

---

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

---

MPAI-Boost: Optimización de algoritmos de IA aplicando técnicas enfocadas al 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 los 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 algoritmos.

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 propuestas para mejorar los algoritmos. 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 in reducing the execution time of AI algorithms by 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 algorithms. This project includes a description of the theoretical foundations of the algorithms that will be implemented. Moreover, functioning of the MPI library is also presented. Once put in context, the strategies employed to enhance the algorithms are described in detail. In addition, an 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

<b>Índice</b>	<b>I</b>
<b>Dedicatoria</b>	<b>III</b>
<b>1. Introducción</b>	<b>1</b>
1.1. Definición y alcance del proyecto . . . . .	1
1.2. Motivación . . . . .	3
1.3. Objetivo . . . . .	5
1.4. Estructura del documento . . . . .	6
<b>2. Contextualización</b>	<b>7</b>
2.1. MPI . . . . .	7
2.2. Aprendizaje por Refuerzo . . . . .	10
2.2.1. Algoritmo Q-Learning . . . . .	11
2.2.2. Deep Q-Network (DQN) . . . . .	12
2.3. Aprendizaje No-Supervisado . . . . .	14
2.3.1. Clustering jerárquico aglomerativo . . . . .	15
2.3.2. Clustering basado en particiones: K-Medias . . . . .	16
2.4. Aprendizaje Supervisado . . . . .	17
2.4.1. K-Vecinos más Cercanos - KNN . . . . .	18
2.4.2. Redes Neuronales . . . . .	19
2.5. Programación Evolutiva . . . . .	20
<b>3. Diseño e Implementación de estrategias para aumentar el rendimiento de algoritmos de IA</b>	<b>22</b>
3.1. Programas sencillos . . . . .	22
3.2. Algoritmos de Clustering . . . . .	28
3.2.1. Jerárquico Aglomerativo . . . . .	28
3.2.2. K-Medias . . . . .	31
3.2.3. K-Vecinos más cercanos (KNN) . . . . .	33
3.3. Aprendizaje por refuerzo . . . . .	36
3.3.1. Q-Learning . . . . .	36
3.3.2. Deep Q-Network . . . . .	40
3.4. Algoritmos Evolutivos . . . . .	45
3.5. Redes Neuronales . . . . .	50

<b>4. Estudio empírico</b>	<b>55</b>
4.1. Entornos de ejecución . . . . .	55
4.2. Programas sencillos . . . . .	57
4.2.1. Ordenaciones . . . . .	57
4.2.1.1. Algoritmos de complejidad cuadrática . . . . .	57
4.2.1.2. Algoritmo <i>MergeSort</i> . . . . .	59
4.2.2. Multiplicación de matrices . . . . .	62
4.3. Algoritmos de Agrupación . . . . .	64
4.3.1. Jerárquico Aglomerativo . . . . .	65
4.3.2. K-Medias . . . . .	69
4.3.3. KNN . . . . .	73
4.4. Q-Learning . . . . .	76
4.4.0.1. Mejora: Ejecuciones en paralelo . . . . .	78
4.4.0.2. Cluster . . . . .	79
4.5. PEV . . . . .	80
4.5.0.1. Algoritmos sin mejoras . . . . .	81
4.5.0.2. Mejora 2: Modelo de islas . . . . .	82
4.5.0.3. Mejora 1: Dividir con el master . . . . .	83
4.5.0.4. Mejora 3: PipeLine . . . . .	85
4.5.0.5. Cluster . . . . .	87
4.6. Redes Neuronales . . . . .	89
4.6.0.1. Mejora 1: PipeLine . . . . .	89
4.6.0.2. Mejora 2: Dividir el trabajo en procesos . . . . .	91
<b>5. Conclusiones y trabajo futuro</b>	<b>93</b>
<b>Bibliography</b>	<b>97</b>

# 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 las dificultades enfrentadas 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 y están mejorando nuestra calidad de vida. Desde la automatización de tareas hasta la asistencia virtual<sup>12</sup>, 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<sup>20</sup>.

El entrenamiento y ejecución de estos algoritmos para modelar inteligencias artificiales consumen mucha energía, además de provocar una cantidad excesiva de emisiones de CO<sub>2</sub>. La empresa tecnológica *Hugging Face*, desarrolladora de *BLOOM*<sup>1</sup>, la primera LLM (Grandes Modelos de Lenguaje) multilenguaje entrenada de forma transparente, tuvo la colaboración de muchos investigadores. Este proyecto, con 176 mil millones de parámetros, es capaz

de generar texto en 46 idiomas y 13 lenguajes de programación, pero estimaron que el entrenamiento de esta inteligencia artificial emitió 25 toneladas de CO<sub>2</sub>. Cifra que se duplicó al contar el coste de producción del equipo informático usado<sup>10</sup>. 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*<sup>9</sup> (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<sup>15</sup>.

El uso de la programación distribuida, más específicamente, aplicaciones basadas en MPI, permite ejecutar varios procesos en paralelo, dividiendo la carga de trabajo, reduciendo así el tiempo de ejecución. Al contrario de los programas basados en el modelo de memoria compartida, en el cual se pueden dar problemas de 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 complejidad espacial de la esperada. Las conexiones entre los procesos se pueden configurar 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 procesos *Workers*, para, en paralelo, ejecutar la misma tarea pero con conjuntos mucho más reducidos de datos. 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 optimi-

zación 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 adecuada 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 tiene que ser controlada para el correcto funcionamiento, y, en algunas ocasiones, estas mejoras derivan en implementaciones más complejas.

Generalmente, en estos casos, 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 costes de sus sistemas. Para ello, es necesario un uso eficiente de los recursos computacionales. Además, existe un interés en reducir el consumo de energía y las emisiones de CO<sub>2</sub>, 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 excesivamente significativa, es un avance.

### 1.3. Objetivo

El objetivo principal de este trabajo es **paralelizar algoritmos de IA, empleando técnicas de HPC para reducir 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<sup>18</sup>. Sin embargo, al ser un lenguaje interpretado (el código se traduce en la misma ejecución), aumenta la sobrecarga y hace que -generalmente- 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, es posible utilizar distintos números de procesos en la ejecución. 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. 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, es necesario tener una cohesión con el algoritmo original. Es decir, si queremos maximizar una función de evaluación, la implementación de la mejora tiene que proporcionar resultados parecidos o mejores. No resulta útil implementar una mejora que reduzca el tiempo de ejecución, pero que obtenga resultados peores.
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, posteriormente, se analizan 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.

## 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.
- Capítulo 3: Diseño e implementaciones. Este capítulo comienza describiendo ejemplos básicos fuera del ámbito de la inteligencia artificial. Seguidamente, se muestran las implementaciones desarrolladas para las diferentes técnicas abordadas.
- Capítulo 4: Estudio empírico, presenta el proceso experimental realizado, el cual consiste en analizar las mejoras propuestas, variando el número de procesos y el tamaño de los datos.
- Capítulo 5: Conclusiones y trabajo a futuro. El último capítulo finaliza el trabajo con una síntesis de los resultados obtenidos, y presenta la proyección del trabajo a futuro.

# Capítulo 2

## Contextualización

En este capítulo se presenta una descripción de los algoritmos de Inteligencia Artificial que se van a utilizar en el proyecto, así como sus usos y características.

### 2.1. MPI

Message Passing Interface<sup>5</sup> (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, mediante el envío y recepción de mensajes. Comúnmente se utiliza en sistemas de alto rendimiento<sup>19</sup> (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 (*rank=0*) generalmente, es llamado *Master*, y se encarga de distribuir los datos entre los demás procesos, llamados *Workers*. La Figura 2.1, muestra la comunicación entre los procesos en este modelo. El proceso *Master* reparte los datos mientras que los *Workers* los procesan y los devuelven.

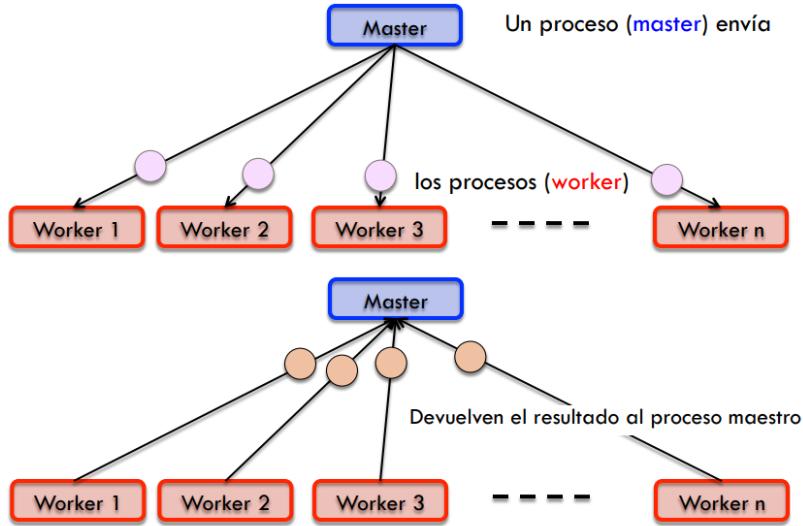


Figura 2.1: Comunicación Master-Worker

Esta técnica para paralelizar utiliza memoria distribuida, es decir, cada proceso tiene su propia memoria local. Así los procesos no tienen que preocuparse por los problemas de la memoria compartida, como la sincronización para el acceso de variables compartidas, *riesgo de carrera* o *deadlocks*. Asimismo, la memoria compartida no es fácilmente escalable a un gran número de procesadores<sup>17</sup>.

Un programa ejecutado en paralelo, donde múltiples procesos se ejecutan en el mismo programa de manera independiente, pero trabajan con diferentes conjuntos de datos. Se denomina SPMD, este modelo es comúnmente utilizado en computación de alto rendimiento y en entornos de procesamiento paralelo. La escalabilidad y eficiencia de este modelo son sus principales ventajas.

Un proceso (*Master*) contiene los datos del programa y se encarga de gestionar los mismos y repartirlos de manera eficiente. 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. Los emisores tienen que llamar a la misma función para recibir el mensaje.

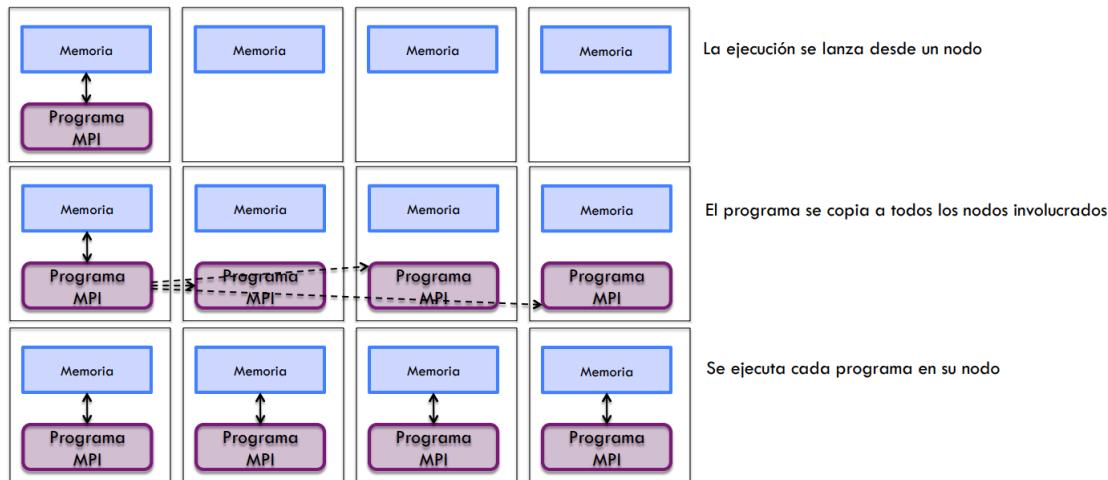


Figura 2.2: Ejecución MPI

Un programa MPI (ver Figura 2.2), 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 de los datos, envía los datos procesados al proceso correspondiente. La Figura 2.3 representa en código esta idea.

```

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, por sus siglas en inglés), 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 cumplir un objetivo en un entorno ejecutando un determinado número de acciones. Este tipo de algoritmos no requiere de entradas etiquetadas como en el aprendizaje supervisado, sino que recibe una retroalimentación, *feedback* en inglés (recompensas o castigos), al realizar acciones en los estados. Aprendiendo con prueba y error, el agente explora el entorno para almacenar las mejores acciones para cada estado.

Los componentes esenciales del algoritmo son los siguientes:

- Agente, que interactúa con el entorno y aprende de él, ejecutando sus acciones.
- Entorno, con el cual el agente interactúa. Responde a las acciones tomadas por el agente y provee el *feedback*.
- Conjunto de acciones o decisiones que el agente puede realizar.
- Estados, son las configuraciones que el entorno puede tomar.

- Feedback, recompensas o castigos del entorno al realizar una acción en un estado.
- Condición de finalización. La cual abarca desde encontrar la función óptima, hasta realizar un número de acciones

### 2.2.1. Algoritmo Q-Learning

El algoritmo Q-Learning es una mezcla entre programación dinámica y Monte Carlo<sup>21</sup>. 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, por sus siglas en inglés)<sup>8</sup>.

El procedimiento se realiza actualizando iterativamente las estimaciones de calidad de realizar dicha acción en el estado actual, conocido como valor-Q. Se suele representar en forma de matriz  $Q(S,A)$ , guardando los valores-Q de las acciones en los estados. Este valor representa cómo de buena es la acción a realizar en un estado después de realizar una etapa de entrenamiento. Para ello, la Figura 2.4 muestra la fórmula para actualizar los valores, para cada acción tomada por el agente.

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

- $Q(S, A) \leftarrow$  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.
- $\max_i(Q(S', A_i))$  : Es el valor máximo obtenible de realizar las posibles acciones en el estado siguiente.

Figura 2.4: Cálculo del Q-Value de un estado y acción

El agente toma las decisiones de ejecutar una acción dependiendo del hiper-parámetro  $\epsilon$

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 acción aprendida hasta el momento, y es posible que no aprenda otras formas de alcanzar el objetivo.

Este algoritmo se ha aplicado en muchos dominios, como puede ser videojuegos de Atarí<sup>14</sup>, robótica o problemas de optimización. Sin embargo sufre cuando el entorno tiene muchos estados, ya que la complejidad espacial aumenta considerablemente, y no es práctico tener las dos matrices. Por eso se diseñó el algoritmo DQN que usa una red neuronal. Así eliminamos la maldición de dimensionalidad<sup>15</sup>, problemas que surgen con el exceso de variables independientes en un dataset. En este algoritmo, el problema es el elevado número de estados que el agente ha de recorrer.

### 2.2.2. Deep Q-Network (DQN)

Deido a los problemas de escalabilidad mencionados anteriormente, se desarrolló el algoritmo de Redes Neuronales Profundas (DQN, por sus siglas en inglés). Este algoritmo combina redes neuronales con la base de aprendizaje por refuerzo, eliminando así la Q-Table.

La estructura de la red neuronal depende del entorno del problema. Los valores de la capa oculta se pueden modificar dependiendo de las necesidades del programador, pero la capa de entrada y salida depende del problema. La entrada se adapta para recibir un estado del entorno, y la salida tendrá tantos nodos como las acciones del agente.

El problema a realizar tendrá dos redes neuronales, una del estado actual y otra de antícpio, es decir, el siguiente estado. Esto sirve para ayudar a tener más contexto del estado actual, pues con una sola imagen (estado del juego), la información del estado puede variar considerablemente. En la fase de entrenamiento se realizan varios episodios, que consisten en ejecuciones hasta que se dé una condición de finalización. Además de ejecutar repeticiones

de estados guardados anteriormente (replay buffer). En este algoritmo se usan tres hiperparámetros:

1. Gamma, factor de descuento. Utilizado para saber cuánto resta a la recompensa adquirida al realizar una acción en un estado.
2. Epsilon, tasa de exploración. Probabilidad utilizada para ejecutar una acción aleatoria o la mejor hasta el momento.
3. Learning rate, tasa de aprendizaje. Para la propagación hacia atrás de las redes neuronales. Esencialmente, mide cuánto cambian los pesos de los nodos al tener un fallo.

En este trabajo, el entorno del problema será el juego de la empresa *Namco*, la versión de *Atari 2600*, *Pac-Man* (ver Figura 2.5). El juego consiste en recolectar todas las monedas del laberinto sin ser comido por un fantasma. Implementamos el juego -desde cero- para moldear a nuestro gusto la implementación y que el algoritmo DQN sea más eficiente y sencillo. En el capítulo 3, diseño e implementaciones, se desarrolla en profundidad.



Figura 2.5: Juego Pac-Man versión Atari 2600

## 2.3. Aprendizaje No-Supervisado

Los métodos no supervisados (unsupervised methods, en inglés) 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 estructuras 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, entre otros, la 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 ello, cual medidas ya estudiadas como el diagrama de “codo”, cuyo valor óptimo de clusters se calcula visualmente, cuando empieza a crearse un codo (la diferencia con el número anterior no es tan pronunciada como en puntos anteriores). Hay otros coeficientes que calculan la optimalidad con algoritmos, como el coeficiente de Davies-Bouldin, cuyo valor mínimo indica el número óptimo de clusters, o el coeficiente de Silhouette, siendo similar al anterior, pero con el valor máximo. Se pueden apreciar los diferentes coeficientes en la Figura 2.6.

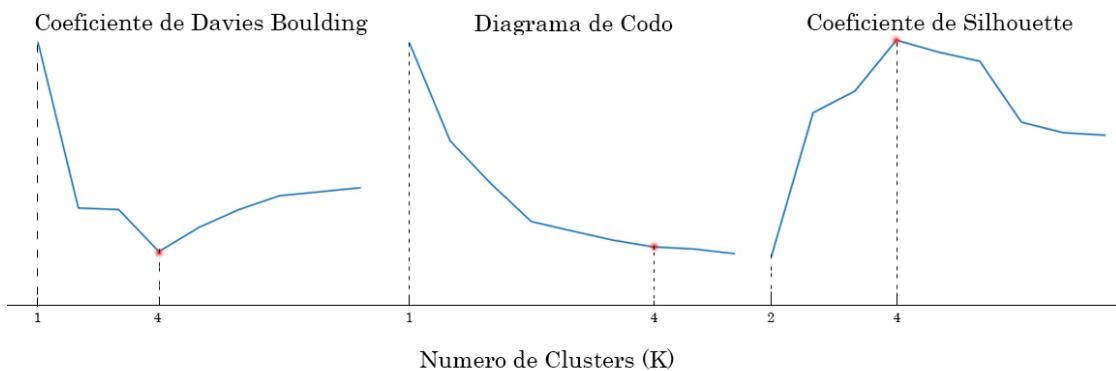


Figura 2.6: Coeficientes

Los llamados métodos jerárquicos<sup>3</sup> 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  $N$  clusters, 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 distancias entre *clusters* pueden ser:

- Centroides: cada cluster tiene un centro.
- Enlace simple o compuesto: la distancia entre *clusters* 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)
    busqueda_min(poblacion) // Agrupar los cluster (i, j)
    agrupar_clusters(poblacion, i, j); // Eliminar la fila y columna de mayor índice
    eliminar_cluster(poblacion, max(i, j))
    // Calcular nuevas distancias al cluster agrupado (i)
    nuevas_distancias(poblacion, min(i, j))

```

---

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 dendrograma<sup>7</sup>, y comprobar el número óptimo de *clusters* para la población calculada. Sin embargo, no es igual de preciso que los coeficientes mencionados anteriormente. (ver Figura 2.7)

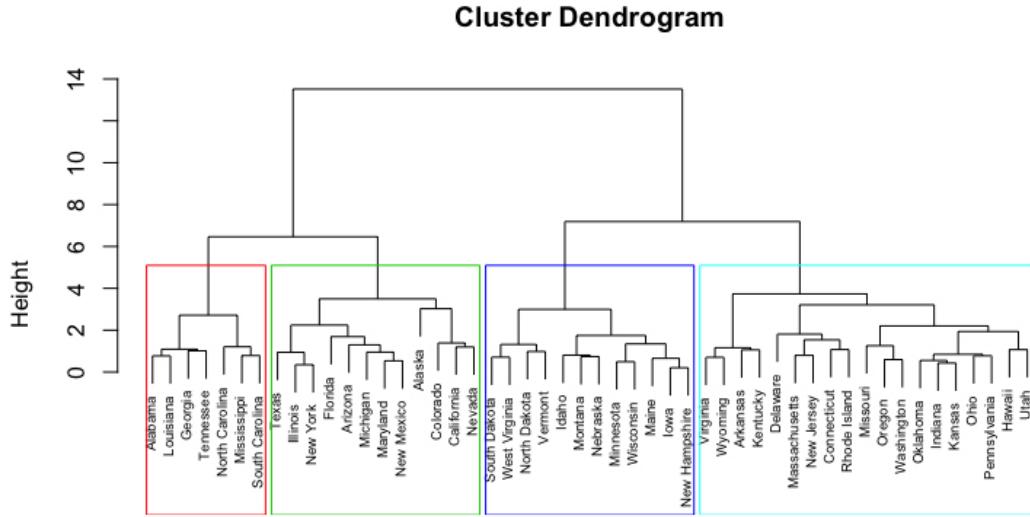


Figura 2.7: Dendograma

### 2.3.2. Clustering basado en particiones: K-Medias

La meta de este algoritmo es particionar la población inicial en  $K$  clusters, 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

**Result:** agrupacion

centrosNuevos := init(K) // Inicializar los centros de manera aleatoria

**repeat**

centros := centrosNuevos

agrupacion := asignar(poblacion)

centrosNuevos := calculaCentros(poblacion, asignacion)

**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. La Figura 2.8 muestra una ejemplo de ejecución de este algoritmo. En este caso, se ejecuta varias veces el algoritmo, para ver las posibles asignaciones. El intento con menor valor del sumatorio de distancias de individuos y su cluster asignado, es la mejor asignación entre los intentos ejecutados.

