre los workers, pero la complejidad espacial para esta implementación aumenta de manera lineal en proporción al número de procesadores. Por lo que teniendo en cuenta ambos aspectos es mejor la segunda implementación, dividir el array entre los workers y recibir elementos para comprobar su posición ordenada.

MergeSort tiene un coste temporal de $O(N \log N)$, y al aplicar la mejora comentada se puede apreciar una notoria reducción del tiempo de ejecución, llegando a tener un speedup aproximado de 15.5 usando dieciséis workers, casi el speedup ideal. Es cierto que se podrían aplicar otras técnicas para reducir mucho más el tiempo, pero quería demostrar que en la computación de alto rendimiento se pueden obtener buenos resultados con estrategias no tan efectivas pero bien paralelizadas.

Con dos workers ejecutándose no reduce el tiempo de ejecución, lo duplica. Al aplicar ordenaciones cuadráticas y solo tener dos workers, el cómputo es igual que aplicar una ordenación cuadrática con la mitad del array original.

La memoria está optimizada puesto que el master se encarga de dividir eficientemente el array entero entre todos los workers, teniendo solo dos copias del array, la que tiene el master y la otra dividida entre los procesos que se encargan de ordenarlo. Al terminar un

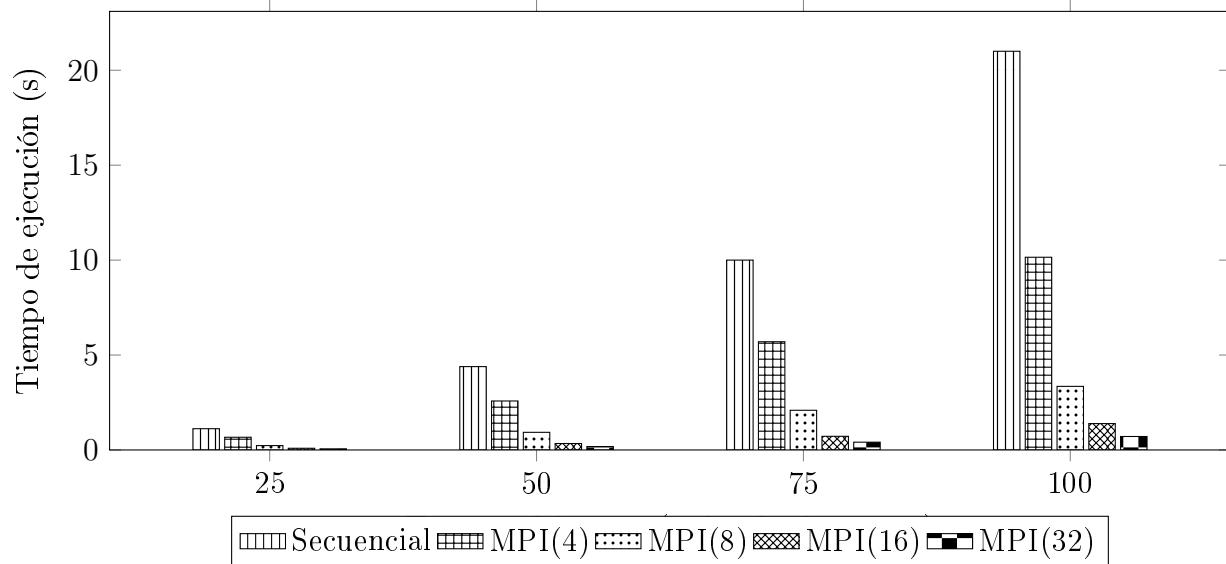


Figura 4.4: Tiempo de ejecución MPI MergeSort

proceso con la sincronización en mariposa, se termina la ejecución del proceso liberando memoria una vez ha enviado al proceso correspondiente su subarray ordenado.

TODO CLUSTER

Matriz

TODO...

Algoritmos de Agrupación

Pruebas

Las poblaciones que se usan en cada algoritmo se han generado previamente de manera aleatoria, delimitando un intervalo $[-10, 10]$ para todas las dimensiones disponibles. Para crear los gráficos se ejecutó muchas veces con diferentes tamaños para guardar los tiempos de ejecución con poblaciones de diferentes tamaños. Se añaden x elementos a las poblaciones.

Para las implementaciones de K-Medias y Jerárquico Aglomerativo se aplican el mismo incremento que en las ordenaciones: El incremento del tamaño viene dado de la siguiente forma:

- $[20-1.000) \rightarrow 20$ elementos.
- $[1.000-10.000) \rightarrow 250$ elementos.
- $[10.000-100.000) \rightarrow 1.000$ elementos.

Para K-Vectores más cercanos (KNN), al ser un algoritmo lineal, se guarda el tiempo de ejecución cada veinte nuevos individuos categorizados.

Jerárquico Aglomerativo:

Cuando la complejidad del cálculo de las distancias entre clusters es constante (aplicando centroide), el tiempo de ejecución no varía al usar un determinado tipo de distancia. No obstante, cuando se aplican distancias entre clusters con coste cuadrático (enlace simple o completo), la distancia euclídea tarda más, porque para realizar el cálculo, usa potencias y raíces cuadradas.

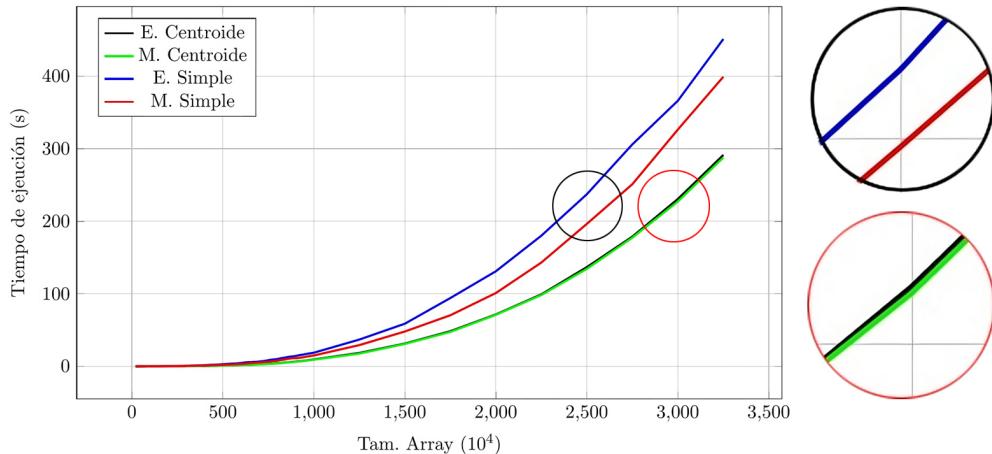


Figura 4.5: Tiempo de ejecución del algoritmo básico Jerarquico Aglomerativo

Al principio no hay tanta diferencia, pero conforme se aumenta la población, los tiempos de ejecución empiezan a distinguirse. El tiempo de ejecución para calcular la distancia entre dos clusters depende del número de individuos de los mismos, y conforme se aumenta

la población aumenta el número de repeticiones del cálculo de las distancias, además de aumentar los individuos en cada cluster lo que también perjudica al tiempo de ejecución.

Distancia entre clusters por centroide. La segunda mejora de dividir entre los workers el cálculo de distancias, no se puede aplicar. Debido a que el cálculo de distancias es constante, no surte mucho efecto dividir la fila que se tiene que recalcular, porque el coste total es lineal. Pero la mejora principal reduce considerablemente el tiempo de ejecución.