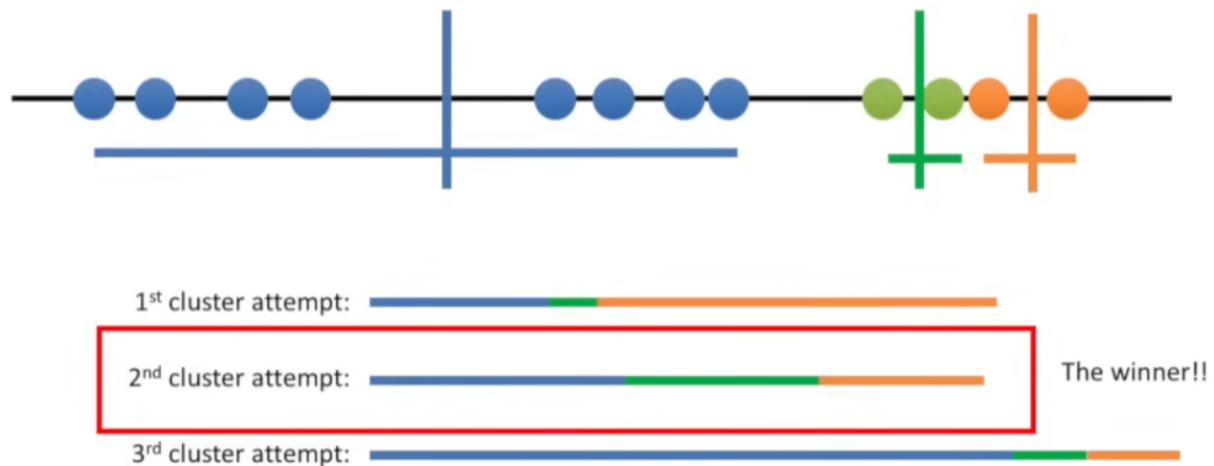


Figura 2.8: KMedias - Búsqueda

## 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. Para ello, 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