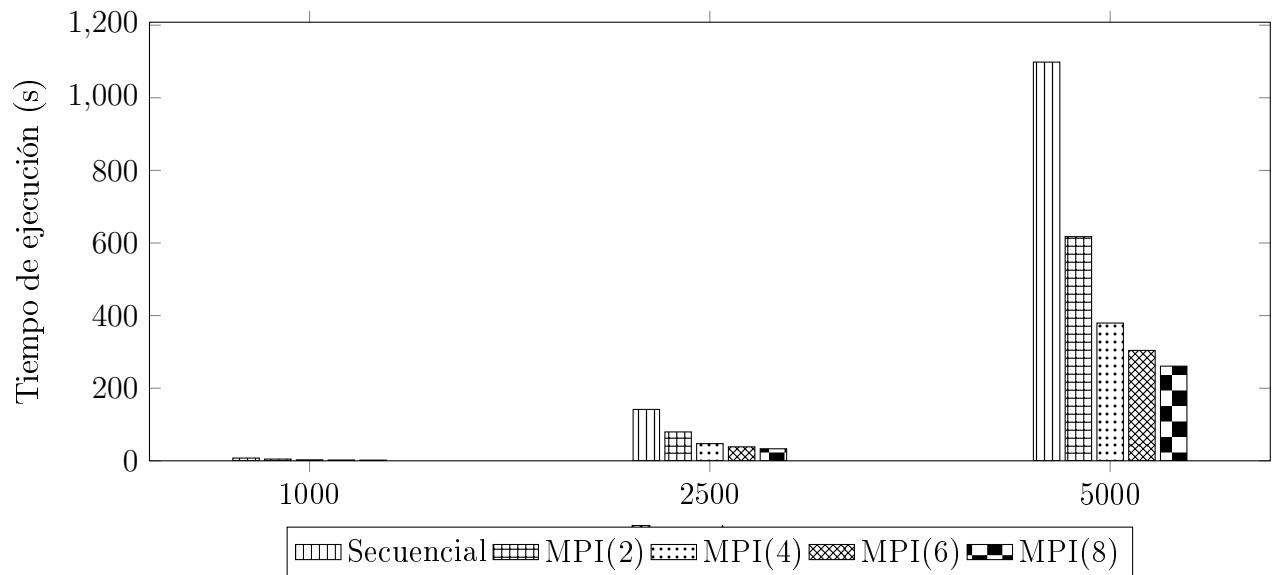


Figura 4.6: MPI Jerarquico Aglomerativo, Dist. por centroide

[TODO PRUEBAS D. SIMPLE/COMPLETO ... TODO CLUSTER](#)

K-MEDIAS

El algoritmo anterior no tiene ninguna variable que modifique el tiempo de ejecución. Sin contar la distancia entre clusters. Esta técnica de agrupación tiene un coste temporal mucho menor que el aglomerativo, $O(N*K*iter)$ siendo N el tamaño de la población, iter las iteraciones hasta que no cambien los centros. ($N \gg K, iter$) K e iter no son valores muy altos por lo que la complejidad no llega a ser cuadrática.

Para las pruebas realizadas se usa un valor K=10, y se usan cuatro procesos workers para la mejora MPI:

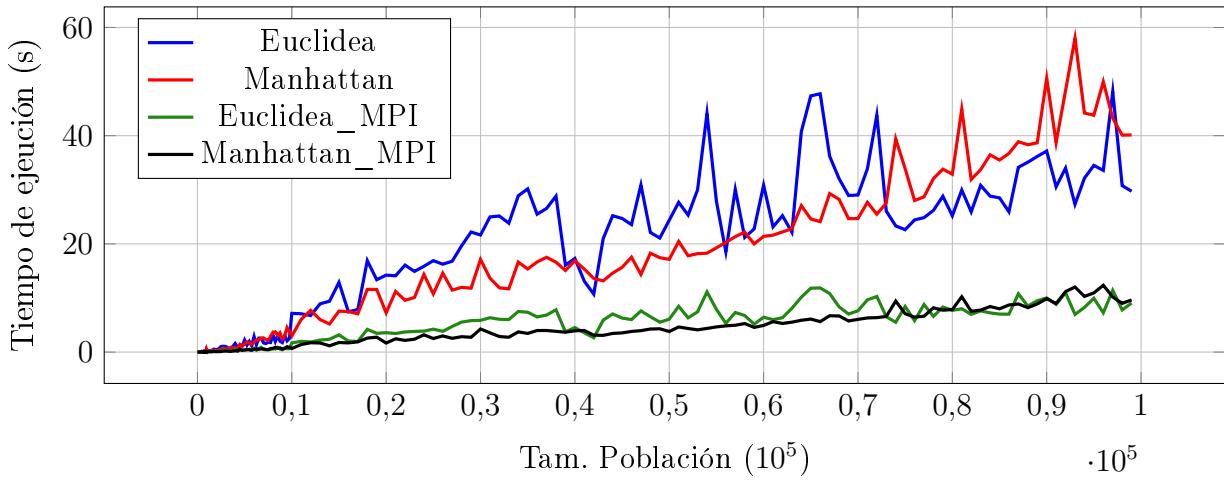


Figura 4.7: Tiempo de ejecución de KMedias

A medida que la población crece, los centros varían, debido a la inclusión de más individuos en el cálculo de las nuevas posiciones de los centros, provocando una variación en el número de iteraciones del algoritmo para que los centros no cambien. El tiempo de ejecución no aumenta en proporción al tamaño, si no que varía dependiendo de la composición de los individuos y por eso hay tantos picos. El aumento de la población no necesariamente implica una ejecución más lenta en comparación con una población menor. Las mejoras MPI, son bastante buenas. Haciendo el mismo número de iteraciones que las implementaciones secuenciales, no hay picos muy pronunciados. Lo más seguro es que al aumentar la población se empiecen a pronunciar.

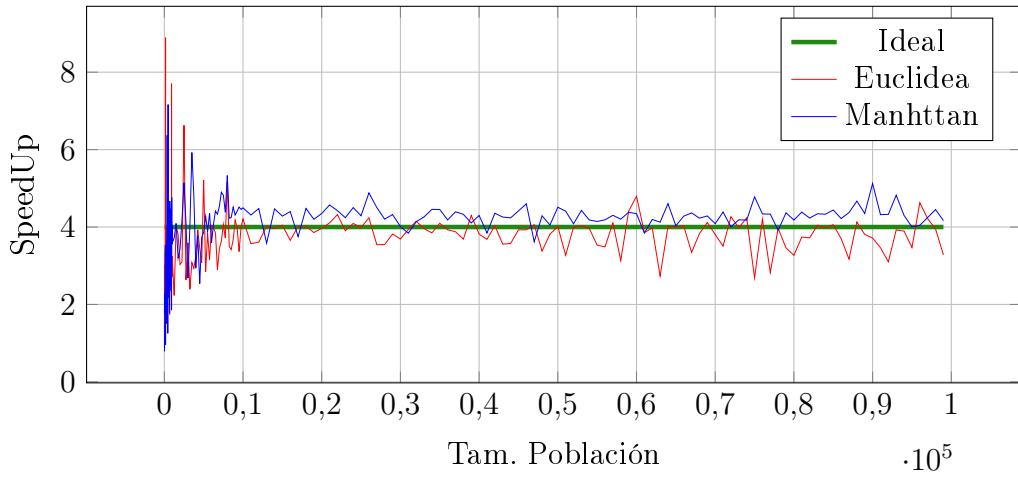


Figura 4.8: SpeedUp - KMedias

El speedup es muy curioso. A partir de diez mil individuos de población, el speedup es aproximadamente el ideal para ambas distancias, contando solo que los workers son los que trabajan y el master solo recibe la asignación y calcula los nuevos centros. Pero lo curioso es que con la población generada aleatoria, y un tamaño relativamente pequeño llega a duplicar el speedup ideal. Lo primero que se puede venir a la mente es que hace menos iteraciones, pero esto no es así, puesto que ejecuta el mismo número de iteraciones en ambas implementaciones.