KNN es un algoritmo 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. Perteneces 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, de forma que hay que probar con varias ejecuciones. Valores pequeños de K crea sonido, provocando que inicialmente se categorice con un cluster no tan idóneo, y a la larga se categoricen muchos de forma incorrecta. Valores grandes con pocos datos favorece a los *clusters* con mas individuos. Un valor diferente de K puede cambiar la categoría de un individuo. En la Figura 2.9a 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 variarmos 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 := ∅
for each individuo ind in poblacionPred do
    // Recorrer toda la población guardando los K individuos más cercanos
    vecinos := recorrer_poblacion(poblacion, individuo)
    // Clasificar según los K vecinos
    cluster := clasificar_individual(vecinos)
    agrupacion.append(cluster, etiquetas)
return agrupacion
```

---

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

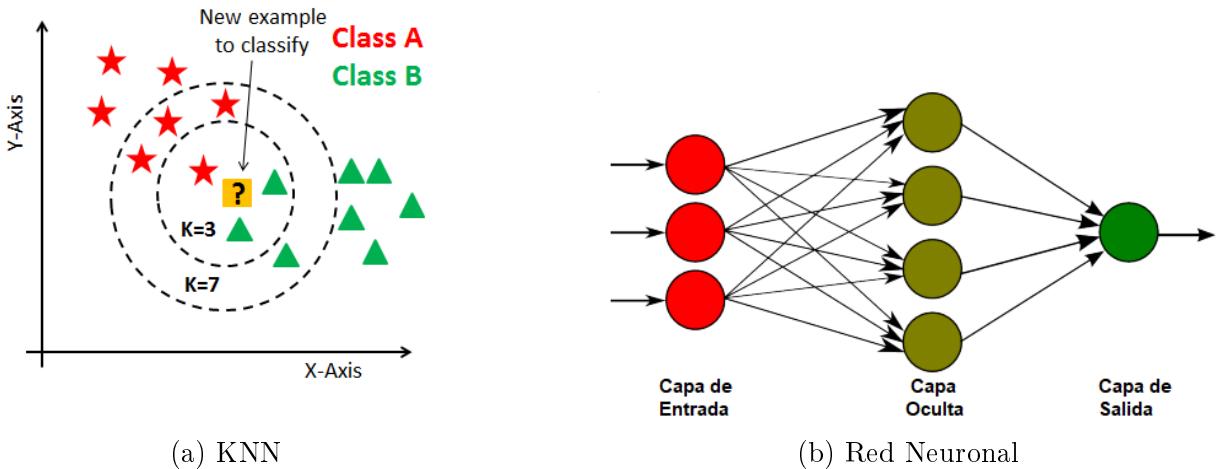


Figura 2.9: Aprendizaje Supervisado

#### 2.4.2. Redes Neuronales

Las redes neuronales son un modelo computacional inspirado en el funcionamiento y estructura de las neuronas del cerebro humano. Consiste en capas de nodos interconectados, llamadas neuronas artificiales. La estructura del modelo se muestra en la Figura 2.9b. En este modelo se aprecian:

- 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 contiene un número determinado 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, este ajuste se lleva a cabo con algoritmos de optimización como descenso de gradiente, donde se comparan las predicciones del modelo con la categoría correcta, y se actualizan 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
    cont := 0 for each 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.
        predicion := forward(pesos, ind)
        // 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(pesos, predicion, etiqueta[cont]) cont++
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 no proporcionan el rendimiento deseado. Se han aplicado a varios dominios, como por ejemplo la bioinformática o robótica<sup>6</sup>.

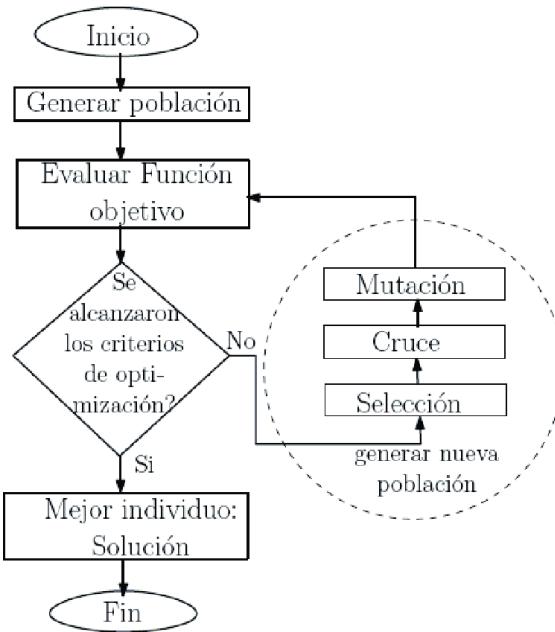


Figura 2.10: Algoritmo Evolutivo

# Capítulo 3

## Diseño e Implementación de estrategias para aumentar el rendimiento de algoritmos de IA

En este capítulo se presentan los diseños e implementaciones desarrollados a lo largo del trabajo. Inicialmente se presenta una introducción a la librería MPI, mejorando programas sencillos fuera del ámbito de la inteligencia artificial. Posteriormente, se desarrollan los algoritmos de IA, empezando por los más sencillos y terminando por los más complejos.

### 3.1. Programas sencillos

Para introducir MPI en el proyecto se implementan y comentan varios programas sencillos. Primero la multiplicación de matrices, que tiene un coste cúbico  $O(N^3)$ , al tener que recorrer, para cada elemento de la matriz, una fila y columna entera. Segundo, algoritmos de ordenación, que para simplificar, solo se realiza 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 conllevan 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  $O(N^3)$ . Esencialmente, es necesario recorrer toda la matriz, y para cada elemento, realizar el sumatorio de las multiplicaciones fila, columna.

Por ello, es un buen primer ejemplo para presentar una estrategia basada en MPI, debido al alto coste computacional. Para ello hay que plantear cómo dividir el trabajo entre los procesos.

Inicialmente, se puede pensar que es mejor enviar 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, paralelizando así el cálculo. El *master* se encarga de dividir la matriz entre los procesos, y se puede abordar con dos enfoques distintos.

- Reparto estático: Los datos se dividen antes de empezar el cálculo.
- Reparto dinámico: Los datos se reparten en tiempo de ejecución, asignando los mismos de forma proporcional a la velocidad de cada proceso.

En la primera estrategia, dividimos la matriz entre todos los *workers*, dejando al *master* en espera de recibir datos. Esta mejora depende de la velocidad de los procesos, pues se puede generar un cuello de botella si todos terminan y envían los datos al mismo tiempo. Además de aumentar la complejidad espacial entre los procesos, pues se divide la matriz entera entre los *workers*.

En la segunda estrategia, hay un flujo constante de nuevos datos y resultados obtenidos, reduciendo el tiempo perdido en un posible cuello de botella. En la Figura 3.1 se muestra como el *master* divide la matriz, marcando en negro la parte ya procesada. A su vez cada *worker* tiene una sola fila en su memoria, reduciendo la complejidad espacial.

Los algoritmos de ordenación tienen que iterar varias veces hasta que el array de elementos esté completamente ordenado. Y los métodos pueden variar considerablemente el

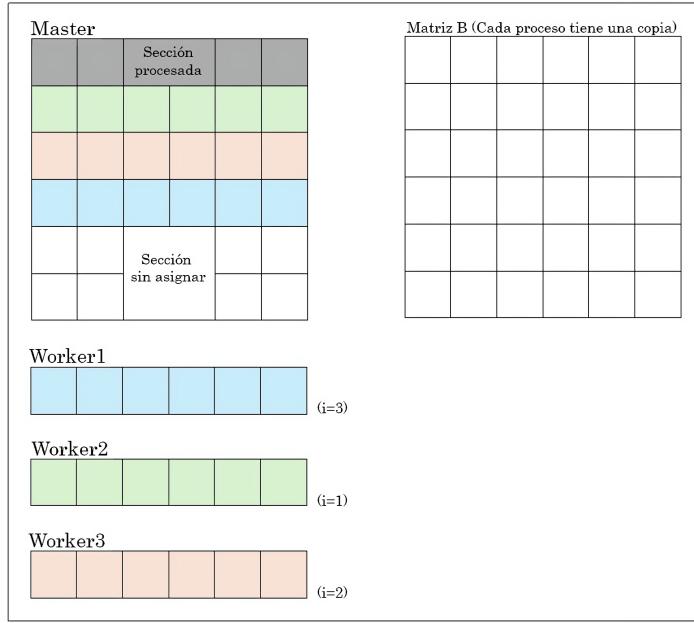


Figura 3.1: MPI - División de datos (ejemplo de multiplicación de matrices)

tiempo de ejecución.

Para las ordenaciones cuadráticas, los métodos populares como BubbleSort, Insertion-Sort y SelectionSort, han sido estudiados y optimizados para que, aunque tengan un coste cuadrático  $O(N^2)$ , en el caso peor, proporcionen un buen rendimiento. Basándose en estos algoritmos, se ha diseñado uno adicional llamado SequentialSort. Este algoritmo recorre todas las posiciones del array, y para cada elemento compara todos los datos, sumando en un contador los elementos mayores que el, para calcular así su posición en el array ordenado. Una vez finalizada una iteración, se coloca el elemento en el array ordenado, si la posición actual esta ocupada es porque hay una repetición del elemento, y se tiene que colocar en la siguiente celda libre. La Figura 3.2 representa el proceso de ordenación, marcando en gris el elemento que ha de compararse con los demás en la iteración i-ésima.

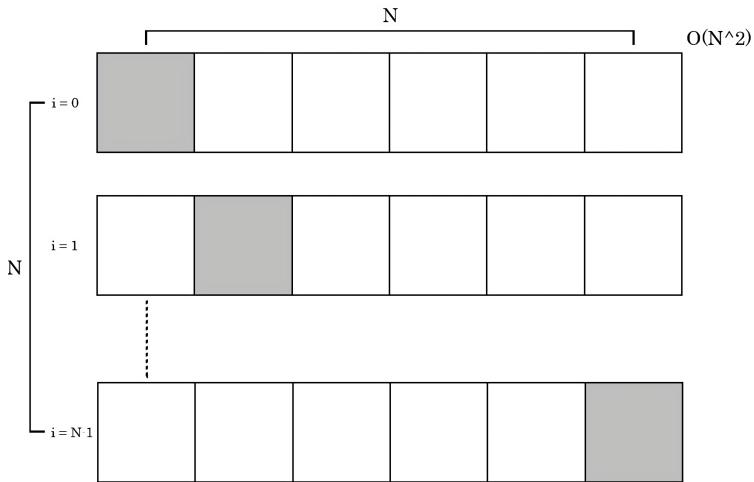


Figura 3.2: MPI - SequentialSort

Este método siempre tendrá coste cuadrático  $O(N^2)$ . No es como los anteriores que van reduciendo el espacio conforme aumentan las iteraciones, pero es fácilmente paralelizable.

Para lograr el objetivo, se desarrollan las siguientes dos estrategias. Cada con sus ventajas e inconvenientes.

1. Enviar a todos los *workers* el array entero para que trabajen de manera independiente.
2. Dividir el array entre los *workers* para trabajar conjuntamente.

En la primera estrategia el *master* envía a todos los *workers* el array entero. Una vez recibido el array de elementos, el *master* envía elementos sin procesar del array orginal a los *workers* disponibles. Al recibir un elemento lo procesan (hacen las comparaciones), y envían la posición del elemento al *master*, recibiendo de vuelta otro si todavía faltan elementos por procesar.

No obstante la segunda estrategia se logra reducir el uso de memoria de tal forma que entre todos los procesos ejecutados solo haya dos copias del array que hay que ordenar, en lugar de mantener  $M$  copias (siendo  $M$  el número de procesos ejecutados). En cada iteración, el *master* envía un elemento a todos los *workers*. Estos hacen todas las comparaciones en sus

subarrays y devuelven cuantos elementos son mayores que el recibido por parte del *master*. Ambas estrategias tienen la misma complejidad temporal.

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 eficientes sin aplicar mejoras, y MergeSort es muy popular, tanto que se aplica en Python para el método de ordenación por defecto, TimSort<sup>4</sup>. Este último combina InsertionSort, una ordenación cuadrática muy eficiente para ordenar para ordenar pequeños conjuntos de datos, teniendo una baja sobrecarga en términos de operaciones. 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 simplicidad, el número de procesos tiene que ser potencia de dos), y se divide el array entre los procesos. Las fase de esta mejora son:

- Primera fase de ordenación: cada proceso ordena su sub-array con el método de ordenación correspondiente. En el capítulo 4, se realiza un estudio de los algoritmos cuadráticos y SelectionSort es el algoritmo 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.

En la comunicación entre procesos, cada uno se conecta con el proceso activo más cercano. El proceso de mayor ID (*rank*) envía su array ordenado y finaliza su ejecución. El proceso receptor se encarga de ordenar ambas mitades en una sola. Utilizando una barrera (barrier MPI), garantizamos que todos terminen al mismo tiempo. Esta mejora aplica la idea de sincronización con *barrera simétrica mariposa*. Esta técnica de sincronización conecta los

procesos dos a dos, aumentando la distancia de los procesos para que en aproximadamente  $M$  iteraciones ( $2^M =$  número de procesos), todos los procesos estén sincronizados. Para la mejora implementada, se sincronizan por orden de cercanía entre IDs. La Figura 3.3 muestra el proceso de sincronización, en cada iteración se finaliza la ejecución de los procesos en rojo.

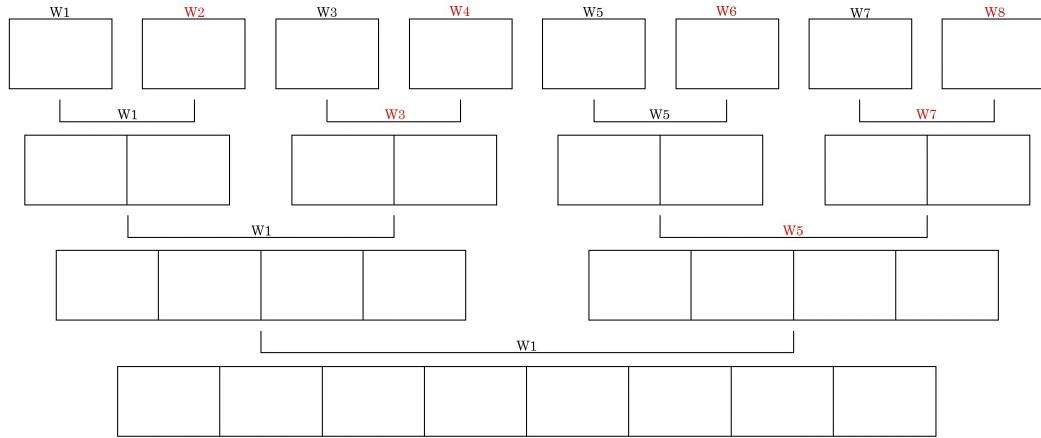


Figura 3.3: MPI - MergeSort con 8 procesos worker  $W_1 \dots W_8$

Para aplicar MPI y paralelizar programas hay que tener en cuenta que la comunicación entre procesos requiere un tiempo para enviar/recibir mensajes. Si queremos reducir el tiempo de ejecución de un programa tenemos que asegurarnos que la estrategia es viable para mejorar el rendimiento. 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*?

Para responder esta pregunta es necesario tener en cuenta el tiempo de paso de mensajes (overhead). Si no se tuviese en cuenta, se podría garantizar la reducción, pero la comunicación entre procesos tiene un coste, y con un tiempo lineal, generalmente 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, ya que no siempre es eficiente aplicar paralelismo.

## 3.2. Algoritmos de Clustering

Una vez introducido MPI con programas básicos, podemos presentar las implementaciones de los algoritmos relacionados con la inteligencia artificial. Las técnicas de clustering toman una población y, dependiendo del conjunto de datos, categoriza los individuos. Los algoritmos pueden ser supervisados, si ademas de la población a categorizar, tenemos un poblacion categorizada previamente, o no-supervisados, 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).

El calculo de la distancia por enlace simple y completo, es más complejo. Cada cluster almacena las coordenadas de sus individuos, para, a la hora de calcular la distancia entre dos clusters ( $C_i$  y  $C_j$ ), comprobar la distancia de cada par de puntos (donde uno pertenece a  $C_i$  y el otro a  $C_j$ ). La nueva distancia usando enlace simple es la mínima distancia entre cualquier par de puntos, mientras que la completa es la máxima distancia.

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 realizar sumatorios de multiplicaciones (suma de productos, para la multiplicación de matrices), pues la matriz está en constante cambio. Tendría que realizarse un proceso de comunicación constante para gestionar la matriz. Esto y añadir más operaciones del algoritmo para agrupar los individuos, provoca que no sea viable realizar esta mejora. Si dividimos la matriz entre los *workers*

cada proceso se encarga de una zona y paralelizamos así el trabajo a realizar.

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, parando la ejecución por “culpa” del primer proceso. La Figura 3.4 muestra el cálculo de elementos a procesar entre el primer y último *worker* si no se divide el espacio de manera óptima.

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

$N$  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:

- $W_1$  tiene las filas de 1-25, con 2175 elementos.
- $W_4$ , las filas de 76-100, con solo 300 elementos.

$W_1$  tiene 7.25 veces más elementos, no se reducirá el tiempo de ejecución.

Figura 3.4: Jerárquico Aglomerativo - Cálculo de elementos a procesar usando una distribución secuencial de filas

Dividiendo las filas por pares (parte superior, parte inferior) conseguimos una distribución mucho más eficiente. La Figura 3.5 muestra como se distribuyen las filas en cada worker. Así cada *worker* tiene aproximadamente el mismo número de elementos que calcular y analizar, inicialmente. Puede variar si el número de filas no es divisible entre en número de procesos *workers*

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.

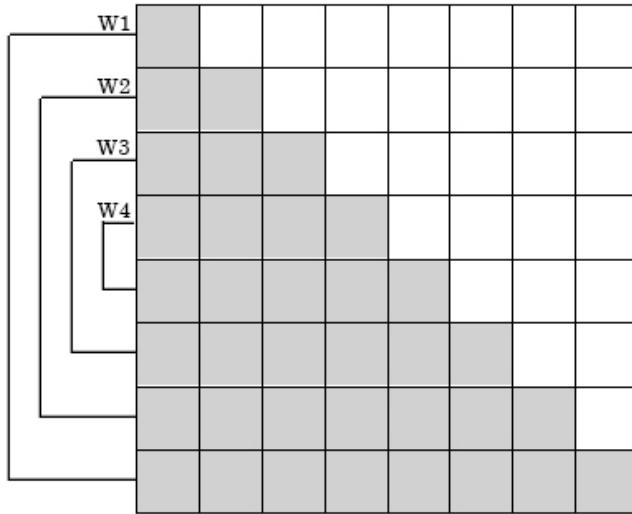


Figura 3.5: Jerarquico Aglomerativo - División de las filas

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*.

El objetivo es encontrar una manera lo más optimizada posible 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 entre los procesos disponibles.
- Sin embargo, cuando aumenta el número de individuos, se reducen las columnas y esta idea ya no resulta viable. Cada proceso requiere de un cómputo significativo realizando los cálculos. Por eso, es mejor calcular las distancias de forma escalonada usando todos los *worker*, dividiendo los individuos del *cluster* para encontrar la distancia mínima (simple) o máxima (completa).

Antes de procesar los datos, el *master* divide las filas con el método comentado.

El siguiente proceso se repite hasta que el solo haya  $C$  *clusters*: **TODO CREAR UNA**

## FIGURA CON PHOTOSHOP. POR AHORA DEJO EL PROCESO

El master, que se encarga de:

- Dividir las filas con el método 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, optimiza 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  $c_2$ ).
  - Todos eliminan los elementos de la columna  $c_2$ , y actualizan los elementos de la columna  $c_1$ .
  - El worker con  $ID=c_1$  actualiza la fila  $c_1$ .
  - El worker con  $ID=c_2$  elimina la fila  $c_2$ .

Figura 3.6: Jerarquico Aglomerativo - Proceso de la mejora

### 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 del algoritmo anterior, no se usa una matriz, y solo se usa distancia por centroides. Una posible mejora puede consistir en las siguientes estrategias:

1. Reparto estático. Dividir la población entre los workers, como muestra la Figura 3.7.
2. Reparto dinámico. En cada iteración del algoritmo, *master* envia individuos a los *workers*. Cuando finalicen el procesado, envían de vuelta la agrupación y esperen a recibir otros individuos.

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.

TODO CREAR UNA FIGURA CON PHOTOSHOP. POR AHORA DEJO EL PROCESO

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 implementar la primera opción, reparto estático, no es una buena idea implementar un reparto dinámico, pues el constante flujo de mensajes aumenta el tiempo de ejecución. La población a categorizar no cambia, por lo que distribuir la población una única vez en toda la ejecución es más eficiente. Lo único que cambia en cada iteración es la asignación de los individuos, pues el algoritmo finaliza cuando la asignación no varíe entre iteraciones (centros de los clusters).

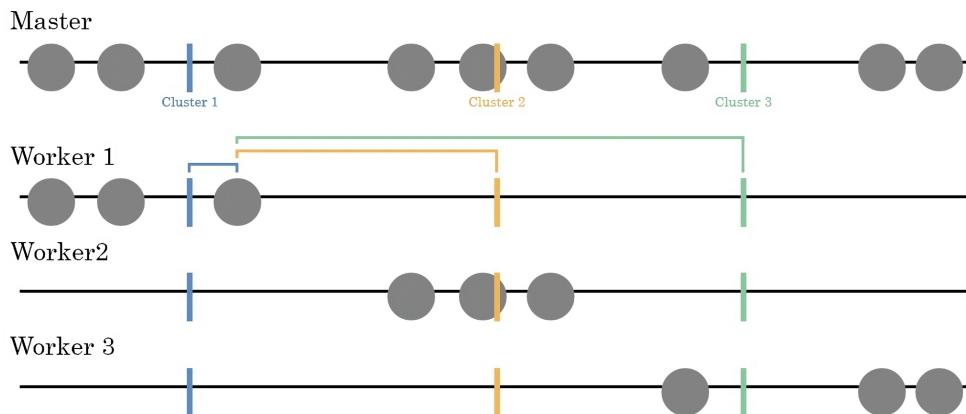


Figura 3.7: KMedias - División de poblaciones

Como se comentó en el capítulo anterior, este algoritmo se tiene que repetir varias veces para encontrar la mejor asignación, el óptimo general. A su vez, también hay que variar el valor de  $K$  para buscar la mejor asignación para los individuos. Para lograr este objetivo se puede enfocar de dos diferentes formas:

1. Aplicar la implementación anterior con varios bucles. El externo para variar el valor de  $K$  y el interior para repetir el algoritmo.
2. Ejecutar en cada proceso *worker* el algoritmo sin mejoras. Los *workers* ejecutan la búsqueda con valores distintos de  $K$  y el *master* se encarga de almacenar los mejores resultados.

Como muestra la Figura 3.8, ambas ideas tienen el mismo coste temporal, sin contar el overhead (mensajes MPI). Pero, la segunda opción al tener en cada proceso una copia de la población entera tiene mayor complejidad espacial, posiblemente causando un menor rendimiento.

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

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

$N$  = número de individuos en la población

$M$  = número de procesos *workers*

$K$  = número de clusters

$\text{Rep}$  = repeticiones para buscar el óptimo general

$K_{\max}$  = valor máximo de  $K$  en la búsqueda.

Figura 3.8: KMedias - Comparación temporal de las mejoras para la búsqueda del óptimo general

### 3.2.3. K-Vecinos más cercanos (KNN)

Al contrario de los algoritmos anteriores, KNN (K-Nearest Neighbors en inglés) pertenece al aprendizaje supervisado, por lo que necesita una población ya categorizada para poder agrupar los nuevos individuos. Al igual que el algoritmo K-Medias, y como menciona su nombre, esta técnica de clustering utiliza una variable  $K$ , que representa los vecinos que se van utilizar para categorizar los individuos.

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 de forma significativa. Aunque es menos precisa a la hora de predecir, el algoritmo es más veloz al no aumentar la población categorizada, pues no tiene que recorrer un individuo más conforme se categoriza la población a predecir. Al contar con una población extra (la categorizada) se propone dos estrategias posibles:

1. Dividir la población categorizada entre los *workers* (ver Figura 3.9a)
2. Dividir la población a predecir entre los *workers* (ver Figura 3.9b)

Si dividimos la población categorizada (primera estrategia), los *workers* cuentan con menos cómputo en cada individuo. Comparan el individuo a predecir con su subpoblación, y el *master* tiene una mayor carga de trabajo, al recibir los  $K$  vecinos de cada *worker* y tener que ver los  $K$  mejores de todos los individuos recibidos. Mientras que el *master* comprueba las distancias recibidas, los *workers* operan en la siguiente iteración, enfocándose en el próximo individuo a predecir.

Para la actualización de individuos, el *master* reparte de forma equitativa los individuos que se categorizan. En cada iteración envía el individuo categorizado a uno distinto, utilizando un contador con el *ID* del *worker* respectivo.

Con la división de la población a predecir (segunda estrategia), cada *worker* realiza un cómputo equitativo, pero predicen menos individuos. El *master* no realiza tantas tareas como en la estrategia anterior, solo recibe la categorización de los *workers*. El proceso de actualizar