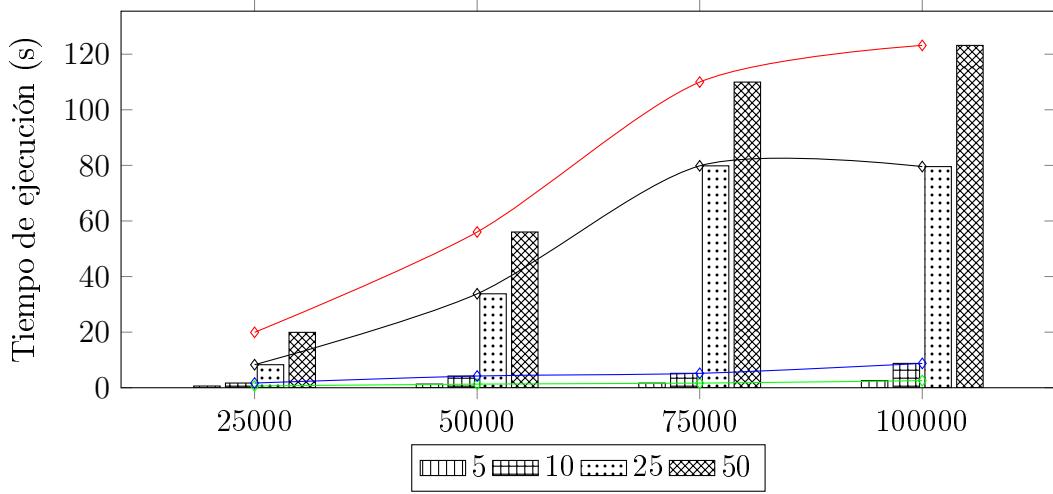


Figura 4.9: KMedias variando K

TODO CLUSTER

KNN

En cada iteración de este algoritmo de aprendizaje supervisado, usa una población categorizada para agrupar un único individuo, no como en los anteriores que tiene que terminar todas las iteraciones para agrupar toda una población.

Para las pruebas realizadas se usa un valor de $K=15$, impar para que no pueda existir empates. Hay dos formas de realizar la agrupación. Si se actualiza la población conforme se categorizan los individuos nuevos, la población final será mucho más precisa que si no se actualiza, pero tardará más tiempo en ejecutarse.

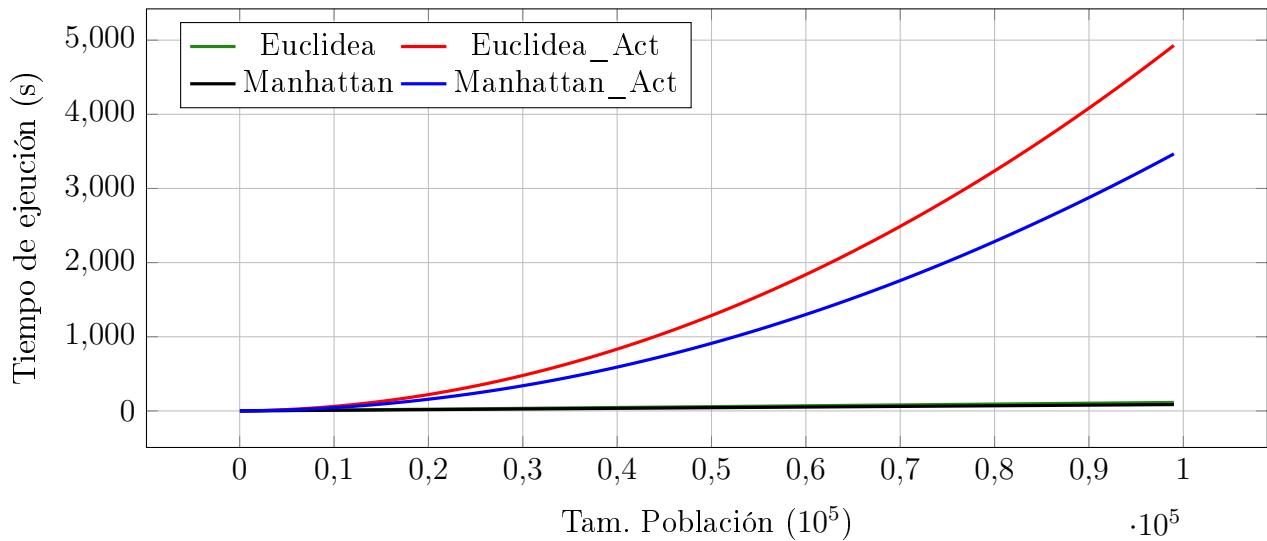


Figura 4.10: Tiempo de ejecución para KNN

Cuando la población se actualiza, aumenta el tiempo de ejecución, y se puede apreciar la diferencia entre los tipos de distancia. A largo plazo la distancia Euclídea tardará mucho más que la Manhattan, por tener una complejidad de cálculo mayor. Cuando la población es constante, los tiempos de ejecución aumentan con el tamaño de la población a predecir de forma lineal, y el uso de distancias no afecta casi al rendimiento.

Las dos implementaciones son aproximadamente iguales, siendo mejor la segunda implementación de la primera mejora. Añadir los individuos categorizados de la iteración anterior cuando cada worker finaliza la búsqueda de los K vecinos más cercanos en la iteración ac-

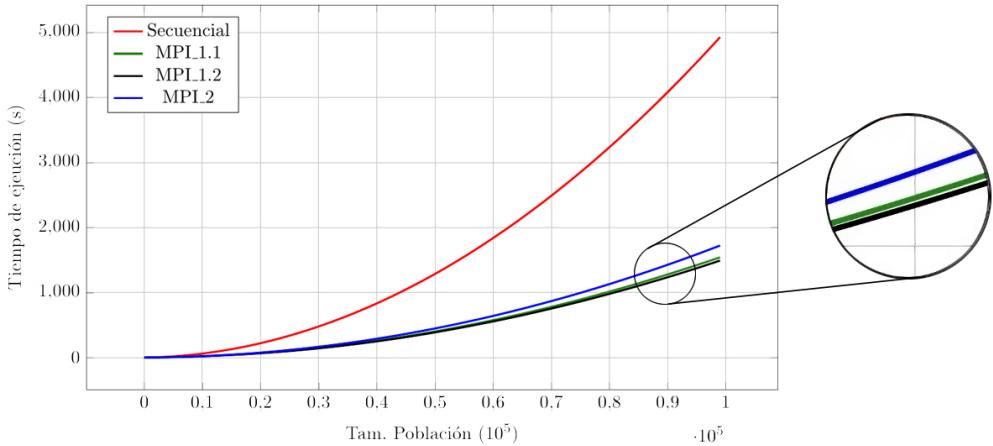


Figura 4.11: Tiempo de ejecución del algoritmo básico Jerarquico Aglomerativo

tual, elimina el tiempo de espera que tenía la primera implementación, reduciendo ligeramente el tiempo. La segunda mejora, además de ser ligeramente peor en cuestión de complejidad temporal, es mucho peor en complejidad espacial. Dividir la población a predecir lleva un mayor consumo de memoria, al tener que tener la población entera en cada proceso. En la primera mejora se reparten equitativamente los individuos nuevos, es decir, en cada iteración el master envía a un único proceso el individuo categorizado.

1. La primera mejora acaba con dos copias de la población inicial y predicha (una en el master y la otra repartida entre los procesos). 2. La segunda mejora acaba con N copias, siendo N el número de procesos ejecutados.

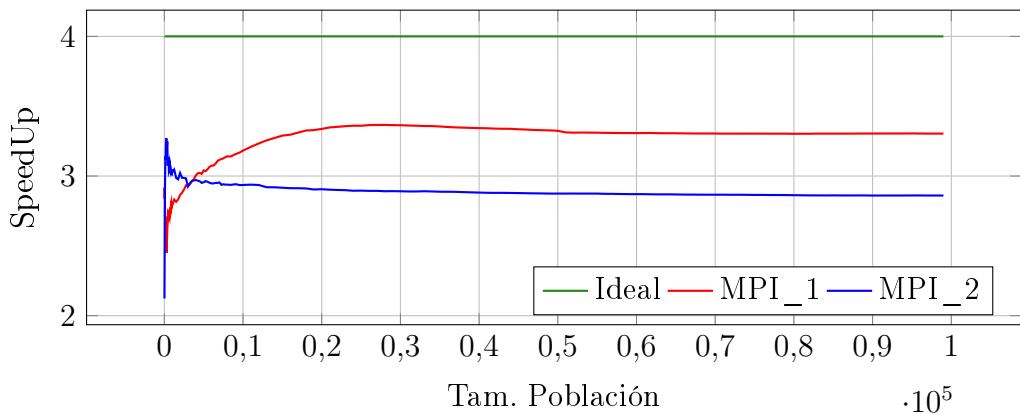


Figura 4.12: SpeedUp - KNN

Al principio es mejor dividir la población a predecir, pero a largo plazo es más efectivo dividir la población categorizada, además de tener menos complejidad espacial.

TODO ? Añadir un estudio del algoritmo sin actualizar? Así a lo mejor se puede comprobar que es mejor la segunda implementación cuando las poblaciones no cambian.

TODO CLUSTER

RL

Una vez implementada este preprocesado y comparándolo con el sin procesar, los resultados son parecidos. Pero con en la búsqueda de los mejores hiper parámetros, da mejores resultados al preprocesar. No hace acciones innecesarias y le permite explorar mejor el entorno y no entrar en bucles.

Estas pruebas se realizan con tres distintos laberintos, ejecutando varias veces para hacer un cálculo más eficaz del tiempo de ejecución para cada laberinto. Las matrices son cuadradas, y se generan de manera aleatoria con *30, 50 y 100 filas*.

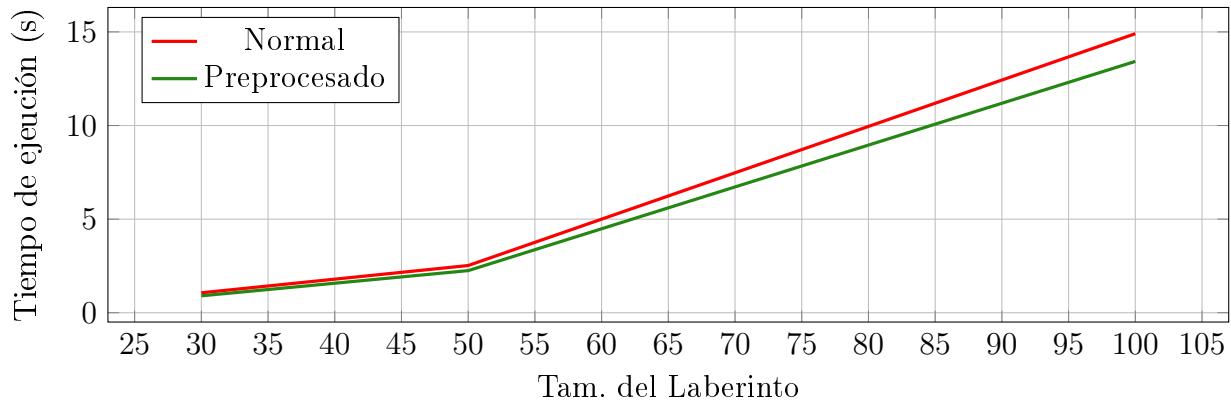


Figura 4.13: Tiempo de ejecución para RL

Para realizar la búsqueda mencionada se aplica la mejora de dividir el trabajo entre varios procesos. El master se encarga de mandar combinaciones a los workers. Dependiendo de la precisión, puede llegar a haber muchas por el poder de la combinatoria, y este proceso ser muy lento. El speedup es proporcional al número de nodos ejecutando combinaciones en paralelo.

Esta búsqueda es muy útil para encontrar configuraciones que funcionen en el entorno. La mejora de matriz dividida no funciona correctamente, se queda en bucles en la mayoría de configuraciones que funcionan para la implementación secuencial del algoritmo. Con unos valores-Q previamente entrenados si funciona, pero la etapa de entrenamiento no funciona.

TODO ? Añadir un estudio del algoritmo sin actualizar? Así a lo mejor se puede comprobar que es mejor la segunda implementación cuando las poblaciones no cambian.

TODO CLUSTER

PEV

Pruebas

Para el algoritmo se ha aplicado un elitismo de 5% conservando los mejores individuos de cada generación. Para el problema de árboles se aplica un método de control de bloating, para intentar reducir la alutra de los individuos. Para mejorar la aptitud de los individuos se aplica un desplazamiento, cuya finalidad es todos los individuos tengan valores positivos. Además de aplicar un escalado lineal, controlando la diversidad de las aptitudes.

El método de evaluación depende del tipo de individuo.

- Si es binario se calcula su valor real y se aplica una función matemática.
- Si es real, el problema del aeropuerto.
- Si es árbol, el problema del cortacésped.

Las pruebas realizadas para todas las gráficas se han ejecutado con las siguientes características:

Tam. Población = 100

Núm. Generaciones = {25,50,100,250,500,1000,2000} (**Eje X**)

Met. Selección: Torneo Determinístico, con un valor k=5.

- Individuo Binario:

Met. Cruce (p=0.6): Básica

Met. Mutación (p=0.05): Básica

P(x)=precision: {P2: 30 bits, P10: 76 bits}

- Individuo Real:

Met. Cruce (p=0.6): PMX

Met. Mutación (p=0.3): Inserción

AER(x)=aeropuerto: {AER1: 10 vuelos, 3 pistas, AER1: 25 vuelos, 5 pistas, AER3: 100 vuelos, 10 pistas}

- Individuo Binario:

Met. Cruce (p=0.6): Intercambio

Met. Mutación (p=0.3): Terminal

M(x)X(y)=matriz: {M8X8: 8 filas, 8 columnas y 100 ticks; M100X100: 100 filas, 100 columnas y 10000 ticks}

Con las tres mejoras implementadas hay que tener en cuenta el tipo de individuo para cada problema, pues dependiendo del tipo, tardará más tiempo en determinadas funciones. Además de tener en cuenta el coste de la comunicación entre procesos.

Tiempo de ejecución (en segundos) de los métodos, para una operación. Es decir, el tiempo que tarda en inicializar, evaluar, seleccionar y mutar un único individuo, o cruzar dos individuos.