la población se puede realizar de varias formas, ya que todos los *workers* comparten la población categorizada. Una posible solución consiste en enviar  $M$  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 estrategia, al dividir la población inicial, es más eficiente cuando esta población inicial es mayor que la población de predicción.
- En su contraparte, en la segunda estrategia, al dividir la población a predecir, tiene un mejor rendimiento con poblaciones de predicción mayores.

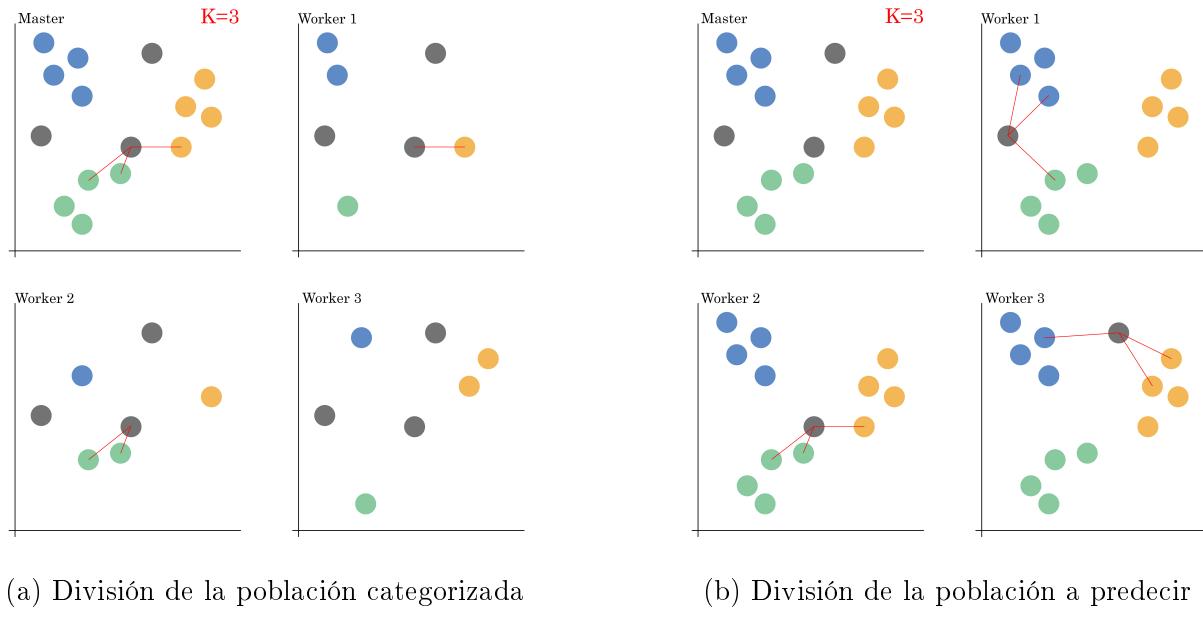


Figura 3.9: MPI - Estrategias para paralelizar el algoritmo KNN

El proceso *master* (el de la esquina superior izquierda en ambas figuras) tiene el dataset completo. En la primera estrategia (gráfico de la izquierda) divide la población categorizada (individuos representados con colores) entre los procesos y mantiene una copia en todos los procesos de la población a predecir (individuos representados en gris). La segunda estrategia

(gráfico de la derecha) tiene una copia de la población categorizada en todos los procesos y una subpoblación de la población a predecir.

En este algoritmo también hay que realizar una búsqueda del mejor valor para  $K$ . Sin embargo, al contrario que en K-Medias, no hace falta repetir varias veces el mismo algoritmo, ya que el proceso es determinista. Es decir, con la misma población, siempre se obtiene la misma predicción. Aunque puede llegar a cambiar dependiendo del orden de categorización de la población a predecir.

No hay que repetir el mismo algoritmo varias veces, pero puede llegar a ser útil variar el número de vecinos (valor de  $K$ ). Las mejoras de esta búsqueda son las mismas que en el algoritmo anterior:

1. Aplicar alguna de las dos implementaciones comentadas anteriormente, con un bucle que varíe la variable  $K$ .
2. Ejecutar en cada proceso *worker* el algoritmo sin mejoras. El *master* se encarga de almacenar los mejores resultados.

### 3.3. Aprendizaje por refuerzo

Los algoritmos de este tipo de aprendizaje actualizan iterativamente las estimaciones de calidad de las acciones permitidas en el entorno de desarrollo.

#### 3.3.1. Q-Learning

El algoritmo Q-Learning, es el más básico del aprendizaje por refuerzo. Las estimaciones de las mejores acciones para cada estado se almacenan en una Q-Table representada como una matriz en la que cada fila es un estado, y las columnas son las acciones disponibles.

Este algoritmo tiene numerosas aplicaciones. Nos centramos en la técnica de minimizar las acciones, para llegar desde una celda origen a un destino. El laberinto tiene un tamaño y semilla variable por parámetros de inicialización. Para lograr su objetivo dispone de acciones

de movimiento en los dos ejes cardinales, norte, sur, este y oeste. No puede atravesar ni situarse en un muro del laberinto, y para que el agente aprenda a moverse por el laberinto y llegar a la meta hay que fijar unas recompensas:

- Si se choca con un muro castigamos al agente con valores altos para que no añada movimientos innecesarios para alcanzar su objetivo.
- 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, para que aprenda a la celda destino.

Con estas recompensas, el agente aprende a llegar a la meta minimizando las acciones ejecutadas. El código que genera los laberintos ha sido implementado por @ChlouisPy en [github](#)<sup>2</sup>

Antes de enfocarnos en las implementaciones MPI, hay que comentar una mejora que se puede aplicar a el algoritmo básico de Q-Learning. Realizar un preprocesado. Modificar la Q-Table, convirtiéndola 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. O eliminar estados innaccesibles como situarse en un muro.

Esta mejora puede reducir el tiempo de cómputo, al no perder tiempo realizando acciones innecesarias para alcanzar su objetivo. Además de reducir la complejidad espacial, al reducir el número de estados. Un desventaja es que añade otras estructura adicional (array bidimensional) para almacenar las acciones para cada estado.

Este preprocesado tiene complejidad cuadrática  $O(4 * N^2) \equiv O(N^2)$ . Recorre toda la matriz, comprobando para cada celda si no es un muro, y en caso afirmativo itera en las cuatro direcciones permitidas para almacenar las acciones disponibles para la celda actual (estado). Con tamaños de laberintos pequeños no hace falta paralelizar el preprocesado, porque no se consigue reducir el tiempo significativamente. Laberintos con más de mil filas y columnas si se consigue reducir el tiempo de cómputo.

En los algoritmos del bloque anterior se necesitaba realizar una búsqueda para encontrar el óptimo general. En este algoritmo conviene realizar otra búsqueda, pero esta vez para encontrar combinaciones de los hiper-parámetros ( $\alpha$ ,  $\gamma$ ,  $\epsilon$ ). Son muy importantes para el desarrollo del agente en el entorno. Una mala configuración de éstos hace que sobre aprenda -o no aprenda- correctamente, generando bucles infinitos. Por este motivo es importante comprobar las diferentes combinaciones de hiper parámetros, y comprobar, cuales funcionan correctamente en el entorno. La búsqueda en laberintos grandes es muy lenta, ya que 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 una cantidad significativa de combinaciones de hiper parámetros.

Para paralelizar esta búsqueda se ejecuta el algoritmo básico con el preprocesado mencionado en cada proceso *worker*. Hay que desarrollar una estrategia para que cada *worker* reciba una combinación de parámetros distinta, y así no haya repeticiones, perdiendo tiempo de cómputo.

El *master* se encarga de repartir combinaciones de hiper-parámetros. Con una precisión previamente inicializada, el *master* aumenta un hiper-parámetro hasta llegar al 100 %. Cuando llega a dicho límite, se reinicia la variable y se aumenta la siguiente. Este proceso continua hasta llegar al 100 % de todos los hiper-parámetros. Cada *worker* se encarga de una combinación recibida. Cuando termina el algoritmo, ya sea por bucle infinito o finalización correcta, envía un mensaje al *master* con la información de finalización y los hiper-parámetros usados.

Si el mensaje de finalización indica “bucle”, el proceso ha terminado para evitar que se convierta en un proceso inactivo, pues dicha combinación no converge hacia el objetivo. Estos bucles se detectan en el entrenamiento o la evaluación. Hay un bucle en el entrenamiento si un episodio tarda más de  $X$  segundos en finalizar. En la evaluación se comprueba teniendo

en cuenta los últimos cuatro estados visitados. Pues avanza y retrocede constantemente.

$Estados[0]==Estados[2] \text{ and } Estados[1]==Estados[3] \text{ and } Estados[0]!=Estados[1]$ :

Una vez realizada una búsqueda de las combinaciones de hiperparámetros eficaces, se pueden emplearse para las siguientes mejoras:

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 (primera mejora), cada proceso controla una zona, y se genera un flujo constante de episodios (iteraciones del algoritmo). 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. La Figura 3.10 muestra como se divide un posible laberinto entre los procesos. Además de mostrar tres episodios con sus respectivos agentes (circulo en rojo). El *master* se encarga de iniciar a los agentes en la celda en verde y cuando sale de su dominio envia al respectivo proceso y genera otro agente.

Para garantizar el correcto funcionamiento, el *master* no puede recibir agentes de los *workers*, en caso contrario no se podría garantizar el flujo de nuevos episodios. Solo puede existir  $M$  agentes en todos los procesos (siendo  $M$  el número de procesos ejecutados), debido a que un proceso solo puede gestionar a lo sumo un agente.

Cada proceso tiene su propio dominio, lo que provoca que la Q-Table se divida entre éstos. Aplicando el preprocesado comentado anteriormente se dividen los dos arrays bidimensionales (acciones y Q-valores).

Aplicando la estrategia realizada en la búsqueda de hiper-parámetros, de ejecutar el algoritmo en varios procesos, se puede obtener una mejora. El *master* recolecta las experiencias de los *workers*, haciendo la media de los valor-Q obtenidos de los procesos, calculando así las mejores acciones para cada estado. Hay que tener en cuenta la posición de inicialización de los agentes en los procesos ejecutados. Si todos los procesos comparten el mismo

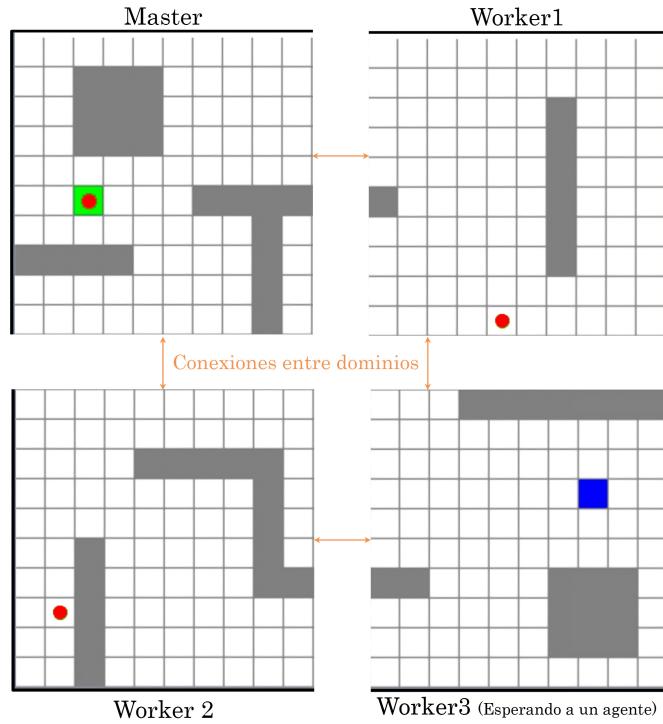


Figura 3.10: Aprendizaje por Refuerzo - División del entorno

punto de salida, los resultados serán parecidos a ejecutar el algoritmo en un solo proceso. Sin embargo al cambiar el punto de origen, los Q-valores obtenidos al realizar las medias de las experiencias varían y se recorre más espacio en menos tiempo. Y para lograr buenos resultados se asegura que al menos un proceso empieza desde el punto origen, de otra forma no se podría garantizar que el agente haya aprendido a alcanzar el destino desde la celda origen.

### 3.3.2. Deep Q-Network

Este algoritmo utiliza redes neuronales para obtener la mejor acción para un determinado estado, eliminando así los problemas que tiene el algoritmo anterior. Con este método podemos abarcar entornos más complejos, y por eso se propone el juego de *Namco Pac-Man*, cuya implementación creamos desde cero para moldear a nuestro gusto la dificultad del entorno, así como facilitar el aprendizaje de la red neuronal. Nos centraremos en obtener el

mayor número de monedas antes de provocar una condición de finalización (ser comido por un fantasma o recoger todas los puntos). Para simplificar el entorno no se desarrollan niveles en la ejecución, al igual de limitar la vida del agente a un único corazón, por lo que si es comido una única vez se termina la ejecución. Antes de profundizar en el algoritmo de IA, explicamos como funciona y hemos realizado la implementación del entorno.

- Acciones disponibles. Como en el algoritmo anterior, son de movimiento. El agente y los fantasmas no pueden atravesar muros.
- Entorno. Laberinto con muros, del cual no se puede escapar.
- Objetos del juego.
  - Pac-Man: el agente que mueve el usuario. Su objetivo es comer todos los puntos.
  - Fantasmas: se mueven siguiendo unos objetivos en el laberinto.
  - Túneles: puntos que se conectan de manera toroidal para no salir del entorno.
  - Puntos (pellets en inglés): son las "monedas" que el agente tiene que recoger.
  - Puntos de energía (powers): si el agente consume uno, durante un periodo de tiempo es invencible y puede comer a los fantasmas.
  - Laberinto: entorno por el cual el agente y fantasmas se mueven.
- Condiciones de finalización. Ganar obteniendo todas las monedas del laberinto o perder si un fantasma come al agente.

Los fantasmas tienen una IA interesante, pues tienen sus propios estados y cada uno tiene unos puntos objetivos que siguen para intentar comer al agente. Cabe recalcar que estos puntos están estratégicamente colocados para que los fantasmas trabajen en conjunto para cerrar huecos y comer al agente. El movimiento para alcanzar los puntos objetivos es simple, cuando se encuentran en una intersección (punto en el mapa con un hueco a la izquierda o derecha con respecto a su dirección actual) eligen la celda que minimice la distancia con respecto al punto objetivo. Los estados de los fantasmas son cuatro siguientes:

- Chase. Cada fantasma sigue unos puntos en movimiento.

- Scatter. Sigue un punto estático fuera del laberinto para dar vueltas en una determinada zona.
- Frightened. El agente puede comerlos, se mueve de manera aleatoria al llegar a una intersección.
- Eaten. Han sido comidos y se encuentran en su casa esperando a salir. (Implementado de forma que espera 3 movimientos del agente para salir)

Los estados iteran con una la secuencia principal *[Scatter, Chase]*. La ejecución empieza con *Scatter* para que los fantasmas, al salir de su casa, se dirijan a sus zonas asignadas. Al ejecutar, el agente, treinta acciones, el estado cambia a *Chase* y se mantiene así sesenta acciones y vuelven a repetir la secuencia. Si el agente come un punto de energía, los fantasmas interrumpen su estado actual para pasar al estado *Frightened* en el que están treinta acciones siendo vulnerables. Si el agente colisiona con los fantasmas en este estado, son comidos pasando al estado *Eaten*. Al finalizar este estado vuelven al inicio de la secuencia principal.

En la estado *Chase* los fantasmas se mueven de la siguiente forma:

- Blinky (Rojo): Persigue directamente al agente.
- Pinky (Rosa): Persigue la celda cuatro posiciones adelantada a donde apunta el agente. Si el agente mira hacia arriba, también añade cuatro celdas hacia la izquierda.
- Inky (Azul): Persigue una celda en concreto que se calcula de la siguiente forma. Primero se calcula una posición como lo hace el fantasma rosa pero en vez de cuatro celdas, se hace con dos. El objetivo se calcula al añadir el vector de distancia de la posición del fantasma rojo a esta posición.
- Clyde (Naranja): Si está a ocho o más celdas de distancia del agente, lo persigue. En caso contrario sigue su objetivo del estado Scatter.

El laberinto (mapa del entorno) se almacena en un fichero de texto, para representar las celdas vacías, muros, puntos o puntos de poder con números enteros (0, 1, 2 y 3 respectivamente).

Al igual que en el algoritmo *Q-Learning* la fase de entrenamiento es crucial, pues modifican los valores de la red neuronal para que el agente tome las mejores decisiones en cada estado, y terminar la ejecución sin perder. El entrenamiento se puede realizar de varias formas.

Si mantenemos el mismo estado inicial, el agente empieza siempre en el mismo punto, y depende mucho de los hiperparámetros, además de la aleatoriedad. El agente empieza a investigar el entorno de manera aleatoria, y es muy probable que los fantasmas alcancen al agente bastante rápido sin explorar en profundidad el entorno. Por eso es mejor añadir varios estados iniciales para que pueda investigar el entorno de manera más eficiente. Hay que tener en cuenta que los estados iniciales tienen que ser puntos accesibles desde el estado inicial original. El agente no puede saltar a otras celdas sin coger los puntos del laberinto. La figura 3.11 muestra un estado accesible y otro innaccesible con la misma posición del agente. El estado accesible ha recogido los puntos del laberinto, así como posicionado correctamente los fantasmas, en su contraparte el estado innaccesible no ha recogido los puntos simulando una acción de salto por parte del agente. Si entrenamos con puntos aleatorios sin cambiar el estado del entorno, este entrenamiento no habrá surtido efecto, pues son estados que el agente no va a alcanzar nunca.

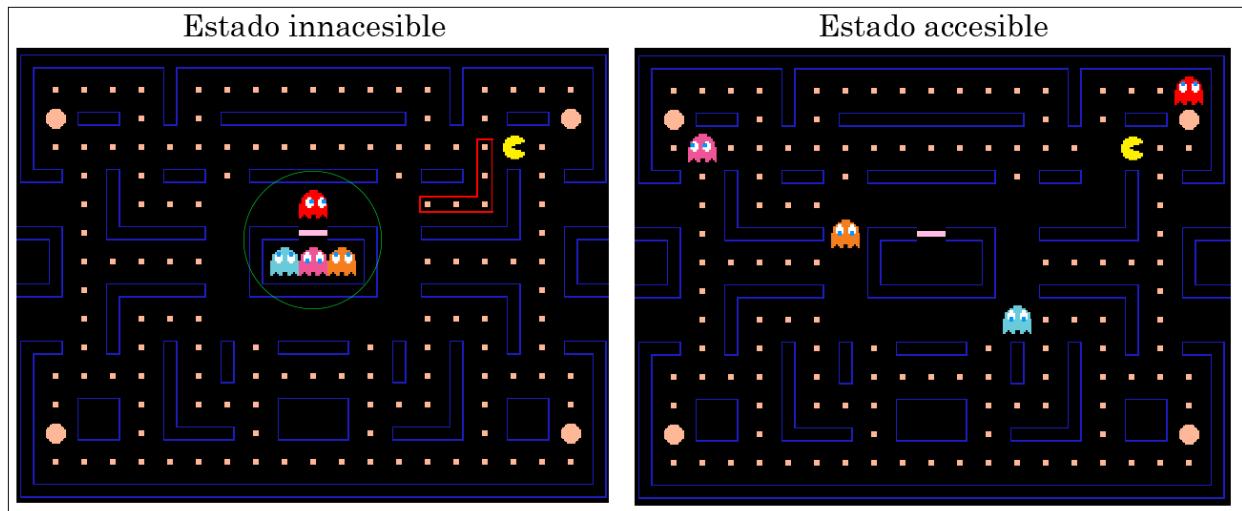


Figura 3.11: DQN - Estados del entorno

Como se comentó anteriormente, este algoritmo usa redes neuronales para aprender a ejecutar la mejor acción para un estado dado. Se van aplicar las mejoras que comentan en la sección ?? de redes neuronales. En este caso, no se pueden aplicar las mejoras del algoritmo anterior, pues no es una matriz que se pueda dividir el trabajo, si no una red neuronal cuyos pesos varían al ejecutar acciones en estados.

## 3.4. Algoritmos Evolutivos

Los algoritmos evolutivos son sencillos de paralelizar. Trabajan con poblaciones de individuos que evolucionan a lo largo de las generaciones. Los individuos se someten a operaciones para producir nuevas generaciones. Estas operaciones son independientes, pues se puede dividir el cálculo entre varios procesos. Y son las siguientes:

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 al gran elevado número de bits que enviar.
  - Reales. Al igual que los binarios son fáciles de inicializar, pero esta vez son más portables, al usar la *base 10* como representación de los números, en vez del sistema binario (0's y 1's).
  - Árboles. Más lentos para inicializar y difíciles de tratar. Se usan punteros y aumenta la complejidad al gestionar los punteros.
2. **Evaluación.** Este es el método que más tiempo de ejecución puede llegar a consumir. Varía dependiendo del tipo de individuo del problema. Como su nombre indica evalúa a todos los individuos dependiendo de una función de fitness, puede ser desde una fórmula matemática hasta una ejecución de un algoritmo en un entorno.
3. **Selección.** Se seleccionan a los individuos. La aleatoriedad predomina en este método, y dependiendo de la estrategia escogida se puede dar más o menos probabilidad a los más aptos.
4. **Cruce.** Todos los individuos se cruzan, y con una cierta probabilidad se eligen a los hijos de estos cruces. Si no se cumple dicha probabilidad no se cruzan los individuos y se dejan como están. Normalmente tendrán un mayor coste temporal que el método anterior.

**5. Mutación.** Igual que el cruce tiene una probabilidad para mutar. Normalmente es un poco más veloz que el cruce, debido a las estrategias implementadas y que la probabilidad de mutación suele ser menor a la de cruce, provocando una menor tasa de ejecución de la estrategia.

En este trabajo se desarrollan los siguientes problemas a optimizar para los tres diferentes individuos implementados.

1. Los individuos binarios, tienen un intervalo y una precisión como variable de inicialización. Queremos calcular el valor máximo o mínimo para ciertas funciones matemáticas. 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 ello hay que convertir la información de binario a real.

Para estos individuos, el algoritmo se ejecuta bastante rápido, además de alcanzar el máximo global con menos generaciones que los otros dos individuos. La función de evaluación es lineal  $O(N)$ , debido a la conversión del número binario. Reducir su tiempo de ejecución es desafiante, por el poco coste temporal y el tiempo necesario para enviar/recibir los individuos.

2. Los individuos reales, se enfrentan a un problema de mayor complejidad, como es la minimización del retraso total obtenible de una serie de aviones en un aeropuerto. El número vuelos y pistas es variable, y cada vuelo tiene asignado la hora de aterrizaje para cada pista. Hay un tiempo mínimo de separación entre vuelos que aterrizan en cada pista, pues hay que garantizar la seguridad de los pasajeros y la integridad de los aviones. Se puede resolver con vuelta atrás pero tiene un coste exponencial  $O(2^N)$ . El valor *fitness* tiene coste  $O(\text{NumAviones} * \text{NumPistas}) \equiv O(N^2)$ . La figura 3.12 muestra como se calcula el valor *fitness* de un individuo.

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