En rojo se marcan los métodos más tardíos para cada problema. Con individuos binarios, conviene dar más recursos a las operaciones de cruce y mutación. Y para los individuos reales y árboles la evaluación.

| Datos | Funciones | Init(1) | Evaluación(1) | Selección(1) | Cruce(2) | Mutación(1) |
|----------------------------|--------------|----------|---------------|--------------|----------|-------------|
| Precision: 2 | Binario | 2.56e-05 | 4.4e-06 | 8.56e-06 | 1.36e-05 | 1.53e-05 |
| Precision: 10 | Binario | 3.33e-05 | 5.44e-06 | 8.9e-06 | 1.71e-05 | 1.88e-05 |
| aviones: 12 pistas: 3 | Aeropuerto 1 | 7.04e-06 | 2.55e-05 | 4.12e-06 | 1.48e-05 | 2.764e-06 |
| aviones: 25 pistas: 5 | Aeropuerto 2 | 1.36e-05 | 6.55e-05 | 4.62e-06 | 2.4e-05 | 3.43e-06 |
| aviones: 100 pistas: 10 | Aeropuerto 3 | 3.97e-05 | 4.3e-04 | 8.05e-06 | 4.18e-05 | 1.04e-05 |
| M10x10 ticks: 150 | Árbol | 6.12e-05 | 6.47e-05 | 7.92e-05 | 2.33e-05 | 3.47e-07 |
| M25x25 ticks: 400 | Árbol | 6.16e-05 | 1.65e-04 | 7.88e-05 | 2.32e-05 | 3.7e-07 |
| M100x100 ticks: 800 | Árbol | 6.41e-05 | 3.66e-04 | 8.07e-05 | 2.09e-05 | 3.23e-07 |

Cuadro 4.1: PEV - Tiempos de cada método

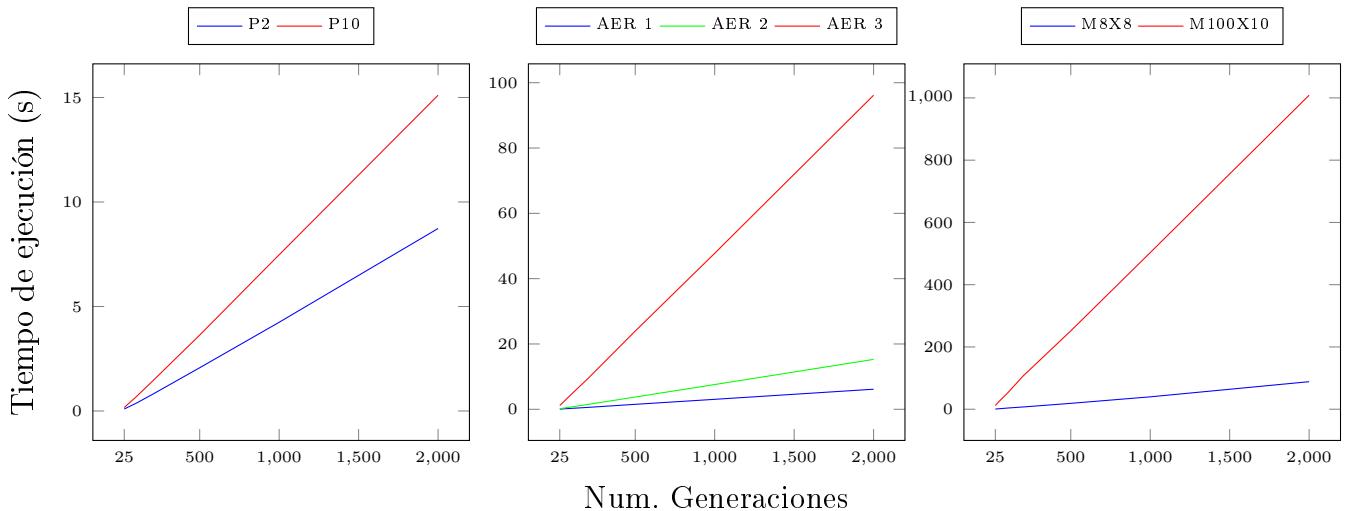


Figura 4.14: PEV Secuencial

Como solo varía el número de generaciones, las gráficas son lineales. Si se modifica de la misma forma el tamaño de la población, serían exponenciales, y tardarían mucho tiempo para ejecutarse.

1. El problema que aplica individuos binarios es bastante rápido, es el que menos tiempo de ejecución tiene entre los tres problemas implementados. La complejidad de la función de evaluación es lineal $O(M)$ siendo M el tamaño del individuo. Recorre todos los bits para

convertirlo a un número real y luego ejecuta una función matemática.

2. Para los individuos reales aumenta en relación al tamaño del problema. La complejidad de la función de evaluación es cuadrática $O(N*M)$, siendo N el número de aviones y M las pistas. Para cada individuo recorre las pistas disponibles asignando la que menor tiempo de retraso genere al vuelo.

3. El problema de los árboles depende del número de ticks. La complejidad de la función de evaluación es lineal $O(\text{Ticks})$.

Mejora 2, Modelo de islas

El modelo de islas, depende de la configuración elegida. Si es el básico, se divide la población entre los workers por lo que se consigue un speedup proporcional a los procesos ejecutados. Para garantizar que funcione igual o mejor que la implementación secuencial, hay que tener una comunicación para garantizar la supervivencia de los más aptos en la población general y de vez en cuando reiniciar las poblaciones de cada proceso con los mejores resultados obtenidos. El maestro se encarga de agrupar los mejores y enviarlos a la hora del reinicio.

Si usamos la configuración en estrella o anillo, podemos ir mezclando poblaciones de y tener más diversidad, y es más probable calcular mejores resultados .

Mejoras: aplicando 4 islas.

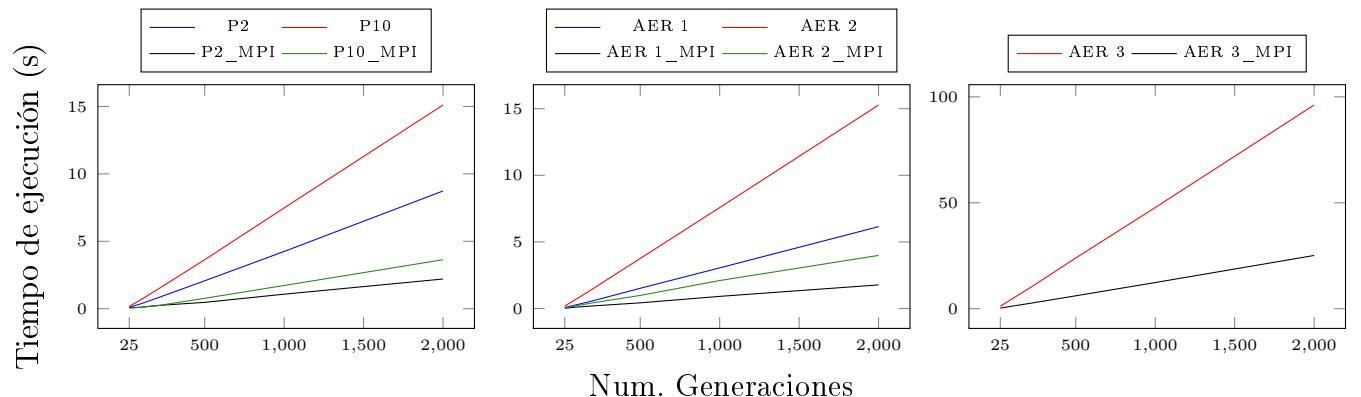


Figura 4.15: MPI - Modelo de Islas

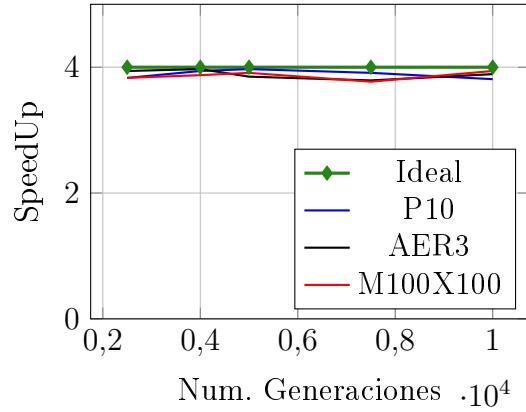


Figura 4.16: SpeedUp - Modelo en Islas

El problema de los árboles calcularía resultados parecidos a estos dos gráficos.