```

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

```

Figura 3.12: PEV - Función de evaluación para los individuos reales

3. Los individuos árbol, han de encontrar una sucesión de acciones en un entorno para maximizar las celdas visitadas en una matriz. Poniendo en contexto, el agente de este problema se encarga de podar un jardín de tamaño ( $N \times M$ ) con unos operadores determinados. Estos operadores pueden ser terminales o nodos hoja (nodos del árbol sin hijos) como avanzar podando el césped o girar a la izquierda (ambos terminales devuelven  $(0, 0)$ ) y constante, que no altera al agente pero devuelve sus valores ( $X, Y$ ) para los operadores funciones. Los operadores funciones, expresiones o nodos intermedios (nodos del árbol con al menos un hijo) como Salta( $E$ ), operación que provoca que el agente salte y pida la posición destino. O los operadores Progn( $E_1, E_2$ ) y Suma( $E_1, E_2$ ) que aunque no modifiquen al agente aumentan el tamaño del árbol (devuelven la posición de su hijo derecho ( $E_2$ ) y la suma de sus hijos ( $E_1$  y  $E_2$ ) respectivamente).

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 estrategias MPI se pueden configurar para cada tipo de individuo:

1. Dividir la población en subpoblaciones.
2. Modelo de islas.

### 3. PipeLine.

La primera estrategia, que consiste en dividir la población entre los 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 que los *workers* procesan los datos recibidos, 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 (segunda estrategia), se aplica la estrategia que se ha desarrollado anteriormente para otros algoritmos. Cada proceso ejecuta el algoritmo básico, dividiendo la población general en los procesos ejecutados. Con este modelo se reduce la comunicación entre los procesos, pues al contrario que la estrategia anterior, no se reserva funcionalidades para diferentes procesos, si no que cada proceso se encarga de ejecutar todas las funciones para que su subpoblación evolucione correctamente. Cada cierto tiempo se reinicia la población con los mejores individuos generales (de todos los procesos). Los tipos de comunicación son los siguientes:

- Estrella. El *master* se posiciona en el centro y solo hay comunicaciones *master-worker* para almacenar los mejores individuos de cada subpoblación.
- En red. No hay proceso *master*, todos los procesos ejecutan el algoritmo. Al finalizar una iteración todos los procesos se comunican entre sí, mandando los mejores individuos para el reinicio de la población.
- En anillo. Igual que el anterior, pero esta vez no se comunican todos los procesos. La comunicación, como su nombre indica se hace con topología circular, cada proceso se comunica con su predecesor y sucesor.

Segmentar el algoritmo entre los procesos (tercera estrategia, representada en la figura 3.13), provoca un flujo constante de generaciones. Como hay cinco métodos principales se necesitan, al menos, cinco procesos, incluyendo al *master*, que se encarga de inicializar y evaluar las cuatro distintas poblaciones que se van a ejecutar al mismo tiempo. Al principio, todos los procesos *workers* están en espera de recibir una población para ejecutar sus operaciones. El primer *worker* se encarga de la selección y se despierta en la segunda iteración al recibir la primera población inicializada por parte del *master*. Una vez ejecutado las cuatro selecciones de las poblaciones que el *master* le envía, empieza a recibir las poblaciones del último *worker*. Los otros tres *workers* se encargan del cruce, mutación y el último de la evaluación, despertándose en la tercera, cuarta y quinta iteración respectivamente por parte de su *worker* predecesor. El último *worker* finaliza una generación al enviar la evaluación de la población mutada al *worker 1*, encargado de realizar la selección. No hay conflicto con el *master*, ya que, como se mencionó anteriormente en este párrafo, este crea cuatro poblaciones (terminando en la iteración cuatro, mientras que la primera evaluación del último *worker* se finaliza en la iteración cinco), y al finalizar su trabajo pasa a un estado de recepción de los mejores individuos. Si se ejecutan 100 generaciones, esta estrategia evoluciona cuatro poblaciones distintas, en 104 generaciones en total, de las cuales 96 se ejecutan las cuatro poblaciones simultáneamente.

El tiempo de ejecución de esta estrategia es proporcional al número de generaciones ejecutadas por el proceso que más tiempo tarda en ejecutarse. Por lo que se puede reducir el tiempo de ejecución si comprobamos que métodos tardan más, añadiendo más procesos para reducir la carga de trabajo. Por ejemplo en los individuos reales y árboles el método que más tiempo consume es la evaluación de individuos, por lo que conviene añadir más procesos para reducir así el tiempo de ejecución.

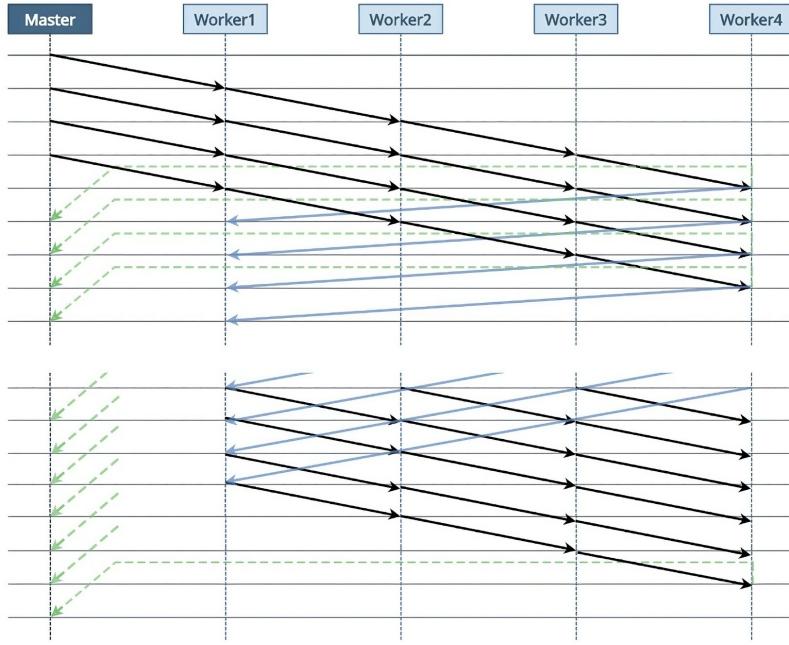


Figura 3.13: PEV - Tercera estrategia: PipeLine

### 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. Para comenzar, vamos a crear una red neuronal que aprenda a predecir el Índice de Masa Corporal (IMC) de una persona. Cada individuo está formado de dos variables, la altura en centímetros y el peso en kilogramos. Para moldear la red neuronal a los individuos, la capa de entrada se estructura con dos neuronas, una para cada variable, y la salida es el IMC, por lo que la capa de salida es una única neurona. 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 algunos de los algoritmos anteriores, necesitamos encontrar la mejor configura-

ción de los hiper-parámetros. En las redes neuronales solo hay uno. La tasa de aprendizaje, controla la magnitud de los ajustes a realizar en los pesos de las neuronas durante el proceso de entrenamiento. Específicamente, determina cuánto deben cambiar los pesos en respuesta al error cometido a predecir un individuo.

Por ello, diseñamos una estrategia para encontrar la mejor tasa de aprendizaje para una red neuronal en concreto. El proceso *master* envía intervalos de tasas de aprendizaje a todos los procesos *workers* ejecutados, para que estos ejecuten el algoritmo y envíen el sumatorio de errores obtenidos en la predicción. La inicialización de los pesos, normalmente es aleatoria, pero para hacer más igualitario el cálculo de los errores, todos los procesos inicializan la red neuronal con los mismos pesos.

Las estrategias MPI realizadas son las siguientes: 1. PipeLine. Como en el algoritmo anterior, pero esta vez con un flujo de mensajes bidireccional.

2. Dividir el trabajo entre los 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 de los individuos, 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, la figura 3.14 muestra dicho flujo y el trabajo de los procesos es el siguiente:

1. El *master* envía un número proporcional de individuos a los procesos en ejecución.

Luego, antes de enviar otro individuo, entra en un bucle en el cual recibe el error de un individuo ya enviado, actualizando sus neuronas, y envía otro individuo. Para finalizar recibe el mismo número de errores (actualizando las neuronas) que individuos envió al principio.

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 lo propagan enviando lo a la capa anterior.
- Reciben de la capa anterior: los nuevos individuos, procesan y propagan hacia adelante.

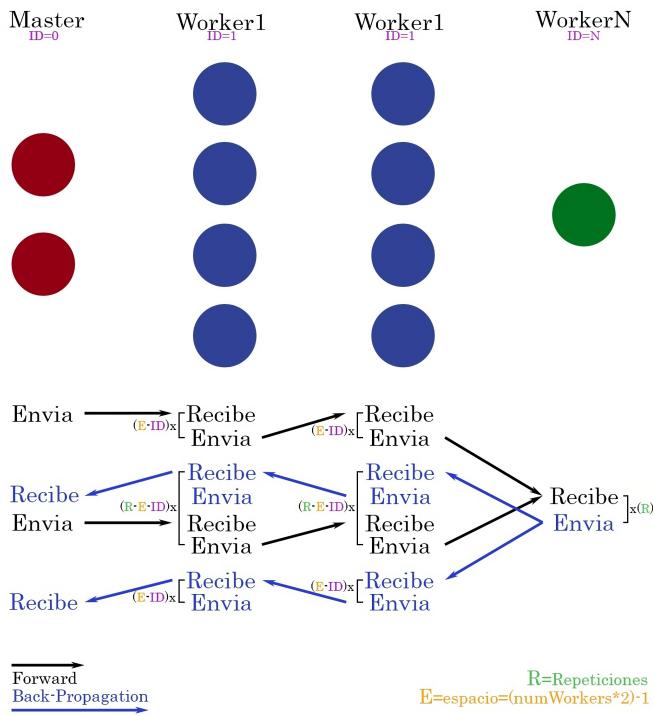


Figura 3.14: MPI - PipeLine Rede Neuronal

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 (segunda estrategia) no parece ser beneficioso para el correcto aprendizaje de la red. Sin embargo, en redes neuronales hay un proceso llamado *fine tuning*<sup>13</sup> que consiste en entrenar una red neuronal, con unos pesos ya calculados. Basándonos en esta técnica, podemos implementar una mejora en la cual

dividamos 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 mayor será -a priori- el rendimiento.

Las neuronas varían sus pesos para adaptarse a las variables recibidas. Para predecir un individuo, todas las neuronas trabajan en conjunto. Lo que supone que una neurona perteneciente a una red de menor tamaño tendrá mas relevancia en comparación con una neurona en una red de mayor escala. Esto plantea un interrogante: ¿es posible paralelizar el trabajo de una red neuronal de gran escala y conservar o reducir el porcentaje de error al predecir individuos? Para ponerlo a prueba hay que probar varias estrategias de agrupaciones en la población que se va a utilizar para entrenar la red neuronal, además de tener en cuenta la inicialización de los pesos.

## 1. Misma población en todos los procesos.

- Si cada proceso tiene la misma población y los mismos pesos en la red, si reduce el tiempo de ejecución, pero no mejora la predicción. Sería como ejecutar el algoritmo sin paralelizar, pero con menos iteraciones, pues se ejecutan en los procesos la misma ejecución y la media no varía con respecto a los resultados obtenidos.
- Inicializando las distintas redes neuronales con pesos diferentes, puede predecir correctamente para este problema en particular pero no tener unos buenos resultados de manera global o viceversa. Además, depende de la aleatoriedad, pues el porcentaje de errores en una ejecución puede variar bastante con respecto a otra.

## 2. Diferentes poblaciones para cada proceso.

Esta estrategia suena mejor que la anterior. Al no haber intersección de poblaciones en los procesos ejecutados, los valores de los pesos se modificarán de diferente forma y puede que al hacer la media la red se estructure de forma que se obtenga un correcto funcionamiento. La inicialización de los pesos no provoca una gran diferencia,

al contrario que mantener la misma población para todos los procesos. Pero conviene probar ambas inicializaciones.

# Capítulo 4

## Estudio empírico

Después de diseñar e implementar las estrategias descritas en la Sección 3, 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 de propósito general. Seguidamente ejecutar las mejores implementaciones en un sistema distribuido con un número elevado de núcleos de CPU.

### 4.1. Entornos de ejecución

Para ejecutar las pruebas y comprobar el funcionamiento de las implementaciones, primero se ejecutan en un ordenador de propósito general. Este sistema computacional tiene las siguientes especificaciones:

- Procesador (CPU): *AMD Ryzen 7*, con 8 núcleos y 16 hilos, es capaz de defenderse con cargas de trabajo y paralelismo de alto nivel.
- Memoria (RAM): 32 GB de *RAM DDR4*, permitiendo una amplia capacidad para manejar grandes volúmenes de datos en memoria. Característica fundamental para ejecutar algoritmos de IA que demandan una cantidad elevada de recursos.
- Tarjeta Gráfica (GPU): *NVIDIA GeForce RTX 3070* con arquitectura *Ampere*<sup>16</sup>, que cuenta con 5888 núcleos CUDA y 8 GB de memoria GDDR6. La arquitectura *Ampere*

es sucesora de la arquitectura *Turing* lanzada en 2020. Fue diseñada para brindar un mejor rendimiento, especialmente en aplicaciones de computación paralela.

- Placa Base: *X570 Gaming*. Soporta las tecnologías de conectividad de alta velocidad, garantizando el rendimiento y estabilidad del sistema en condiciones de carga elevada.

El sistema altamente distribuido, consta de tres ordenadores. Uno que funciona de Front-End y dos como nodos de cómputo, en total 128 núcleos y 256GB de RAM. La figura 4.1 muestra la estructura del cluster.

El Front-End, realiza la conexión remota con los otros dos ordenadores, situados en la Facultad de Informática de la Universidad Complutense de Madrid. Este ordenador no participa en el cómputo, solo mantiene los *scripts* (tipo de fichero de texto con el código escrito en un lenguaje de programación, en este caso Python) y conecta los dos nodos de cómputo para que realicen las tareas. Durante las pruebas, cada proceso tiene un núcleo dedicado, por lo que el rendimiento de cada proceso no se ve afectado por otros procesos del sistema. Esto permite mayor precisión para evaluar el rendimiento del sistema, y las pruebas ejecutadas no compiten por los recursos de la CPU.

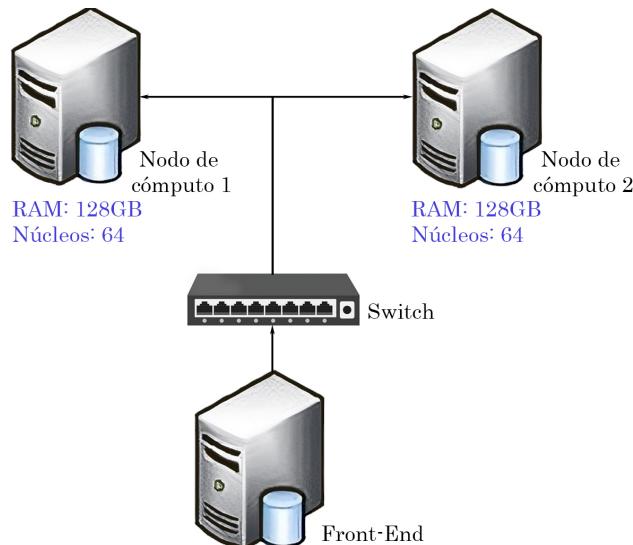


Figura 4.1: MPI - Matriz

En un ordenador de propósito general esto no ocurre, pues están diseñados para ejecutar múltiples procesos simultáneamente, cada uno pudiendo requerir acceso a los mismos recursos del sistema, como la memoria RAM, disco duro o la CPU.

Para realizar las pruebas se usan 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()*.

## 4.2. Programas sencillos

Primero realizamos el estudio de los programas básicos descritos en la Sección 3.1, ordenación de arrays y multiplicación de matrices.

### 4.2.1. Ordenaciones

Las pruebas realizadas para estos algoritmos se realizan para el peor de los casos, es decir, un array de enteros sin repeticiones ordenado de forma decreciente. Cada algoritmo tiene que hacer el mayor número de comparaciones posibles para ordenar el array de manera creciente. El resultado de cada prueba (tiempo de ejecución en segundos) es almacenado en un fichero de texto y aumentado el tamaño del array para ejecutar la siguiente prueba, así hasta llegar a 100.000 elementos.

#### 4.2.1.1. Algoritmos de complejidad cuadrática

Debido al coste cuadrático de estos algoritmos, el incremento entre pruebas del tamaño de los arrays se obtiene de la siguiente forma:

- [20 – 1,000) → 20 elementos.
- [1,000 – 10,000) → 250 elementos.
- [10,000 – 100,000) → 1.000 elementos.

*SelectionSort* es fácilmente paralelizable, pues para cada elemento se comprueba cuantos elementos en el array son mayores. Las estrategias implementadas utilizan el modelo *Master-Worker*. El *master* envía a cada proceso *worker* un elemento del array para que hagan las

comparaciones y devuelvan el indice del elemento, junto con el número de elementos mayores que el recibido, y así el *master* se encarga de ordenar el array y enviar elementos sin procesar. La figura 4.2 muestra los tiempos de ejecución. En rojo el algoritmo sin mejora, y en verde y negro la dos estrategias ejecutadas con cinco procesos. Se puede apreciar una considerable reducción del tiempo de ejecución.

Al comparar las dos estrategias MPI, se obtiene que la primera estrategia es un 34 % más rápida que la segunda. Sin embargo, la segunda estrategia tiene una menor complejidad espacial, mostrando en la gráfica de la derecha, en negro, que la memoria no varía al aumentar los procesos ejecutados.

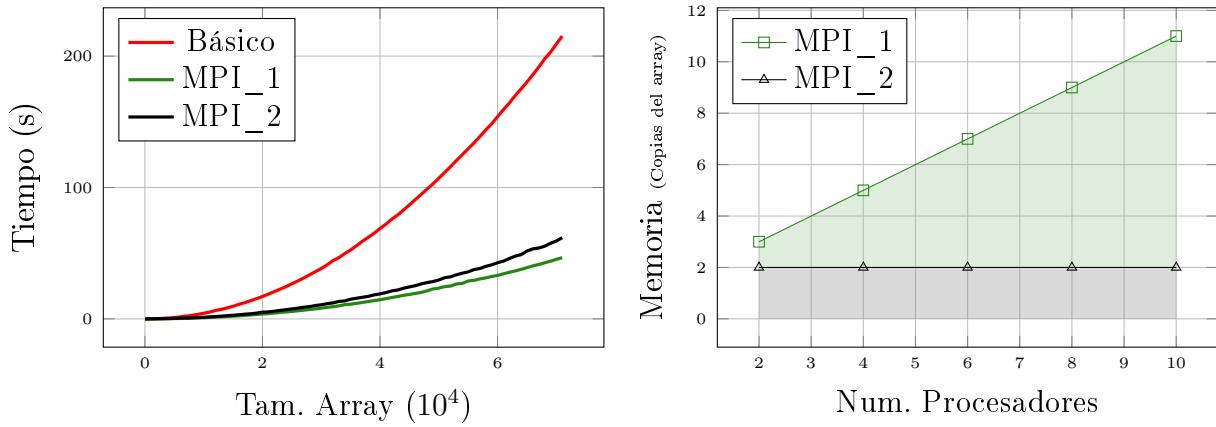


Figura 4.2: SequentialSort - Tiempos de ejecución de las estrategias MPI y su uso de Memoria

Una vez comparado las estrategias MPI con el algoritmo secuencial, podemos comprobar el rendimiento frente a los algoritmos famosos. La figura 4.3 muestra que *SelectionSort* (la función de color negro) es la ordenación que mejores resultados obtiene, y *BubbleSort* (la roja) la que peores. *SelectionSort* es, aproximadamente, 3.5 veces más rápida al ordenar 70.000 elementos. La ordenación *SequentialSort* sin paralelizar, es incluso más rápida que dos de las más famosas. Esto es debido a la simpleza de las operaciones aplicadas en la ordenación, pues solo hace  $N^2$  comparaciones. En *BubbleSort* e *InsertionSort*, además de realizar comparaciones, modifican las posiciones de los elementos en el array, aumentando el tiempo de ejecución.

La estrategia MPI de *SequentialSort* que menos tiempo consume (la primera) no obtiene mejores resultados que la mejor ordenación sin mejoras (*SelectionSort*) hasta llegar a los cuatro procesadores ejecutados, siendo un 20 % más veloz. Para mostrar de forma más clara la diferencia de tiempos entre estas dos ordenaciones, se muestra la ejecución de la primera estrategia con cinco procesadores, obteniendo un 50 % de mejora.

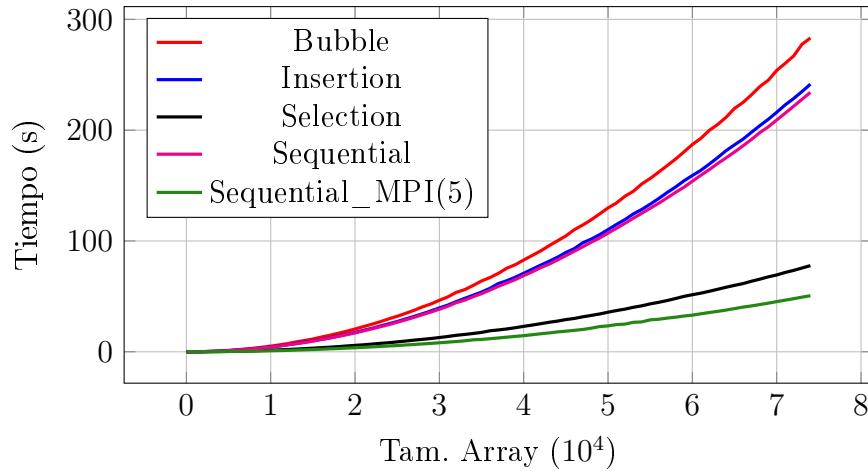


Figura 4.3: Tiempo de ejecución de los algoritmos de ordenación cuadráticos

#### 4.2.1.2. Algoritmo *MergeSort*

Este algoritmo no tiene un coste tan elevado como los anteriores. La complejidad es logarítmica  $O(N \log N)$  lo que provoca que se pueda aumentar el tamaño del array a ordenar. Para la estrategia implementada, no se aplica el modelo *Master-Worker*, todos los procesos creados trabajan de manera igualitaria. Como se dijo en la sección 3.1, esta estrategia usa potencias de dos procesos para ordenar el array. En cada iteración los procesos se comunican con el más cercano, uno le envía su subarray ordenado y termina su ejecución (el de mayor *id* de cada pareja), mientras que el otro reordena los dos subarrays y continua a la siguiente iteración.

En esta ocasión, la prueba realizada consiste en ordenar de manera creciente cuatro arrays de enteros inicializados de manera decreciente (peor de los casos), empezando con 25.000 elementos e incrementar esa misma cantidad entre pruebas. Pese a tener solo ocho núcleos en

el ordenador de propósito general, se comprueba el rendimiento de la estrategia con 4, 8, 16 y 32 procesos. La figura 4.4 muestra los resultados obtenidos en forma de histograma. Como la estrategia aplica ordenaciones cuadráticas en los subarrays al comienzo del algoritmo, no se obtienen buenos resultados con pocos procesos, debido al elevado tamaño del array a ordenar. Con dos procesos ejecutándose no reduce el tiempo de ejecución, lo duplica. El cómputo es equivalente a aplicar una ordenación cuadrática con la mitad del array a ordenar. No obstante, se puede apreciar una notoria reducción del tiempo de ejecución apartir de 16 procesos, llegando a tener un speedup aproximado de 15.5. Es cierto que se podrían aplicar otras ordenaciones con menor complejidad para reducir más el tiempo, pero así se demuestra que en la computación de alto rendimiento se pueden obtener buenos resultados con estrategias no tan efectivas pero bien paralelizadas.

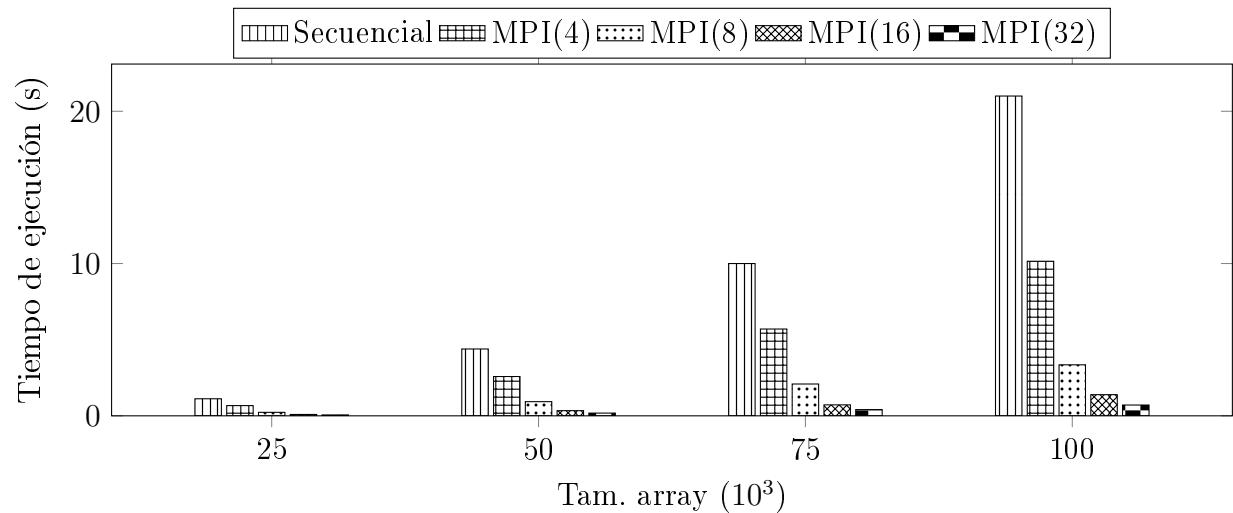


Figura 4.4: Tiempo de ejecución del algoritmo *MergeSort* en ordenador de propósito general

La memoria está optimizada, puesto que el array esta dividido entre los procesos. Al terminar un proceso con la sincronización en mariposa comentada en la sección 3.1, se termina la ejecución del proceso liberando memoria una vez ha enviado al proceso correspondiente su subarray ordenado.

Habiendo obtenido buenos resultados con muchos procesos, pasamos a comentar las pruebas realizadas en el sistema altamente distribuido.

El algoritmo secuencial de *MergeSort* tarda unos *20.16* segundos en ordenar de manera creciente, un array de *100.000* elementos ordenados de manera decreciente (el peor de los casos). Por esos se realiza una prueba para saber cuento tiempo tarda la estrategia en ordenar un array con un millón de elementos. Las pruebas comienzan con *100.000* elementos, incrementando nueve veces su tamaño hasta llegar al millón de elementos. Estas pruebas se realizan con *16, 32, 64* y *128* procesos. La figura 4.5 muestra los resultados obtenidos.

De manera secuencial, sin mejoras, el algoritmo tarda, *20.16* segundos en ordenar *100.000* elementos, mientras que con *128* procesos tarda *0.16* segundos, obteniendo un *speedup* de *125*.

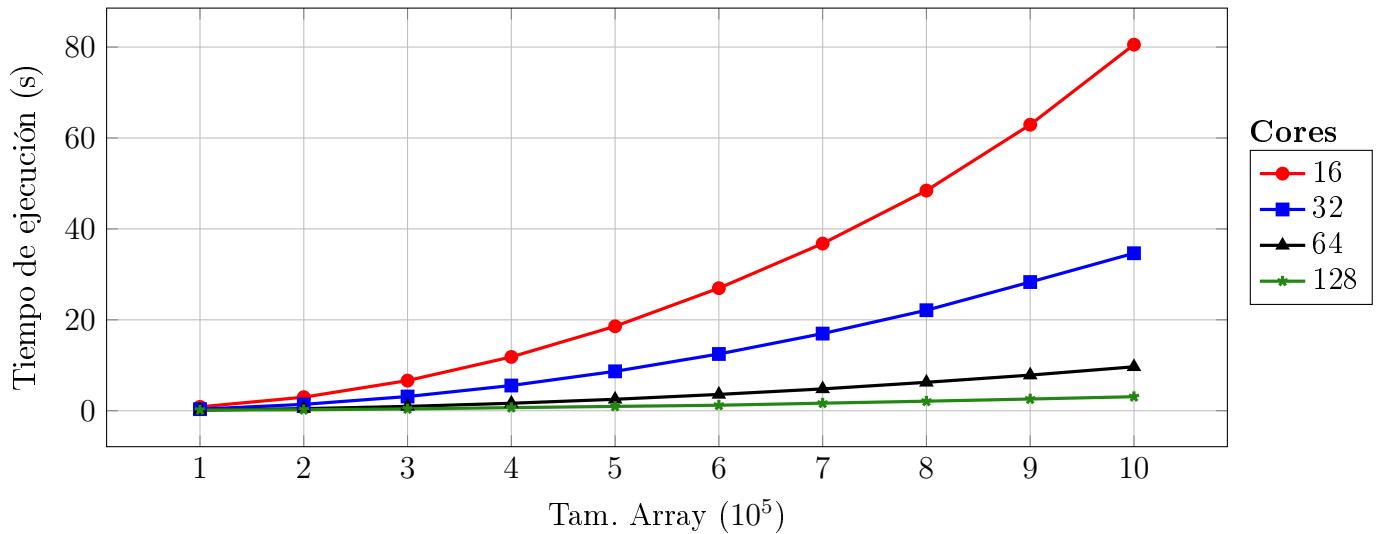


Figura 4.5: Tiempo de ejecución del algoritmo *MergeSort* en Cluster

Al incrementar del tamaño del array, todas las pruebas funcionan correctamente. Ejecutar el algoritmo secuencial con un millón de elementos tardaría mucho. Sin embargo, se puede hacer una estimación como muestra la tabla 4.1, en la cual, aplicando factores de conversión, se obtiene que  $X = 241.67$  segundos, y llegando a tener un *speedup* de *78* al aplicar *128* procesos.

Tamaño (N)	Funcion = N*Log2(N)	Tiempo (s)
100000	1.66e06	20.16s
1000000	1.99e07	X

Cuadro 4.1: Estimación del tiempo de ejecución de *MergeSort* con un millón de elementos

#### 4.2.2. Multiplicación de matrices

Para este algoritmo, al contrario que los anteriores, no hay caso peor, pues siempre se ejecutan el mismo numero de multiplicaciones para cualquier combinación de una matriz.

Las pruebas se realizan con una única matriz cuadrada de tamaño  $2000$ . Se genera de manera aleatoria con valores que oscilan entre  $[1-9]$ , y es almacenada para usar en cada prueba. Inicialmente, la matriz empieza con diez filas y columnas, al finalizar una prueba, se almacena el tiempo de ejecución y se incrementa el tamaño en diez, así hasta llegar a  $1750$  filas y columnas.

La distribución de tareas de los procesos se realiza mediante el modelo *Master-Worker*. El proceso *master* se encarga de enviar una matriz completa ( $B$ ), y enviar filas de la matriz ( $A$ ) a los *workers* para que realicen el cálculo de dicha fila. Al finalizar el procesado envían de vuelta la fila al *master* y esperan otra fila sin procesar, para poder, al final de la ejecución formar, entre todos, la matriz final ( $C$ ). ( $A * B = C$ )

Cada proceso necesita, al menos, una copia de una matriz completa. El uso de memoria es proporcional al número de procesos ejecutados. No hace falta tener las dos matrices porque el *master* se encarga de repartir filas para que vayan realizando el cálculo.

Seguidamente se ejecutan los programas de multiplicación de matrices en el ordenador de propósito general. La figura 4.6 muestra la ejecución del algoritmo secuencial, además de la estrategia implementada en la sección 3.1, con 2, 4 y 6 procesos *workers*. Se puede apreciar, la reducción, aplicando la estrategia con los diferentes números de procesos, llegando a obtener un *speedup* de 8.4 al ejecutar el algoritmo con seis *workers* en una multiplicación de  $1750 \times 1750$ .

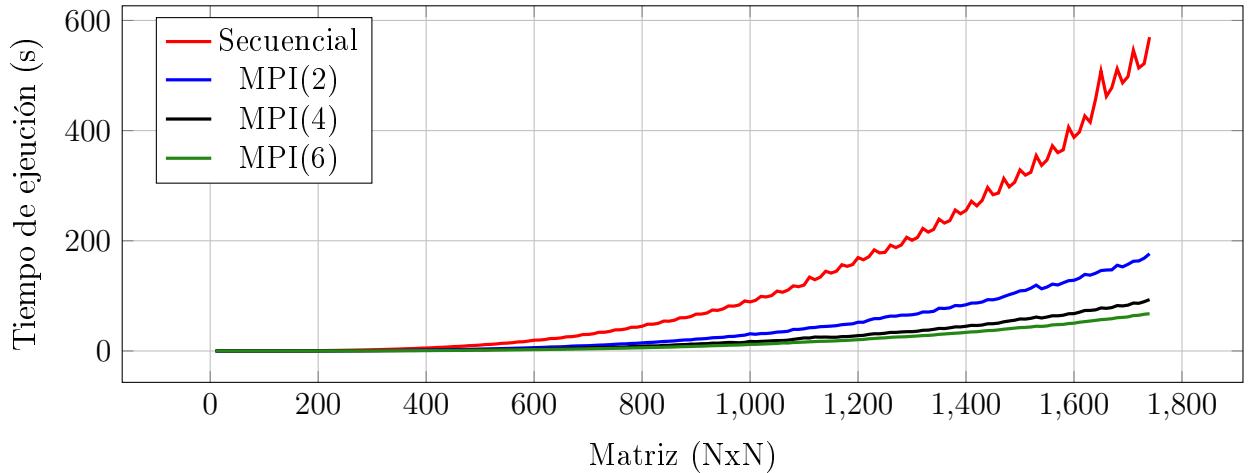


Figura 4.6: Tiempo de ejecución de multiplicación de matrices en ordenador de propósito general

Las oscilaciones en la gráfica se deben al incremento de diez elementos entre pruebas en un algoritmo con complejidad cúbica  $O(N^3)$ . Estas oscilaciones son mas pronunciadas en la multiplicación sin paralelizar, pues el tiempo de ejecución es mayor.

En el cluster, al poder ejecutar muchos procesos, se puede aumentar el tamaño de las matrices de las pruebas a ejecutar. Utilizamos 16, 32, 64 y 128 procesos para medir el tiempo que tarda el algoritmo con la misma estrategia que la prueba anterior. En este caso comenzando con 500 elementos por fila, e incrementando ese mismo tamaño hasta llegar a una matriz de 5000 filas y columnas.

La figura 4.7 muestra los tiempos de ejecución con los procesos y tamaños comentados en el párrafo anterior. No se puede apreciar, pero con una matriz de 1000 filas, ejecutar 128 procesos reduce el tiempo de ejecución hasta unos 1.06 segundos, logrando un *speedup* de 84 con respecto a los 89.1 segundos del cálculo sin paralelizar. La comunicación entre procesos no es tan óptima como en la estrategia de *MergeSort*, debido a que en esta implementación aplicamos el modelo *Master-Worker* y es posible que un proceso *worker*, al finalizar de procesar una fila, tenga que esperar a que el *master* esté libre (puede estar recibiendo y colocando otros datos recibidos de otro proceso) para recibir nuevos datos que procesar. En cada prueba, se pierden  $(N/M)*T$  segundos en la comunicación entre procesos. Siendo  $N$  el

número de filas de la matriz,  $M$  el número de *workers* y  $T$  el tiempo de comunicación.

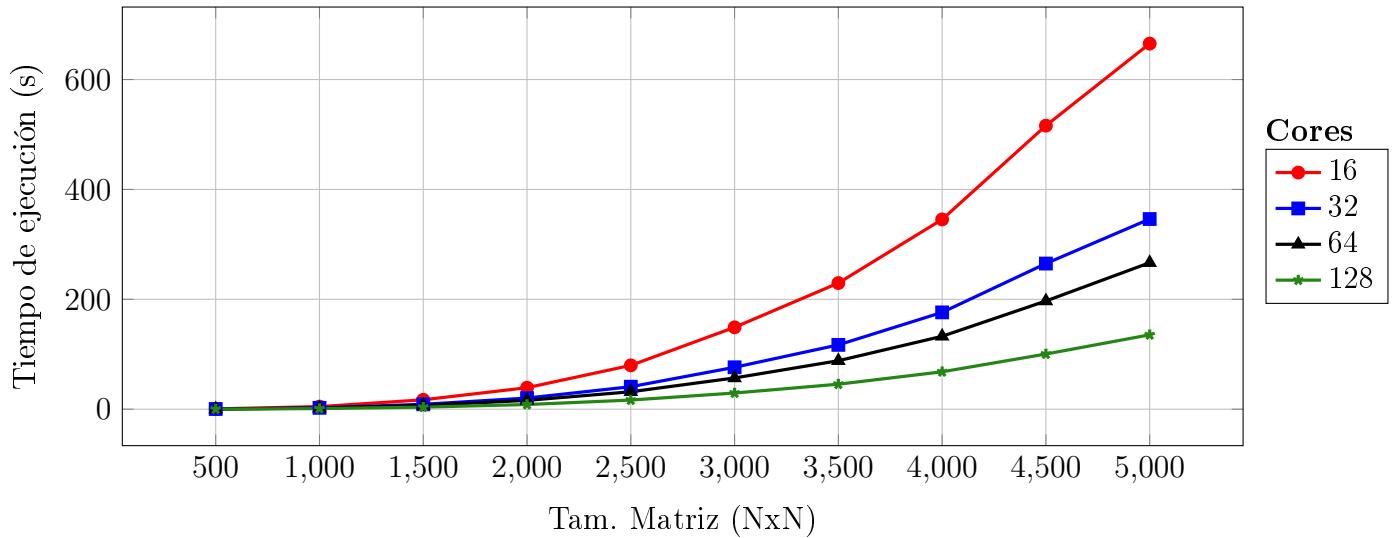


Figura 4.7: Tiempo de ejecución de multiplicación de matrices en Cluster

### 4.3. Algoritmos de Agrupación

Las poblaciones que se utilizan en las pruebas de esta sección se han almacenado en un fichero para usar la misma población generada de manera aleatoria, con dos variables de entrada, es decir, individuos es un plano bidimensional. Los valores son delimitados en el siguiente intervalo  $[-10, 10]$ , ya que los valores no influyen en el cálculo de la distancia. Los tamaños de las poblaciones se incrementan dependiendo de la complejidad temporal de cada algoritmo. En sus secciones se especifica en profundidad.

Los tres algoritmos de esta sección, se basan en el modelos *Master-Worker*. El *master* divide los datos de entrada para que los *workers* hagan el procesado. El *master* en cada algoritmo tiene las siguientes funciones:

- Jerarquico Aglomerativo. En cada iteración se encarga de gestionar que procesos tienen que eliminar o actualizar las filas y columnas de la matriz. El *master* no tiene una copia de la matriz, así mejorando el uso de memoria al estar dividida entre los procesos *workers*.

- KMedias. Su función principal es comprobar la condición de finalización. En cada iteración recibe las asignaciones de los datos procesados de los *workers*, y si esta asignación no varía se finaliza la ejecución.
- K-Veinos más Cercanos (KNN). En las dos estrategias se encarga, de diferente forma, de actualizar las poblaciones de los *workers* para que haya más precisión a la hora de categorizar nuevos individuos.

#### 4.3.1. Jerárquico Aglomerativo

De los tres algoritmos de agrupación, este es el más lento. Su bucle principal itera  $N \cdot C$  veces, siendo  $N$  el número de individuos de la población y  $C$  el número de clusters deseados. En cada iteración, recorre una matriz entera para juntar dos clusters, los que se encuentren a menor distancia.

En este algoritmo, el cálculo de las distancias entre clusters es muy importante. Cada tipo genera diferentes agrupaciones, además de tener diferentes complejidades temporales. La distancia por *centroides* es la que menos tiempo consume, siendo constante, al solo importar los centros de los clusters. Mientras que *enlace simple* y *completo* tienen una complejidad cuadrática  $O(N^2)$ , recorriendo todos los individuos de los ambos clusters para calcular la distancia. Además hay que añadir el cálculo de la distancia entre individuos, que puede ser *Manhattan* o Euclídea, siendo esta última un poco más tardía que la primera.

Para mostrar la importancia de las distancias entre clusters y entre individuos, se realiza un estudio de los algoritmos sin aplicar ninguna estrategia computo de alto rendimiento (HPC). Con la población almacenada, se ejecutan las diferentes combinaciones de distancias (entre cluster e individuo), generando 4 tipos, pues enlace *simple* y *completo* solo varía almacenar la menor o menor distancia entre clusters. Empezando con veinte individuos, y aumentando ese mismo tamaño hasta llegar a mil. A partir de este punto, es mejor incrementar entre prueba 250 individuos, pues ya empieza a tardar bastante. La figura 4.8 muestra dicho estudio.

Al principio no hay tanta diferencia, pero conforme aumenta el tamaño de la población, los tiempos de ejecución empiezan a distinguirse. Como muestra el círculo rojo, la distancia entre clusters por *centroide* no varía mucho usar una distancia *euclídea* o *manhattan* entre individuos. No obstante, aplicando enlace *simple* (o *completo*) es mejor usar distancia *manhattan*. El cálculo de distancias entre individuos no usa potencias o raíces cuadradas, operaciones con un mayor coste que restas en valor absoluto. Cabe recalcar la diferencia de las distancias entre clusters por *centroide* y por enlace *simple* y *completo*. Al aumentar la población a categorizar aumentan los tamaños de los clusters, sobrecargando el cálculo de nuevas distancias.

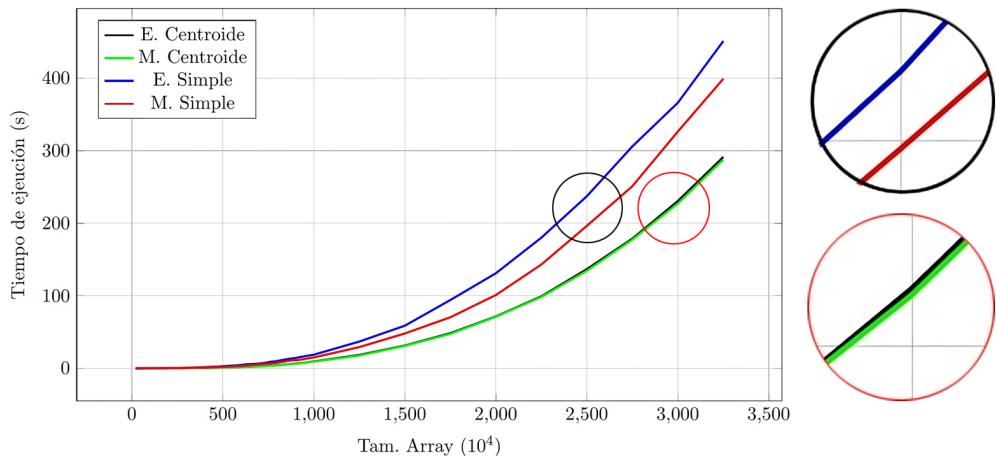


Figura 4.8: Tiempo de ejecución de las combinaciones de distancias en el algoritmo secuencial Jerárquico Aglomerativo