Los resultados son buenos, teniendo aproximadamente un speedup proporcional al número de procesos ejecutándose.

Mejora 1, Dividir con el master

Con una única población, el master se encarga de dividir el trabajo entre los workers, reduciendo el tiempo de ejecución.

Para estas pruebas se ejecutan cuatro procesos workers, cinco contando el master:

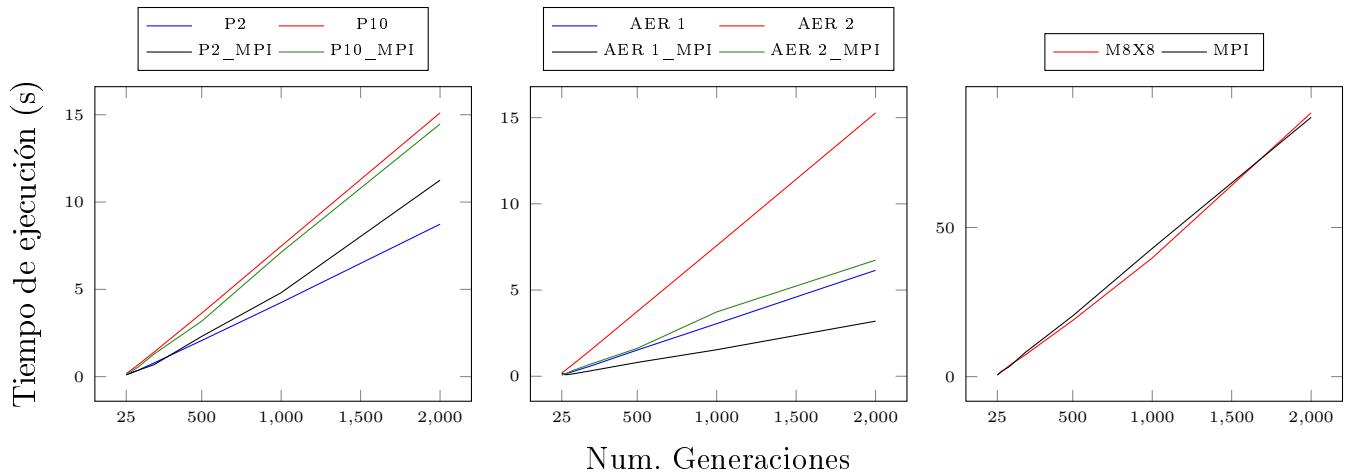


Figura 4.17: MPI1.1 - Dividir Poblacion

(Gráfica izquierda). Para los problemas binarios este método no es efectivo. Se pierde mucho tiempo en el paso de mensajes. Tener para cada individuo muchos bits provoca que

una población no muy grande sea inviable para aplicar esta mejora. Además de que este problema es bastante rápido.

(Gráfica derecha). Aunque se controle el tamaño de los individuos, el problema es muy pequeño para alcanzar alguna mejora. Con matrices más grandes se puede mejorar.

(Gráfica central). Para este tipo de problema hasta con valores pequeños se puede reducir el tiempo de ejecución. Aunque está lejos de llegar a un speedup ideal.

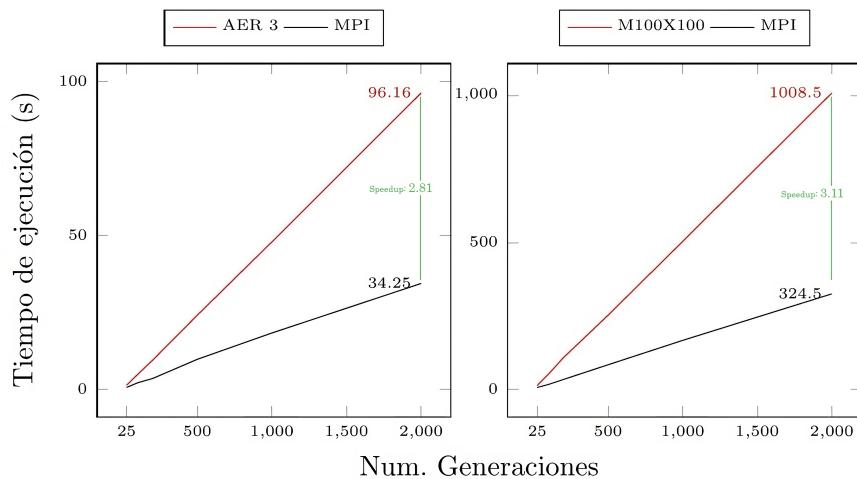


Figura 4.18: Tiempo de ejecución del algoritmo básico Jerarquico Aglomerativo

Como se comentó antes, al aumentar los tamaños se consiguen mejores resultados. Con estas características, la ejecución se aumenta al punto de ser buena opción aplicar esta mejora.

Mejora 3, PipeLine

El método de pipeline varía para cada problema. Los individuos binarios no mejoran mucho la ejecución, al tener muchos datos que enviar. Sin embargo los problemas con individuos reales o árboles si se pueden optimizar. La evaluación es el método que más tarda en estos dos problemas, por lo que repartir la carga de trabajo con más procesos reduce el tiempo de ejecución.

Para los individuos binarios, funciona algo mejor que la mejora anterior, con el funcionamiento de pipeline no se pierde tanto tiempo con el paso de mensajes de individuos binarios,

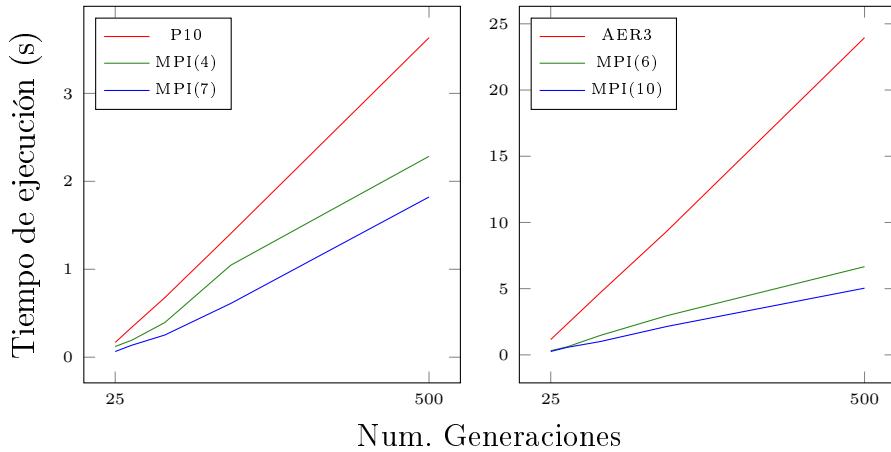


Figura 4.19: MPI3 - PipeLine

y aunque sea poco, se puede reducir el tiempo de ejecución. Al tener varias poblaciones ejecutándose al mismo tiempo no se puede tener una población muy grande. A partir de un tamaño de 1000 no se puede ejecutar con dos decimales de precisión. 500 para 10 decimales de precisión.