Una vez comprobado los tiempos de ejecución del algoritmo sin paralelizar, podemos pasar a las pruebas de las estrategias comentadas en la sección 3.2.1. Primero estudiamos la distancia entre cluster con menor tiempo de ejecución, por *centroide*. Ejecutamos, con 2, 4, 6 y 8 procesos la primera estrategia en tres diferentes poblaciones. No aplicamos la segunda y tercera estrategia, pues estas están diseñadas para las distancias por enlace *simple* y *completo*.

La figura 4.9 muestra un buen rendimiento, reduciendo los tiempos de ejecución hasta con tamaños de poblaciones elevados. Para una población de 5000 individuos se consiguen

los siguientes *speedups* [1.78, 2.89, 3.61, 4.21]. Los *speedups* no crecen en proporción a los procesos ejecutados. La estrategia implementada, para tamaños de poblaciones elevados, no es optima. Si un proceso no elimina filas en muchas iteraciones, acumula muchos más datos que procesar que los demás procesos, provocando que los procesos con menos datos esperen para seguir con la ejecución.

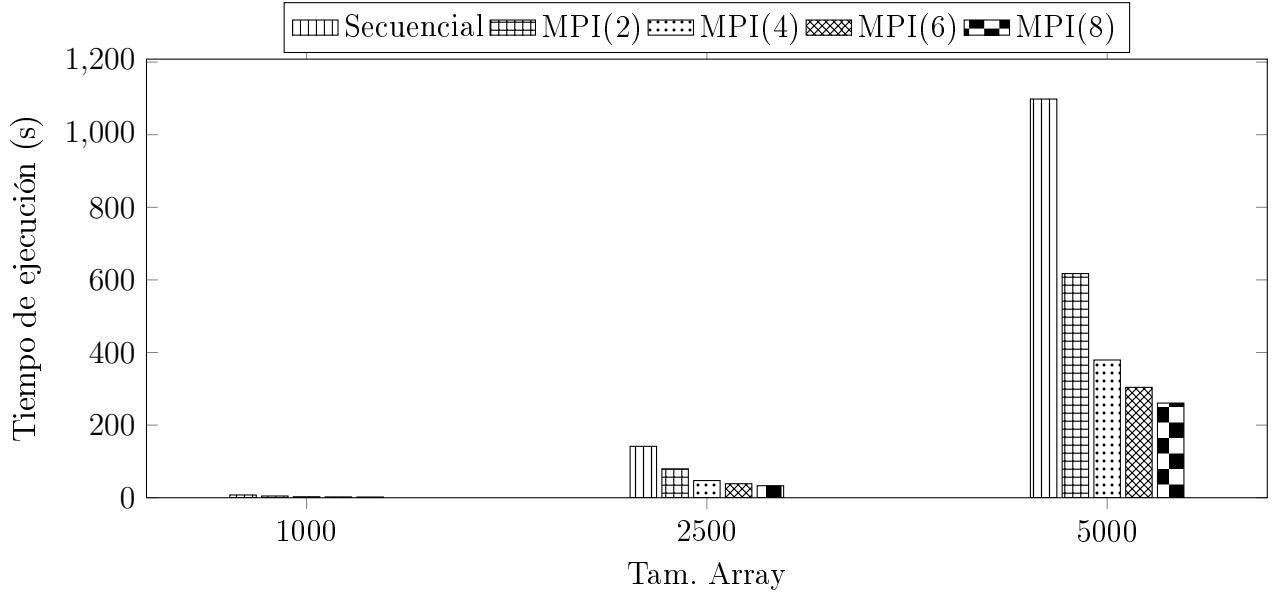


Figura 4.9: Tiempo de ejecución de la distancia entre clusters por *centroide* del algoritmo Jerarquico Aglomerativo en ordenador de propósito general

Ahora veamos el comportamiento de las estrategias para la distancia entre cluster con mayor complejidad, enlace *simple* o *completo*. Las pruebas se realizan con tamaños de poblaciones inferiores a las pruebas anteriores. Estos son los siguientes [100, 200, 500, 1000, 1500, 2000], y se ejecutan las estrategias con cuatro procesos para comprobar el rendimiento. Antes de nada, la tercera estrategia tiene un mismo rendimiento que la segunda, pero con más procesos. Reservar procesos únicamente para el cálculo de nuevas distancias no es eficaz, es mejor dividir entre los procesos activos (segunda estrategia). La figura 4.10 muestra los resultados obtenidos del estudio. Aunque si reduce el tiempo de ejecución, no se obtienen buenos resultados, pues el *speedup* con 2000 individuos de población para la estrategia con mejores resultados es de 1.88. Al usar cuatro procesos, podemos concluir que los tres *workers*

pierden mucho tiempo calculando las distancias en cada iteración. Es posible que mediante la refinación progresiva de la segunda estrategia a través de un proceso iterativo de prueba y error, se logre reducir el tiempo de ejecución. No obstante, hasta el momento, no hemos logrado reducirlo más allá del tiempo actual.

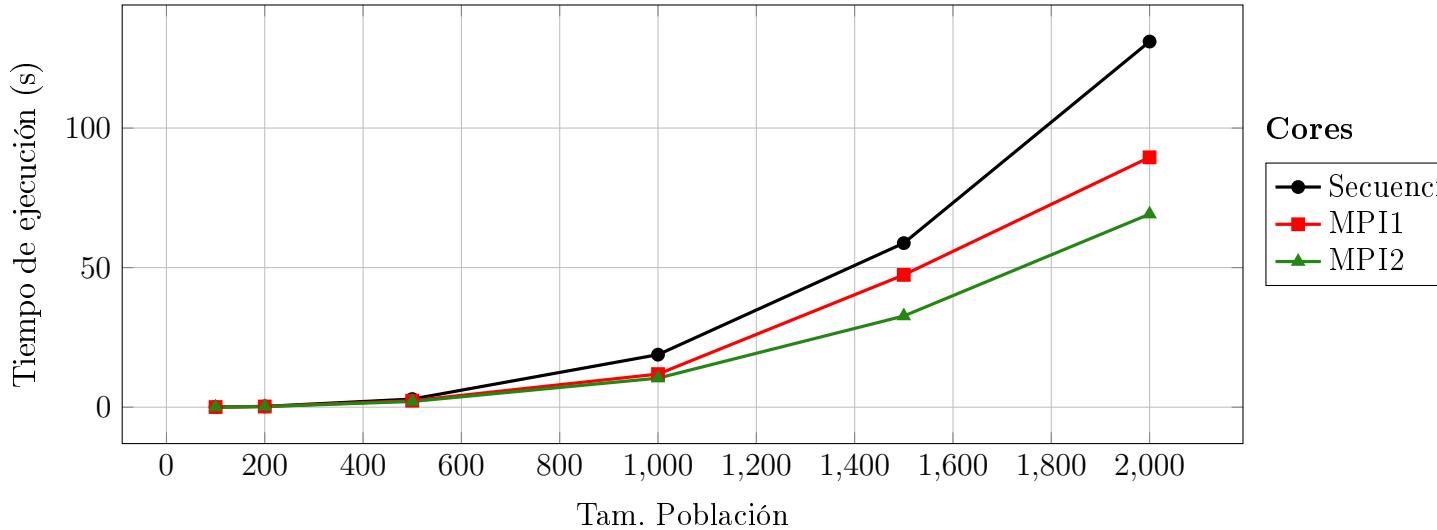


Figura 4.10: Tiempo de ejecución de la distancia entre clusters por enlace *simple* del algoritmo Jerarquíco Aglomerativo en ordenador de propósito general

Los resultados de la prueba anterior, con la complejidad del algoritmo indican que no es viable probar las estrategias implementadas sobre estas distancias entre cluster en el sistema altamente distribuido. Por este motivo, solo se prueba la distancia por *centroïdes* con tres grandes poblaciones. Los tamaños son los siguientes [5000, 7500, 10000], y se prueban con 20, 50, 75, 100 y 128 procesos. La figura 4.11 muestra los resultados, y concluimos que para agrupar tamaños de poblaciones elevados no conviene aplicar este algoritmo. O por lo menos las estrategias implementadas no dan resultados notorios, pues el *speedup* entre usar 20 o 128 procesos en una población de 10000 individuos es de 2.32.

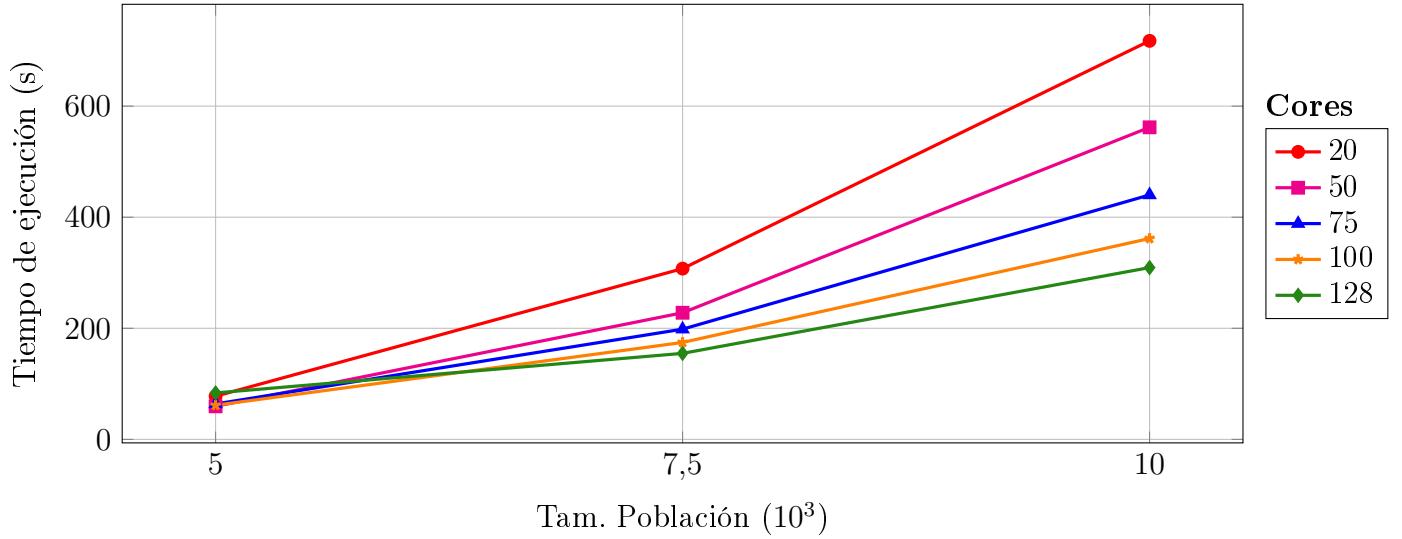


Figura 4.11: Tiempo de ejecución de la distancia entre clusters por *centroide* del algoritmo Jerarquico Aglomerativo en Cluster

#### 4.3.2. 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. Cuanto mayor sea el valor de  $K$  más tiempo va a consumir para realizar la asignación, pues cada individuo de la población es comparado con más centros. No obstante, dependiendo de la asignación de los individuos, una ejecución con más centros puede ser más rápida que otra con menos centros. Todo depende de la variable  $iter$ , es decir, si consigue llegar antes a la condición de finalización (que los centros no cambien entre dos iteraciones). La figura 4.12 demuestra precisamente este punto. Para dos poblaciones distintas, 75000 y 100000 aplicando  $K=25$  centros (línea roja), tardan aproximadamente lo mismo. La primera población itera muchas veces, más en concreto el doble de veces que la segunda población para finalizar la ejecución. Una ejecución del algoritmo sobre una misma población puede variar considerablemente dependiendo del

número de centros, o la disposición de los mismos.

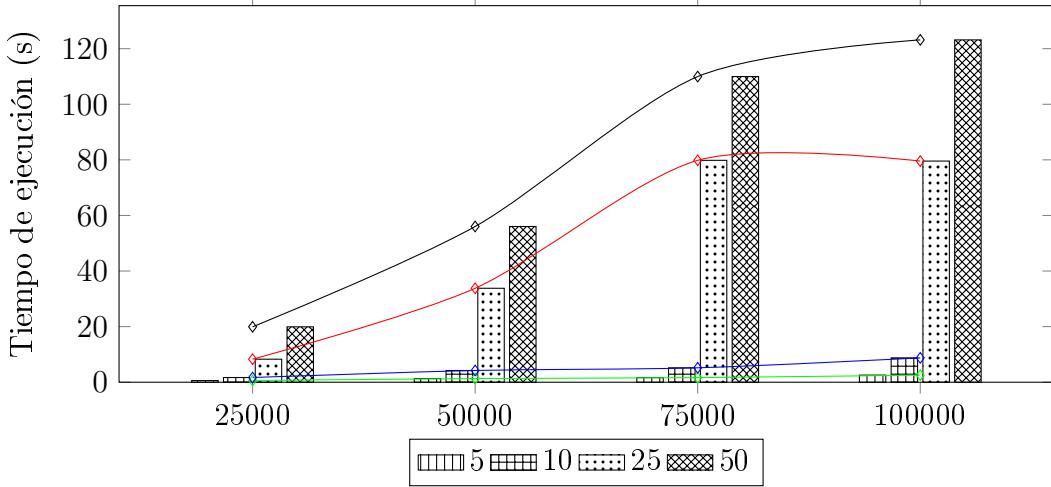


Figura 4.12: KMedias - Variaciones en el numero de clusters (K)

Las distancias entre individuos siguen presentes, pero esta vez, al tener una complejidad menor, no debería afectar tanto usar la distancia *euclídea* o *manhattan*. O eso es lo que parece a simple vista. Como se comprobó en la figura anterior (4.12) el número de iteraciones para llegar a la condición de finalización importa, y usar una distancia u otra va a influir en el tiempo de ejecución. El número de iteraciones varía dependiendo de que distancia se usa, pues la *euclídea*, aunque su cálculo es más lento, tiene una mayor precisión, lo que le da una gran ventaja frente a la distancia *manhattan*. Esta última, al no ser tan precisa, puede hacer que aunque sea por poco, un individuo pertenezca a otro cluster, provocando una reacción en cadena que resulte en un aumento considerable en el número de iteraciones.

El estudio realizado para comprobar el rendimiento de la estrategia comentada en la sección 3.2.2 con cinco procesos frente el algoritmo secuencial, se representa en la figura 4.13, utilizando  $K=10$  centros, y comparando también las distancias entre individuos (*euclídea* y *manhattan*). Los tamaños de las poblaciones utilizadas para medir estas pruebas se realizan como en las pruebas de las ordenaciones cuadráticas 4.2.1.1. Se puede apreciar que las funciones tienen picos, siendo más pronunciados en los algoritmos sin paralelizar. Como se comentó anteriormente, el tiempo de ejecución para una población puede variar dependiendo

de la distancia implementada, además de la posibilidad de que una población con menor tamaño pueda tardar mucho mas que una población mayor, debido a la disposición de los individuos y los clusters en la ejecución.

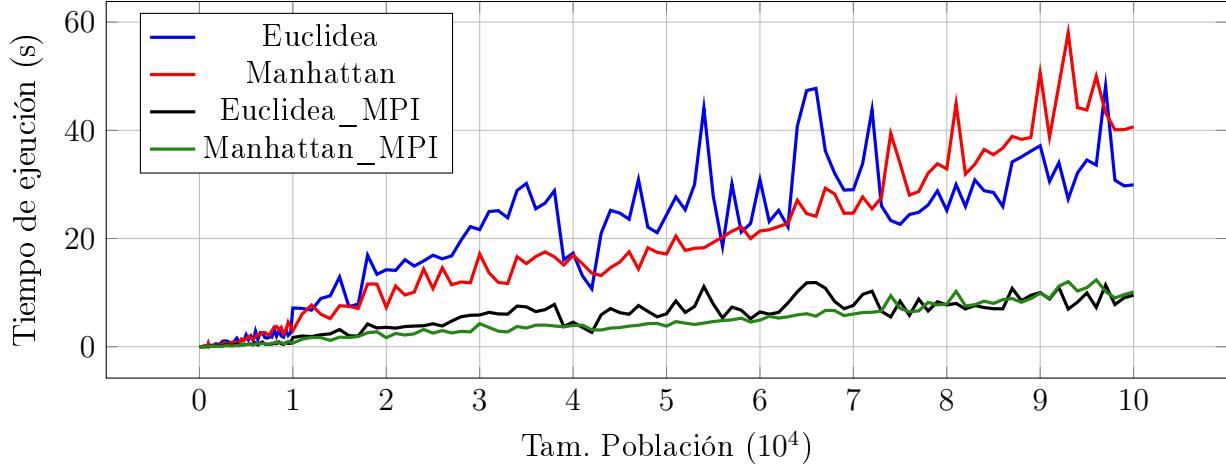


Figura 4.13: Tiempo de ejecución de KMedias

Comparando el algoritmo secuencial y paralelizado se puede apreciar una mejora considerable, y debido a los picos, es interesante medir la evolución de los speedups. La figura 4.14 muestra esta evolución, cuyos *speedups* son calculados con los tiempos utilizados en la anterior figura. Ambas distancias comienzan siendo volátiles, siendo algunas veces peor que el algoritmo secuencial ( $speedup < 1$ ) y otras veces superando por mucho el *speedup* ideal. A partir de diez mil individuos de población, el *speedup* es equivalente al numero de *workers* ejecutados. Y lo más curioso es, que pese a que la distancia *euclídea* es más precisa, a la larga es mejor aplicar distancia *manhattan*, pues aunque itere más veces, el coste es menor, llevando a conseguir mejores resultados. Se puede apreciar la linea azul superando en la mayoría de las veces a la línea roja, probando lo recien comentado.

Para este algoritmo, al contrario que el anterior, se pueden realizar pruebas con tamaños de poblaciones mayores en el sistema altamente distribuido. La siguiente prueba realizada comienza con una población de 20000 individuos, esta vez con cinco variables de entrada. Entre pruebas se aumenta ese mismo tamaño hasta llegar a 240000 individuos, utilizando

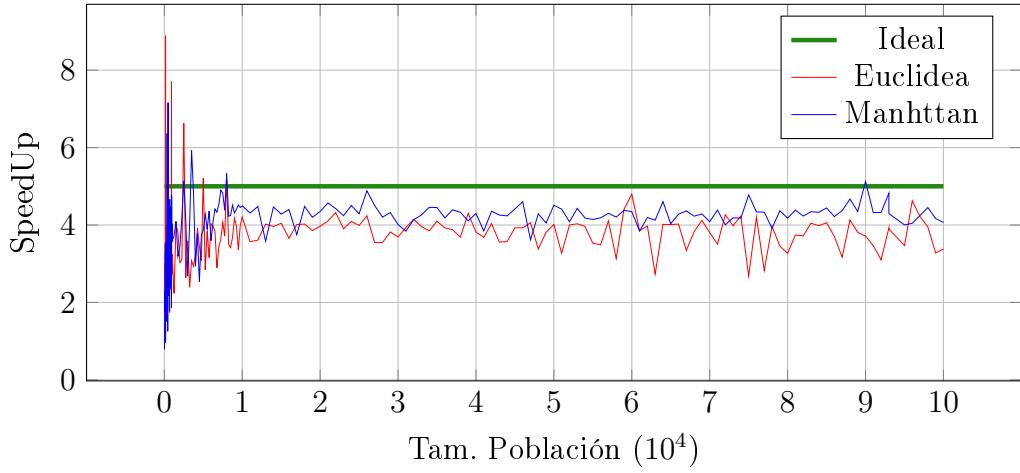


Figura 4.14: KMedias - SpeedUp

en proporción una población seis veces mayor que en el ordenador de propósito general. Se usa el mismo valor de  $K$  ( $K=10$ ), y se ejecuta la misma estrategia con  $10, 20, 35, 50, 75, 100$  y  $128$  procesos. Como se muestra en la figura 4.15, a partir de veinte procesos, la reducción del tiempo de ejecución se ralentiza. Con un número elevado de procesos, esta estrategia no consigue reducir el tiempo de ejecución en proporción a los procesos ejecutados, esto se debe a la gran cantidad de comunicaciones que se deben realizar para finalizar la ejecución.

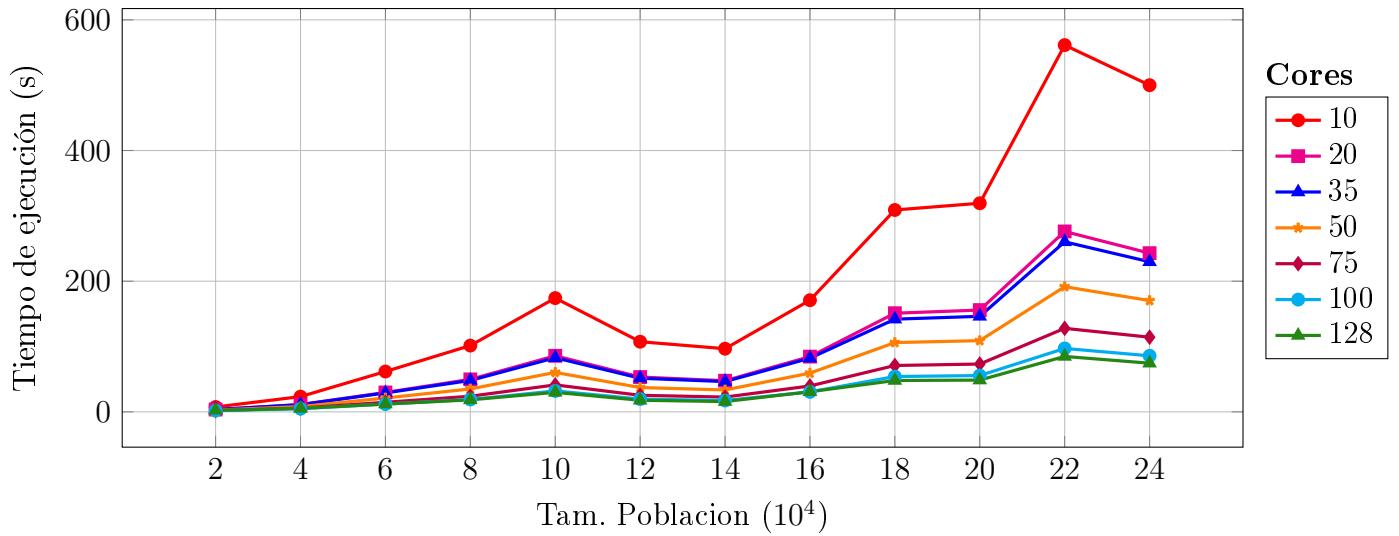


Figura 4.15: KMedias - Pruebas en el cluster

### 4.3.3. KNN

En cada iteración de este algoritmo de aprendizaje supervisado, se clasifica un individuo utilizando una población previamente categorizada. Al contrario que los algoritmos de aprendizaje no supervisado, que agrupan una población entera al finalizar la población. La complejidad temporal de este algoritmo es menor, y el valor de  $K$  no influye en el tiempo de ejecución como el algoritmo de *K-Medias*, pues al aumentar este valor solo aumenta el número de los individuos mas cercanos que se comprueban para categorizar el nuevo individuo. Este algoritmo usa dos poblaciones, y como se comentó en la sección 3.2.3 las dos estrategias dividen una de las poblaciones para parallelizar el algoritmo.

Para las siguientes pruebas realizadas en el ordenador de propósito general, se fija la misma población utilizada en el algoritmo anterior, con un tamaño de 100000 individuos, para la población a categorizar. Y la población inicial se obtiene al realizar una búsqueda del mejor número de clusters y agrupación la final con el algoritmo de *K-Medias*, sobre una población de mil individuos, obteniendo cuatro clusters. Con estas dos poblaciones se ejecuta el algoritmo de *K-Vecinos más Cercanos* con un valor de  $K=15$ , un número impar para que no haya posibilidad de empates a la hora de asignar un cluster a cada individuo.

Primero comprobamos las dos métodos para el algoritmo secuencial, actualizar o no actualizar al categorizar un nuevo individuo. Si se actualiza la población conforme avanzan las iteraciones, la población final será mucho más precisa que si no se actualiza, pero el tiempo de ejecución aumentará considerablemente. La figura 4.16 muestra este aumento. Si no se actualiza, la complejidad es lineal, pues la población categorizada se mantiene constante, y no se puede diferenciar cual de las dos distancias ralentiza más la ejecución. Sin embargo, cuando se actualiza la población, se comprueba una vez más que la distancia *euclídea* es más lenta que la *manhattan*.

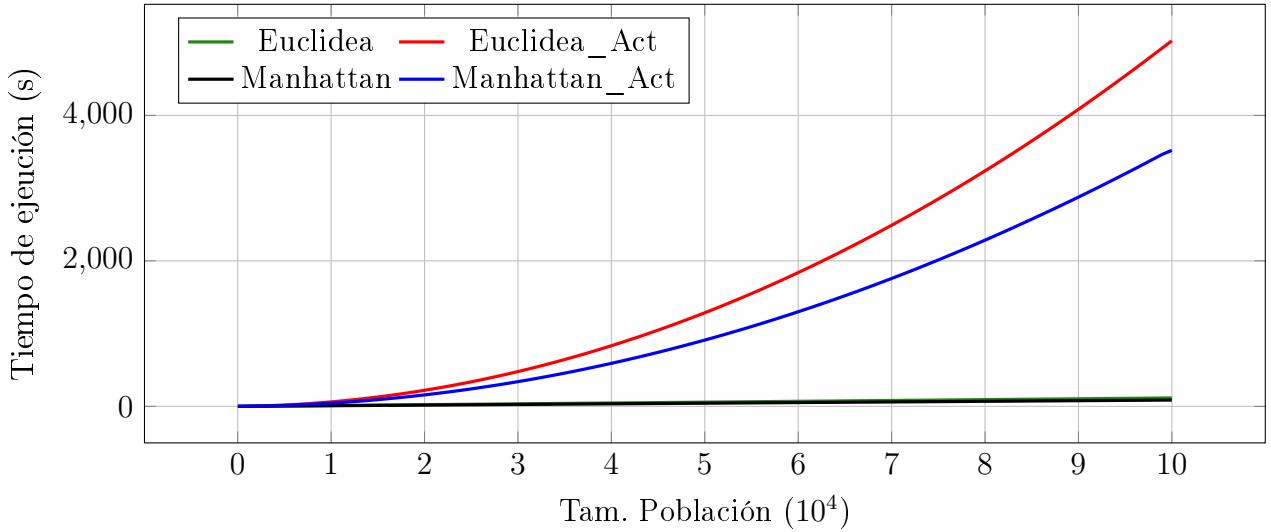


Figura 4.16: KNN - Tiempo de ejecución del algoritmo sin mejoras

Despues de comprobar el algoritmo secuencial pasamos a las estrategias para reducir el tiempo de ejecución. El algoritmo sin actualizar es veloz y es mejor estudiar el comportamiento con una población variable con el tiempo. Por esos se ejecutan las dos estrategias con cinco procesos. La primera, de dividir la poblacion categorizada entre los *workers*, se realizan dos versiones, una en la que cada *worker* espera el individuo categorizado de la iteración anterior y otra en la que no se espera, sino que los *workers* trabajan en la siguiente iteración mientras que el *master* agrupa el individuo. Podemos ver los resultados en la figura 4.17, con una reducción notoria en el tiempo de ejecución. La primera estrategia es ligeramente más veloz que la segunda, y aun perdiendo tiempo esperando a la categorización del individuo (la primera versión), sigue finalizando antes que la segunda estrategia.

En cuestión de complejidad espacial la segunda estrategia consume mucha más memoria. Al finalizar la ejecución, cada *worker* tiene una copia entera de la población categorizada, mientras que la primera mejora se divide esta población entre los procesos.

Comparando las evoluciones de los *speedups* de las estrategias, se puede concluir que 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.

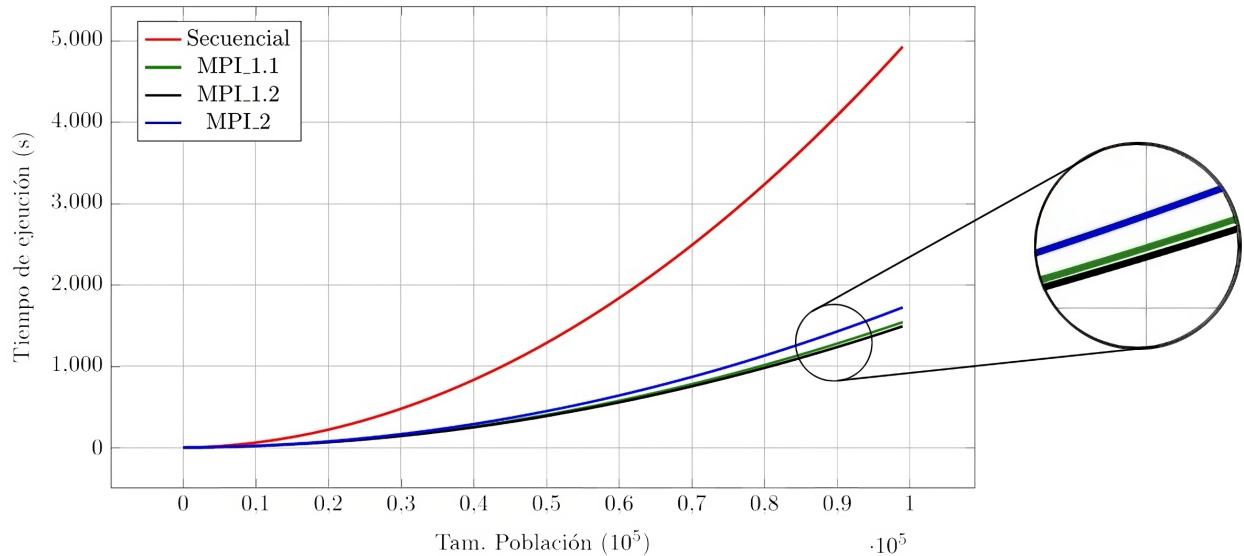


Figura 4.17: KNN - Tiempo de ejecución con las mejoras MPI

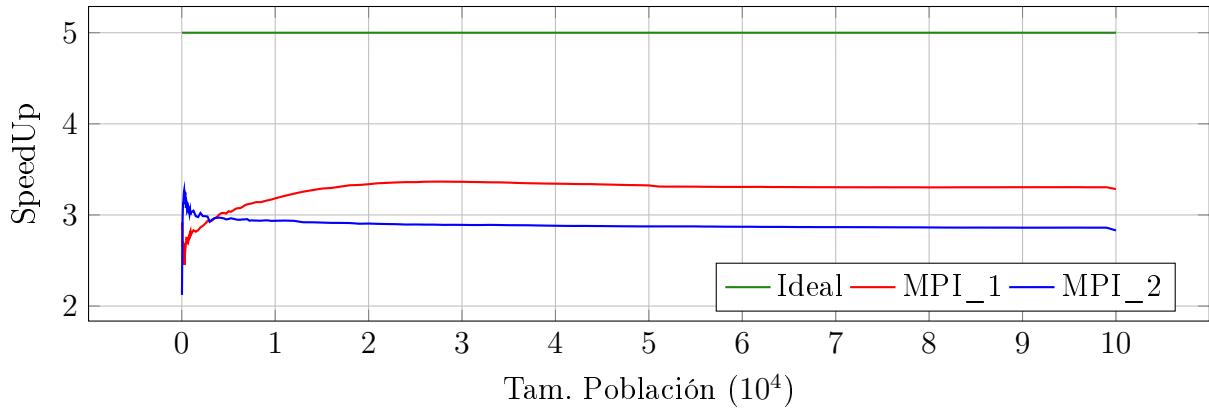


Figura 4.18: KNN - SpeedUp

En algoritmos pasados ya hemos visto el funcionamiento de varias estrategias con tamaños de poblaciones elevados. Esta vez, para las pruebas en el sistema altamente distribuido, ejecutamos la misma prueba que antes, pero con más procesos en paralelo. Se ejecutan 10, 20, 35, 50, 75, 100 y 128 procesos sobre la mejor estrategia obtenida en el estudio anterior, para ver si es óptimo usar muchos procesos o genera mucha sobrecarga. La figura 4.19 muestra que, al aumentar los procesos no reduce considerablemente el tiempo de ejecución, generando sobrecarga a partir de veinte procesos ejecutándose. Al igual que en el algoritmo de *K-Medias*, aumentar el número de procesos provoca que, aunque se reduce el tiempo de

ejecución en cada iteración, el tiempo de comunicación entre iteraciones aumenta.

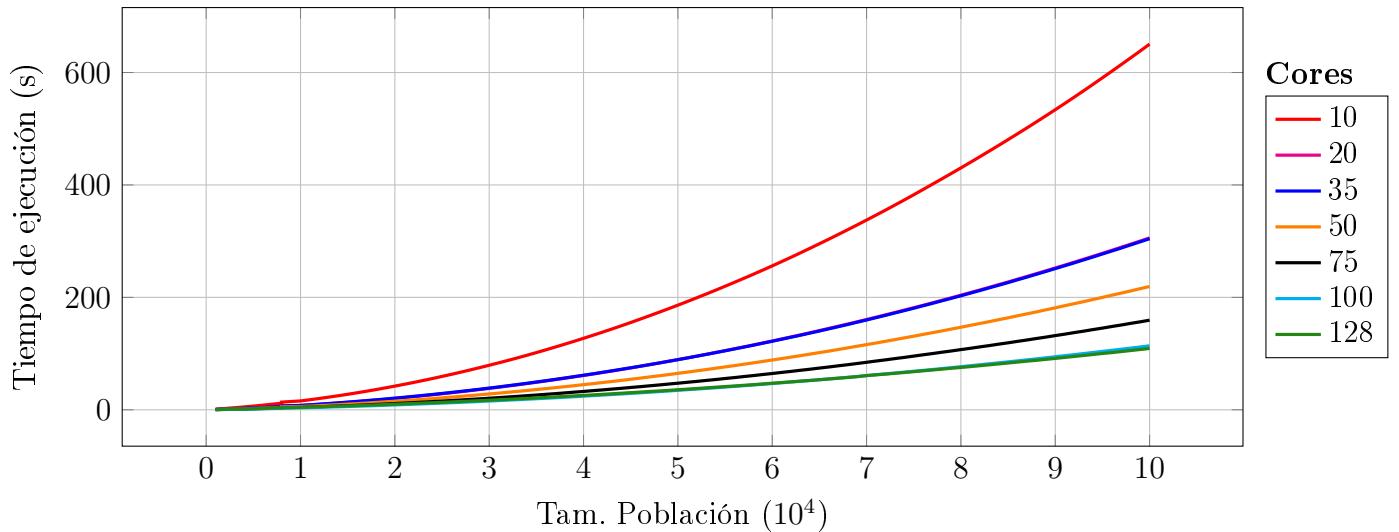


Figura 4.19: KNN - Pruebas en el cluster

## 4.4. Q-Learning

Para el aprendizaje por refuerzo, cuyos dos algoritmos se comentaron en la sección 2.2, primero se estudia el algoritmo de *Q-Learning*. El otro algoritmo, *Deep Q-Network*, se basa en redes neuronales, estudio que se realiza posteriormente en la sección 4.6.

Antes de entrar en profundidad con las estrategias comentadas en la sección 3.3, primero estudiamos el comportamiento del algoritmo de manera secuencial, con y sin preprocesado del entorno. Este preprocesado consiste en recorrer la matriz entera eliminando estados inaccesibles (el agente se sitúa en un muro) y acciones que no queremos que el agente ejecute, como chocar contra una pared. Se ejecuta con tres diferentes laberintos, con 30, 50 y 100 filas. La figura 4.20 muestra una leve reducción en el tiempo de ejecución. Además, obtiene mejores resultados con una variedad mayor de combinaciones de hiperparámetros. Al reducir las acciones disponibles, el agente tiene una mayor probabilidad de explorar más el laberinto, generando más combinaciones con las cuales aprender el camino óptimo hasta la meta.