Binarios

1. Cuatro procesos:

- Master se encarga de inicializar
- Worker1: evaluación y selección, procesos que no tardan mucho en ejecutarse.
- Worker2: cruce
- Worker3: mutación

2. Siete procesos.

Se duplica el numero de workers en cada pipe.

Reales

Solo aumenta el número de workers en el método de evaluación.

1. Seis procesos:

- Master se encarga de inicializar
- Worker [1, 4]: evaluación, función que más tarda
- Worker 5: selección, cruce y mutación

2. Diez procesos.

Se duplica el numero de workers en cada pipe.

Arboles

Es igual que la implementación de individuos reales, pues la evaluación es el método que más tarda.

Estos datos se calcularon teniendo en cuenta los tiempos de ejecución para cada método (tabla del principio).

1. Evaluación $\approx 400\text{e-}06\text{s}$
2. Selección $\approx 8\text{e-}06$
3. Cruce $\approx 40\text{e-}06$
4. Mutación $\approx 10\text{e-}06$

Al juntar los últimos tres métodos, tarda un tiempo aproximado de $58\text{e-}06$.

6.9 veces más rápido que la evaluación. Por simplicidad, es mejor ejecutar potencias de dos procesos, para hacer una división equitativa

[**TODO CLUSTER**](#)

Redes Neuronales

Red neuronal con capa oculta 1x5, una capa y cinco nodos.

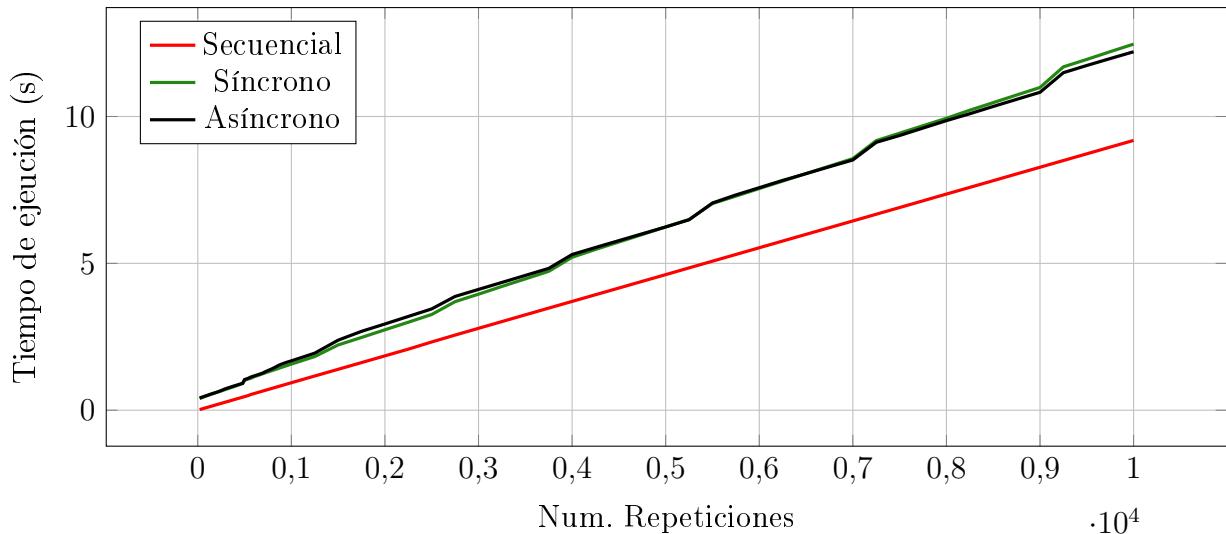


Figura 4.20: MPI1 - Red Neuronal

Una vez implementado para una red neuronal pequeña, no reduce el tiempo de ejecución. En programación evolutiva, el flujo de mensajes es unidireccional, y no se pierde tanto tiempo entre mensajes. Este algoritmo tiene dos métodos en diferentes direcciones, provocando un **flujo bidireccional**, y la comunicación entre procesos se ralentiza. Usando mensajes **asíncronos**, permite a cada proceso ejecutar antes el cálculo de forward y cuando recibe los errores los actualiza. Reduce muy poco el tiempo comparándolo con la versión síncrona, pero empeora la predicción del modelo. También hay que tener en cuenta que el flujo de mensajes hace que el modelo aprenda con valores desactualizados. Y dependiendo de la población puede haber un bucle en el cual aumenta y reduce los pesos, provocando un entrenamiento erróneo.

[TODO PROBAR CON 5X50](#)

Para dividir la población de entrenamiento, aplicando la idea de fine-tuning, entre los procesos hay que tener mucho cuidado. La repartición de individuos es crucial para un correcto aprendizaje de la red. Si cada proceso tiene la misma población de entrenamiento, se reduce el tiempo de ejecución en relación al número de procesos ejecutados, pero no garantiza buenas predicciones, pues la media sería parecida. A no ser que cada proceso inicialmente tuviese pesos distintos, aunque no se podría garantizar un buen entrenamiento. Sin embargo dividiendo la población de manera eficiente se podría intentar provocar que cada proceso aprendiese unos ciertos intervalos, y con una red grande ciertos nodos se especializan en unos datos, y no se entrelazan los resultados, llegando a reducir el tiempo y entrenar correctamente.

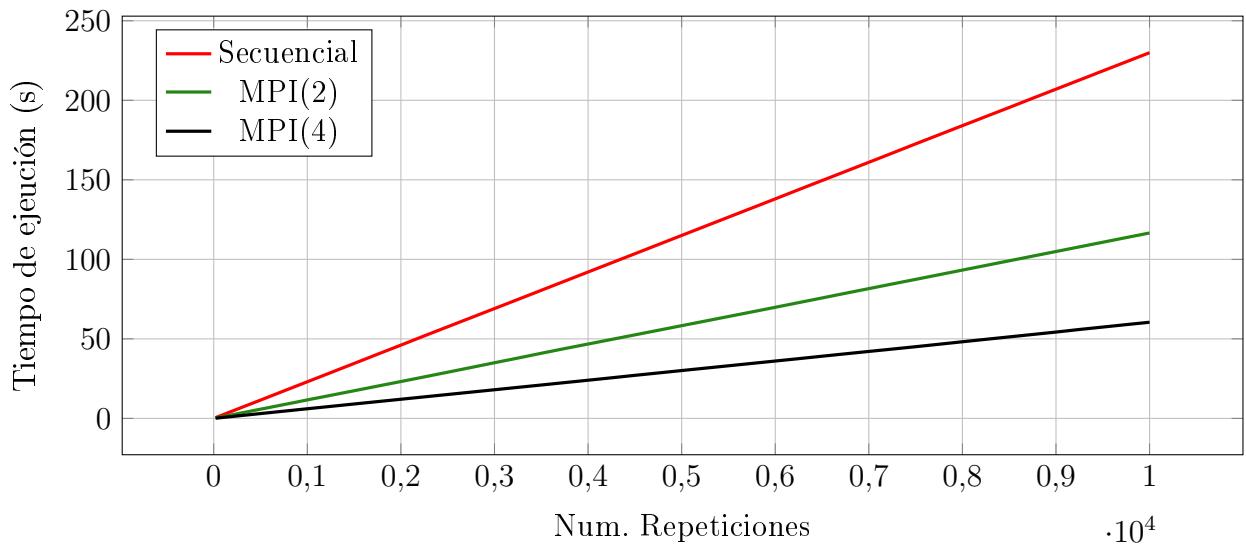


Figura 4.21: MPI2 - Red Neuronal Dividiendo entrenamiento

Esta prueba se ejecutó con 80 individuos en la población de entrenamiento, sobre una red neuronal con diez capas ocultas y diez nodos por capa (10x10). Si aumentamos o reducimos la población o la estructura de la red, el tiempo será proporcional. Al dividir la fase de entrenamiento se puede alcanzar un speedup aproximado al ideal, pero lo difícil es encontrar los valores concretos de los hiper parámetros y la repartición del entrenamiento para tener una buena red neuronal que cumpla con el funcionamiento deseado. Para ello se puede

realizar una búsqueda en paralelo comprobando los mejores resultados, tanto variando la tasa de aprendizaje como la repartición de los individuos con los cuales se entrena al modelo.

Para realizar la búsqueda de mejores hiper parámetros se ejecuta, con los mismos pesos, varias veces con diferentes valores, almacenando los mejores resultados y cuando se obtienen. Con cien repeticiones con un tamaño de población de ochenta individuos y una precisión de 0.01 se pueden comprobar los resultados en un intervalo de [0.01,0.20].

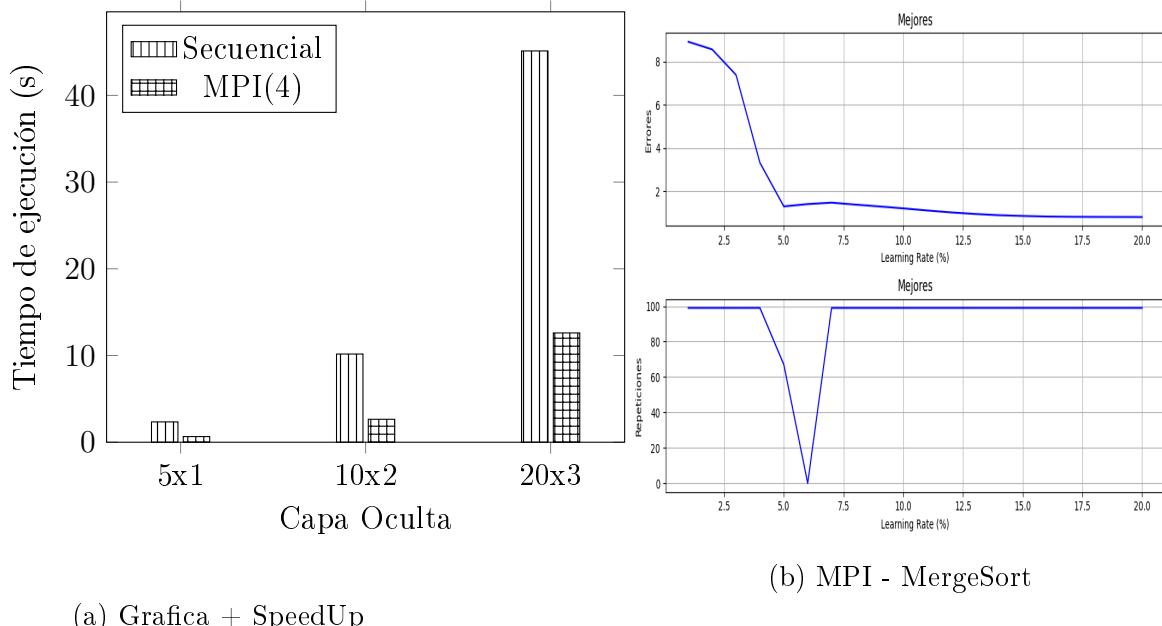


Figura 4.22: Mejoras MPI de las ordenaciones

Se ejecuta el algoritmo sobre varias configuraciones, en cada configuración siempre se utilizan los mismos pesos, para así comprobar cuales son los mejores hiper parámetros.

A la derecha se pueden ver los gráficos de la evolución, siendo el de arriba los errores cometidos en cada tasa de aprendizaje, y el de abajo cuando se obtienen menos errores.

Capítulo 5

Conclusiones y trabajo futuro

En este trabajo se han desarrollado varias mejoras en distintos algoritmos de IA, a través de la biblioteca estándar de paso de mensajes MPI. Los desafíos encontrados durante su desarrollo resultaron ser más complejos de lo que se había previsto inicialmente. Los problemas de configuración de la biblioteca MPI en windows, y adaptar la gestión de bibliotecas de Python usando Anaconda, fueron unos problemas completamente imprevistos. El desconocimiento general de MPI, se debe a la ausencia de asignaturas específicas de programación distribuidas en el grado de Ingeniería Informática, únicamente ofreciendo fundamentos teóricos, sin profundizar en la práctica. La escasa implementación práctica en las asignaturas de IA, en el itinerario Tecnología Específica de Computación del tercer curso ha derivado en tener que invertir más tiempo en investigar e implementar los algoritmos, proceso que he encontrado satisfactorio. La teoría vista en clase fue muy útil para el desarrollo del trabajo, pero usar exclusivamente la librería sklearn, de scikit-learn, provocó un desconocimiento de código para implementar estos algoritmos.

Una vez finalizadas las implementaciones, se ha llevado a cabo una fase de experimentación. Consistiendo en analizar los tiempos de ejecución, variando todos los parámetros disponibles, además de variar los conjuntos de poblaciones para cada tipo de algoritmo. Las ejecuciones de las pruebas requieren un coste computacional alto, además de mucho tiempo para finalizar. Para pruebas pequeñas no se consiguen apreciar reducciones significativas. Normalmente se pierde tiempo al paralelizar. Pero conforme aumentan los parámetros

introducidos, mejora notablemente el speedup de las mejoras.

Cabe destacar TODO

Como trabajo a futuro se propone investigar otros algoritmos de las técnicas desarrolladas. Además de investigar y mejorar otras técnicas de IA, como puede ser el procesamiento del lenguaje natural.

TODO CITAR TODO PARA LA BIBLIOGRAFÍA

Bibliografía

- [1] Marcel R Ackermann, Johannes Blömer, Daniel Kuntze, and Christian Sohler. Analysis of agglomerative clustering. *Algorithmica*, 69:184–215, 2014.
- [2] Brandon Barker. Message passing interface (mpi). In *Workshop: high performance computing on stampede*, volume 262. Cornell University Publisher Houston, TX, USA, 2015.
- [3] Marco A Contreras-Cruz, Victor Ayala-Ramirez, and Uriel H Hernandez-Belmonte. Mobile robot path planning using artificial bee colony and evolutionary programming. *Applied Soft Computing*, 30:319–328, 2015.
- [4] Flor A Espinoza, Janet M Oliver, Bridget S Wilson, and Stanly L Steinberg. Using hierarchical clustering and dendrograms to quantify the clustering of membrane proteins. *Bulletin of mathematical biology*, 74:190–211, 2012.
- [5] Frédéric Garcia and Emmanuel Rachelson. Markov decision processes. *Markov Decision Processes in Artificial Intelligence*, pages 1–38, 2013.
- [6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [7] Harold S Stone. *High-performance computer architecture*. Addison-Wesley Longman Publishing Co., Inc., 1990.
- [8] Christopher A Thomas and Xander Wu. How global tech executives view us-china tech competition. 2021.
- [9] Yi Wang, Kok Sung Won, David Hsu, and Wee Sun Lee. Monte carlo bayesian reinforcement learning. *arXiv preprint arXiv:1206.6449*, 2012.

Acrónimos

MPAI Message Passing Artificial Intelligence

MPI Message Passing Interface

[TODO...](#)