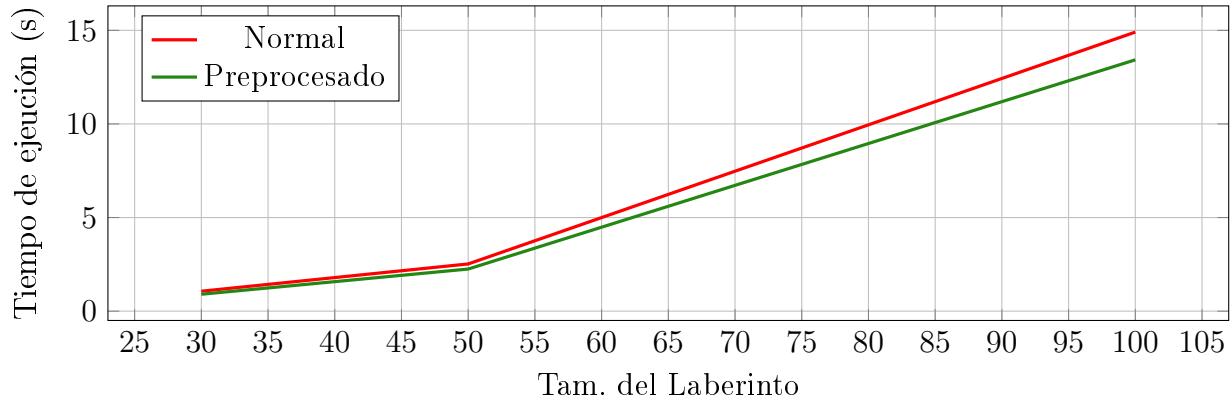


Figura 4.20: Tiempo de ejecución para RL

Una buena configuración de hiperparámetros, genera que el agente logre alcanzar su objetivo. En entornos de gran tamaño, algunas veces, es complicado encontrar configuraciones que funcionen, lo que provoca un aumento en el tiempo de dedicado a la fase de entrenamiento para encontrar estas combinaciones. Por este motivo se desarrolla una estrategia para encontrar combinaciones de los hiperparámetros realizando una búsqueda exhaustiva.

Consiste en ejecutar en muchos procesos el algoritmo secuencial sin preprocesar con diferentes combinaciones. Se inicializa 100 filas y columnas, y con una precisión de 0.01, es decir un 1 %, el *master* aumenta los valores de los hiperparámetros. Cuando uno de estos llega al 100 % se reinicia y aumenta el siguiente, así hasta cubrir todas las posibles combinaciones. Al ser tres hiperparámetros hay  $10^6$  combinaciones distintas. El *master* envía a cada *worker* diferentes combinaciones, y cuando terminan de procesar una combinación, envían de vuelta un mensaje de confirmación. Si termina con éxito, el *master* almacena en un fichero la combinación de hiperparámetros, junto con los movimientos requeridos, así como las veces que el algoritmo falla y llega a la meta. En caso contrario no almacena nada. Dicha prueba se realizó en el sistema altamente distribuido, con 128 procesos, y ha tardado siete días completos en finalizar la prueba, obteniendo 500 combinaciones distintas, de las cuales, solo la mitad dan con el camino óptimo.

La primera

## **Procesos**

Esta mejora se basa en el modelo *Master-Worker*. El proceso Master divide el entorno entre todos los procesos (incluyéndose). También se encarga del punto de partida del agente (parte superior izquierda) y genera nuevos agentes conforme salen de su perímetro. Los demás procesos esperan a recibir un agente de algún proceso para poder explorar su entorno.

Esta mejora no da buenos resultados. No hemos encontrado una configuración de hiperparámetros para que el agente aprenda a llegar al destino, se queda dando vueltas en cualquier proceso. Si en vez de inicializar las experiencias a 0, usamos el mapa aprendido usando el algoritmo sin mejoras, si llega al destino. Hay una probabilidad de que el problema sea de la comunicación entre los procesos, sino de encontrar unos buenos hiperparámetros para que llegue al destino.

### **4.4.0.1. Mejora: Ejecuciones en paralelo**

## **Procesos**

Usando el modelo *Master-Worker*. El proceso Master envía las posiciones aleatorias y el laberinto a los Workers, para que estos investiguen el laberinto y devuelven las experiencias obtenidas para que el Master haga la media y aprenda las mejores acciones para cada estado del entorno.

Esta mejora aplica la misma idea que la búsqueda de hiperparámetros, pero esta vez para investigar por completo el laberinto, desde distintos puntos de partida, para encontrar más rápidamente el mejor camino para llegar a la meta. Al ejecutar N Workers, el Master recibe la experiencia de estos procesos, provocando una reducción en el tiempo de ejecución proporcional al número de procesos ejecutados.

#### **4.4.0.2. Cluster**

TODO CLUSTER, ejecutado el 27 de junio.

## 4.5. 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 altura de los individuos. Para mejorar la aptitud de los individuos se aplica un desplazamiento, cuya finalidad es que 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.

Para la siguiente tabla, se marca en rojo los métodos más tardíos para cada problema.

- Con individuos binarios, conviene dar más recursos a las operaciones de cruce y mutación.
- Los otros dos individuos (reales y árboles) es mejor parallelizar la función de evaluación, dependiendo de los datos de entrada de dichos problemas.

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 4.2: PEV - Tiempos de cada método

#### 4.5.0.1. Algoritmos sin mejoras

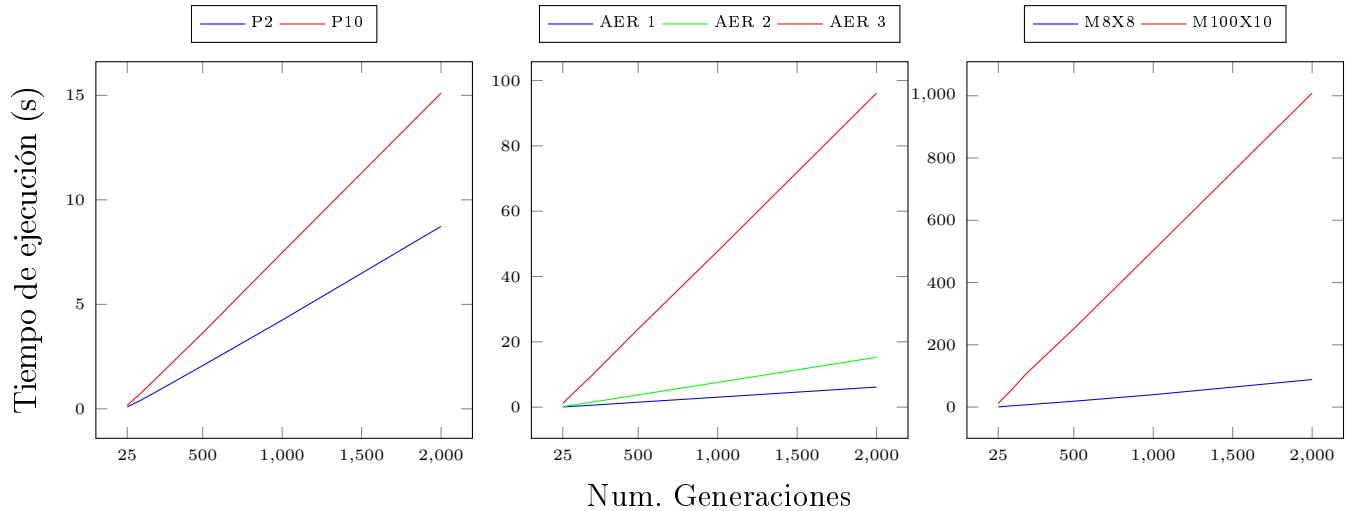


Figura 4.21: PEV Secuencial

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  $O(\text{Ticks})$ .

#### 4.5.0.2. 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 y tener más diversidad, siendo más probable obtener mejores resultados.

##### **Procesos**

Esta prueba se realiza con **cuatro procesos**, tienen las mismas variables iniciales. Cada proceso inicializa aleatoriamente la población que va a evolucionar conforme avanzan las generaciones. Se conectan cada  $X$  ( $X=100$ ) generaciones para reiniciar las poblaciones.

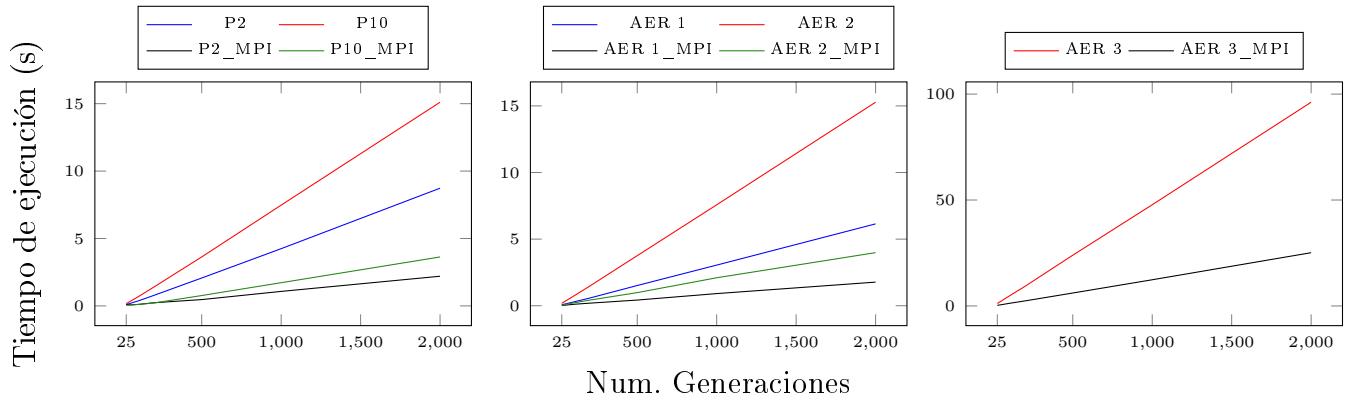


Figura 4.22: MPI - Modelo de Islas

El problema de los árboles calcularía resultados parecidos a estos dos gráficos. Al ser lineales y ejecutar en paralelo los procesos se obtiene un speedup aproximado al número de procesos ejecutados. Como se puede apreciar en la siguiente gráfica:

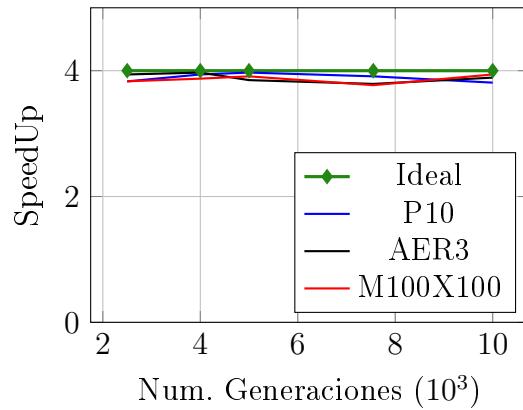


Figura 4.23: SpeedUp - Modelo en Islas

#### 4.5.0.3. Mejora 1: Dividir con el master

##### Procesos

Esta vez tenemos una única población. Aplicando el modelo *Master-Worker* 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 en total** contando el Master.

En cada iteración, el Master envía a los Workers el tamaño de de sub-población con el que van a trabajar. Estos inicializan la población, la evalúan y la envían de vuelta, para que

hacer la selección con todos los individuos. Envía las partes de la selección a los procesos para que estos la crucen, muten y evaluen, así iterando por el número de generaciones.

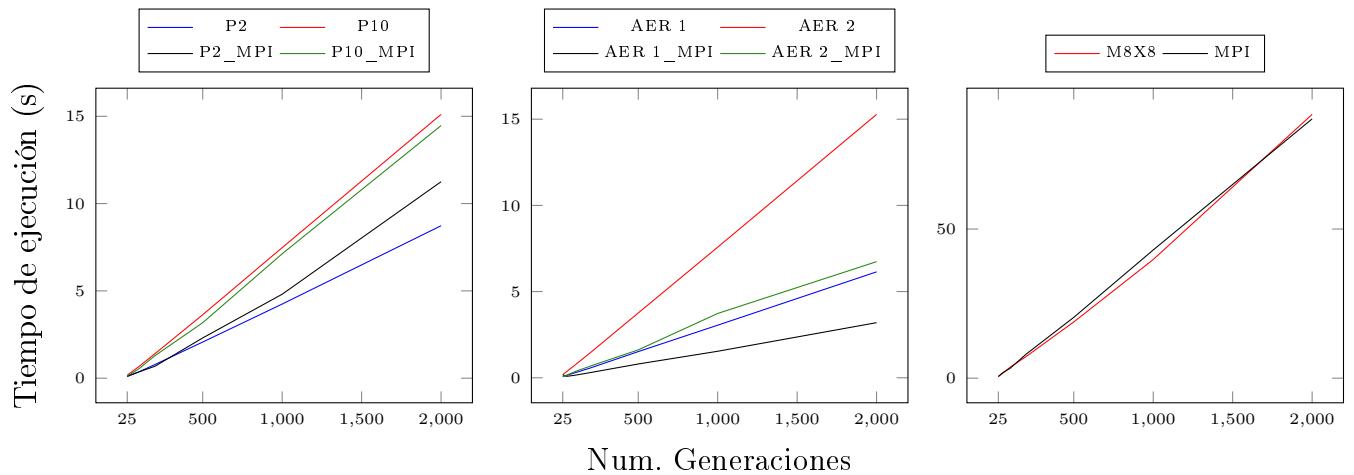


Figura 4.24: MPI1 - Dividir Poblacion con problemas pequeños

- (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.

Con los problemas con pocos datos de entrada, no conviene paralelizarlos, pues son bastante rápidos. Pero al aumentar el tamaño de los problemas, puede ser beneficiosos paralelizarlos.

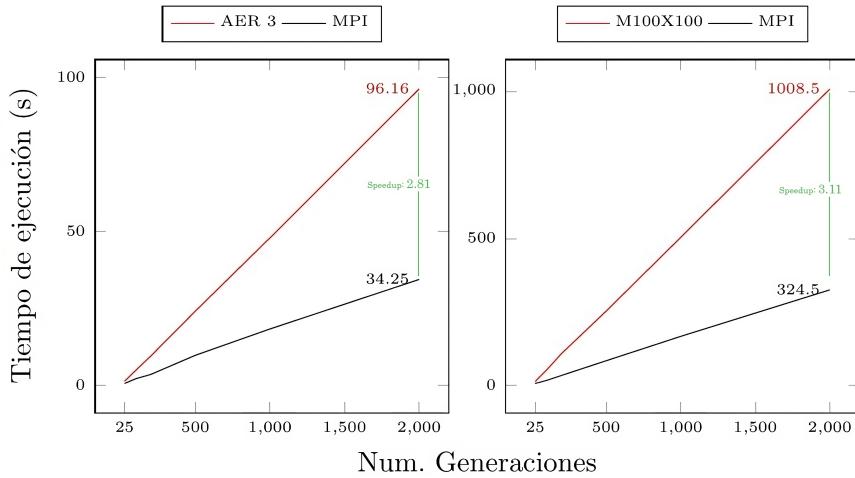


Figura 4.25: MPI1 - Dividir Poblacion con problemas grandes

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.

#### 4.5.0.4. Mejora 3: PipeLine

##### Procesos

Mezclando el modelo *Master-Worker* con segmentación, el proceso Master se encarga de generar N (número de workers) poblaciones que envía al siguiente proceso, cuando genera todos se queda en un estado de recepción de mejores individuos. Cada proceso envía su siguiente los datos procesados según su tarea, generando un flujo constante de trabajo.

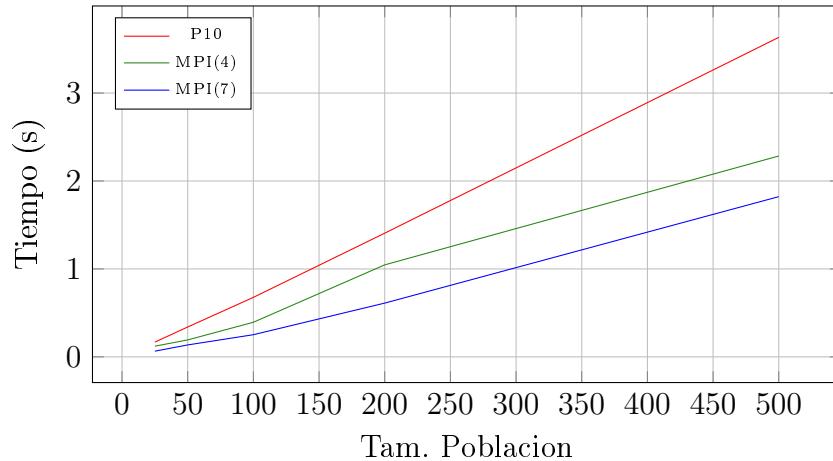


Figura 4.26: PipeLine - Individuos Binarios

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, 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 individuos no se puede ejecutar con dos decimales de precisión con el ordenador de propósito general, y con 500 individuos es el máximo tamaño de población si se usan 10 decimales de precisión.

**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.

Para los individuos reales y arboles, al no tener tantos elementos por individuo, esta mejora funciona correctamente. Estos dos individuos tienen la misma implementación al tener tiempos similares y perder la mayor parte del tiempo en la evaluación de los individuos.

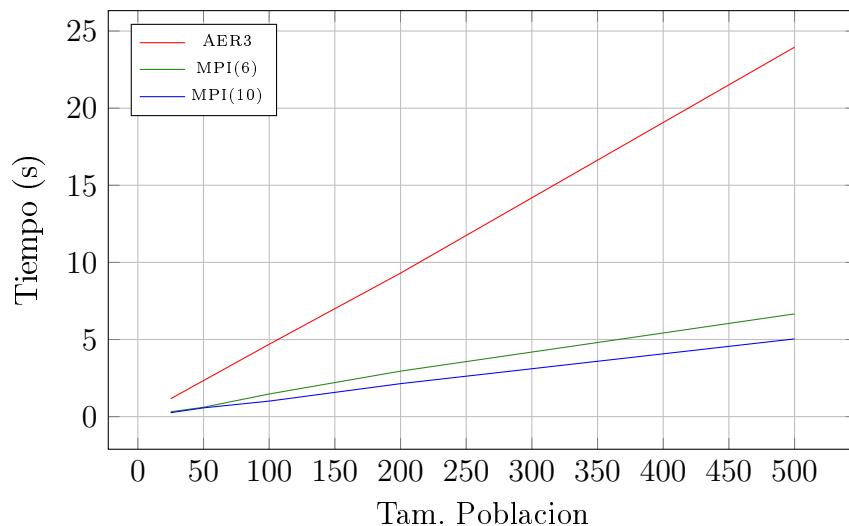


Figura 4.27: PipeLine - Individuos Reales

**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 4.2).

Evaluación	Selección	Cruce	Mutación
400e-06s	8e-06	40e-06	10e-06

Cuadro 4.3: PEV - Tiempos de aproximados usados

Al juntar los últimos tres métodos, tarda un tiempo aproximado de 58e-06. 6.9 veces más rápido que la evaluación. Por simplicidad, es mejor ejecutar potencias de dos procesos, para hacer una división equitativa.

#### 4.5.0.5. Cluster

**Datos y número de procesos**

Para la siguiente prueba se ejecuta la primera mejora, el *Master* se encarga de dividir la población entre los procesos, con 10, 20, 50 y 100 procesos *Workers*. Se ejecutan 25 generaciones, con las siguientes poblaciones [1000, 2000, 5000, 7000].

Ambos problemas, como mencionamos anteriormente tiene tiempos parecidos. La **sobrecarga se empieza a notar con cincuenta procesos**, pues la mejora entre usar veinte y cincuenta es de un 25 % mas rápida, además de ser más eficaz que usar cien procesos.

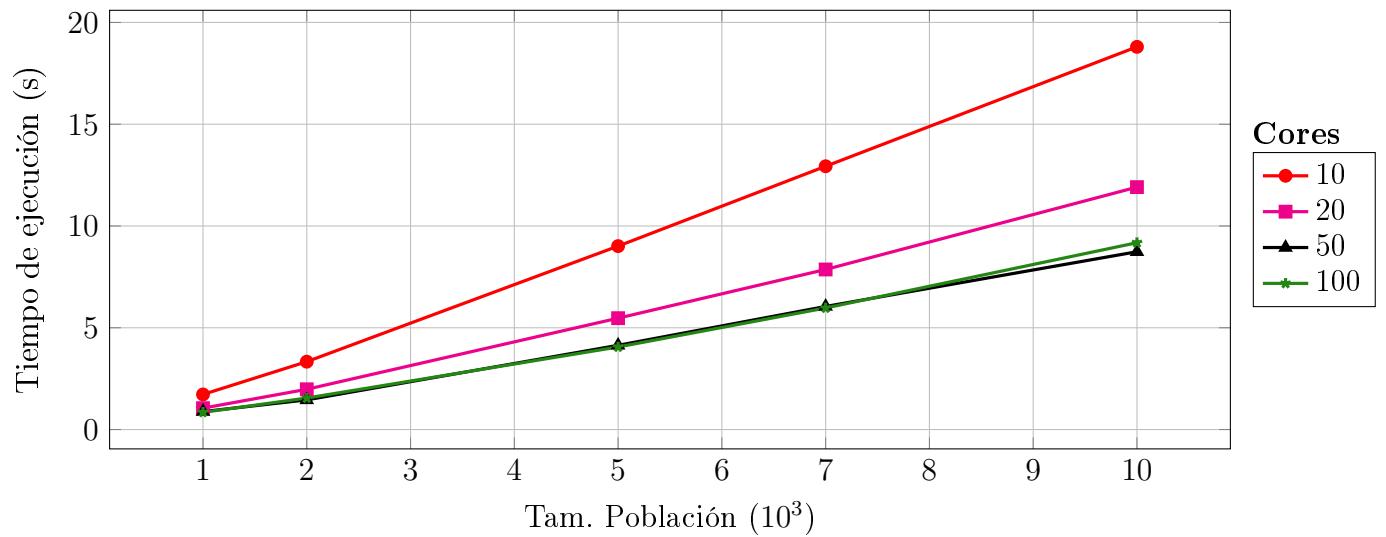


Figura 4.28: PEV Real - Tiempo de ejecución en el Cluster

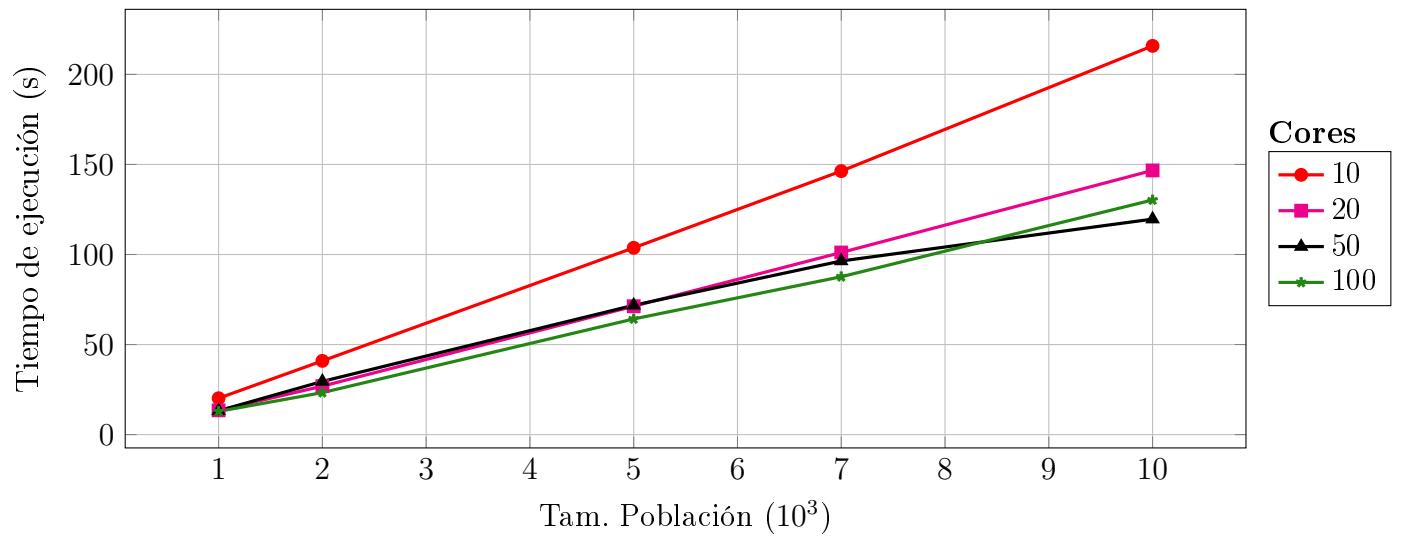


Figura 4.29: PEV Arbol - Tiempo de ejecución en el Cluster

## 4.6. Redes Neuronales

### Pruebas

Se usa una población de 80 individuos, generados previamente de manera no aleatoria, para tener una concordancia con valores reales, de altura y masa corporal, y así calcular IMCs con concordancia. Se generan con combinaciones en las cuales se aumenta la altura cada vez que se crean 10 individuos con diferentes pesos. Los individuos oscilan en alturas de [150, 200]cms.

#### 4.6.0.1. Mejora 1: PipeLine

### Procesos

Aplicando el modelo *Master-Worker* junto con segmentación, cada proceso se encarga de un determinado numero de capas. El proceso Master se encarga de empezar a categorizar los individuos, ejecutando el metodo forward() en sus capas, una vez llega a su última capa envía los datos que ha procesado al siguiente proceso, y genera otro individuo. Los procesos procesan los individuos que reciben, generando un flujo de individuos. El último proceso se encarga de evaluar y gestionar los errores para enviar hacia atrás (método de propagación hacia atrás).

Para la siguiente prueba se utiliza una red neuronal con capa oculta de 1x5, es decir, una capa y cinco nodos.

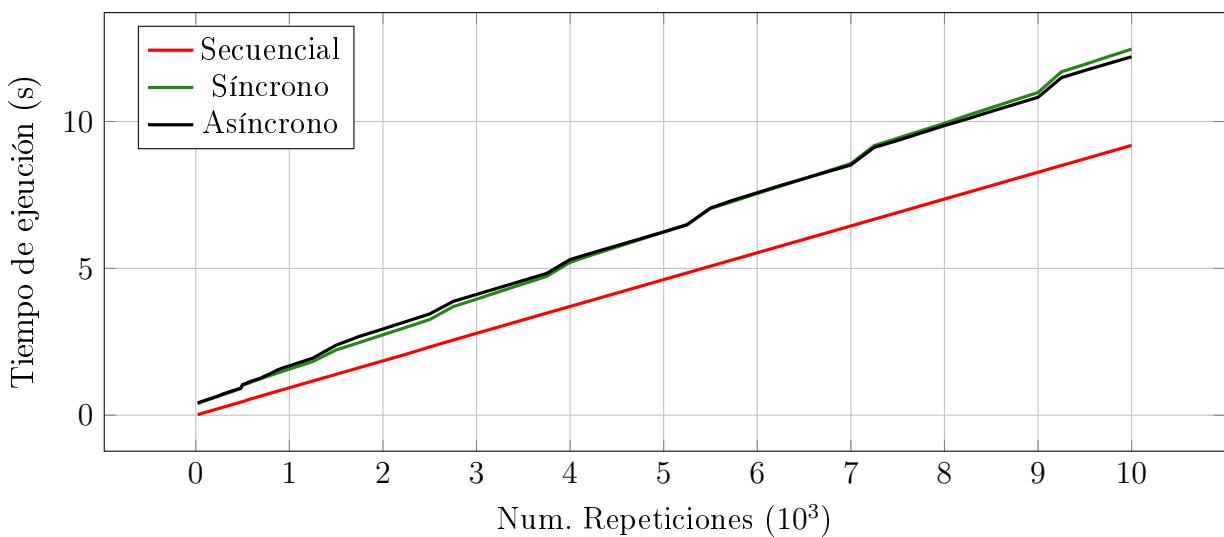


Figura 4.30: MPI1 - Red Neuronal

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**

#### 4.6.0.2. Mejora 2: Dividir el trabajo en procesos

##### Procesos

Para esta mejora se aplica el modelo *Master-Worker*. El proceso Master se encarga de dividir eficazmente la población, para el máximo rendimiento de la red neuronal. Cada Worker se encarga de entrenar con una población distinta para aprender correctamente como categorizar los individuos. Una vez se ha terminado la fase de entrenamiento, envían al Master todas sus experiencias para que este haga la media y pueda aprender a predecir correctamente. (Para el correcto funcionamiento la red neuronal tiene que ser grande, si no no se puede garantizar que prediga correctamente para todos los individuos de la población)

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.

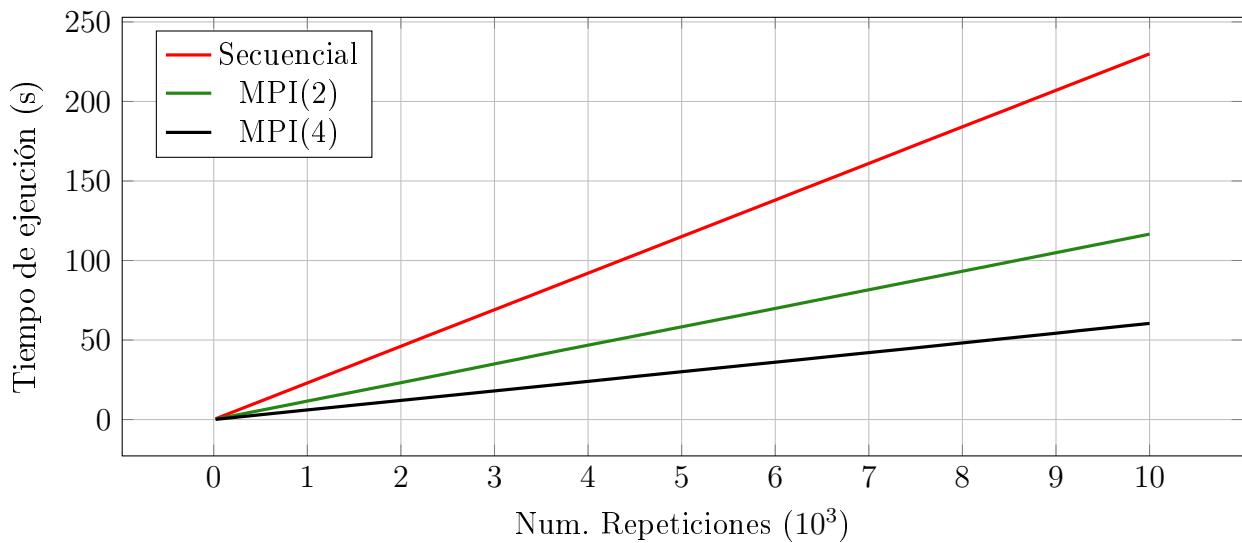


Figura 4.31: MPI2 - Red Neuronal Dividiendo entrenamiento

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 ( $10 \times 10$ ). 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  $[0.01, 0.20]$ .

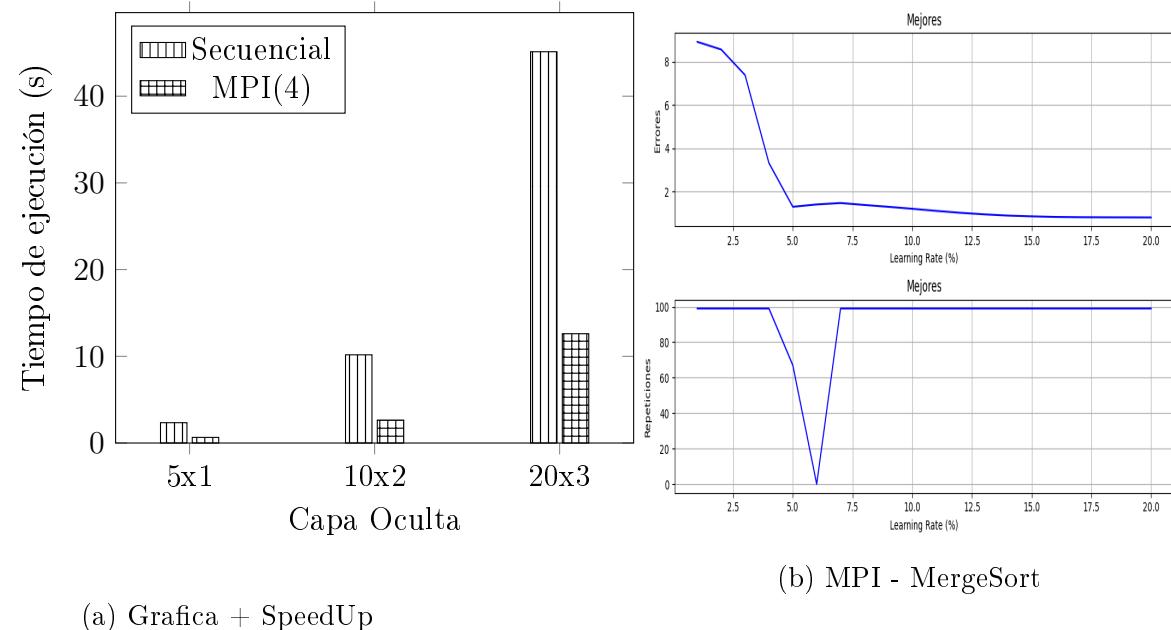


Figura 4.32: Mejoras MPI de las ordenaciones

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 que no siempre utilizar más procesos deriva en un mejor rendimiento, la sobrecarga de los procesos en las implementaciones es un fundamento a tener en cuenta a la hora de ejecutar programas.

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.

# Bibliografía

- [1] Bloom ai.
- [2] Chlouispy - maze generator.
- [3] Marcel R Ackermann, Johannes Blömer, Daniel Kuntze, and Christian Sohler. Analysis of agglomerative clustering. *Algorithmica*, 69:184–215, 2014.
- [4] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Merge strategies: from merge sort to timsort. 2015.
- [5] Brandon Barker. Message passing interface (mpi). In *Workshop: high performance computing on stampede*, volume 262. Cornell University Publisher Houston, TX, USA, 2015.
- [6] 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.
- [7] 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.
- [8] Frédéric Garcia and Emmanuel Rachelson. Markov decision processes. *Markov Decision Processes in Artificial Intelligence*, pages 1–38, 2013.
- [9] Henrik Jeppesen. Carbon tracker initiative. In *World Scientific Encyclopedia of Climate Change: Case Studies of Climate Risk, Action, and Opportunity Volume 1*, pages 63–69. World Scientific, 2021.

- [10] Keith Kirkpatrick. The carbon footprint of artificial intelligence. *Communications of the ACM*, 66(8):17–19, 2023.
- [11] Frances Y Kuo and Ian H Sloan. Lifting the curse of dimensionality. *Notices of the AMS*, 52(11):1320–1328, 2005.
- [12] Giuseppe Lugano. Virtual assistants and self-driving cars. In *2017 15th International Conference on ITS Telecommunications (ITST)*, pages 1–5. IEEE, 2017.
- [13] Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D Lee, Danqi Chen, and Sanjeev Arora. Fine-tuning language models with just forward passes. *Advances in Neural Information Processing Systems*, 36:53038–53075, 2023.
- [14] 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.
- [15] Surender Mor, Sonu Madan, and Kumar Dharmendra Prasad. Artificial intelligence and carbon footprints: Roadmap for indian agriculture. *Strategic Change*, 30(3):269–280, 2021.
- [16] Jeff Pool. Accelerating sparsity in the nvidia ampere architecture. *GTC 2020*, 2020.
- [17] José Jaime Ruz Ortiz. Multiprocesadores de memoria compartida y distribuida. Universidad Complutense de Madrid (UCM), 12/01/2016.
- [18] Mohd Shamrie Sainin. Best programming languages for ai. 2021.
- [19] Harold S Stone. *High-performance computer architecture*. Addison-Wesley Longman Publishing Co., Inc., 1990.
- [20] Christopher A Thomas and Xander Wu. How global tech executives view us-china tech competition. 2021.

- [21] 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

**AI** Artificial Intelligence

**MPI** Message Passing Interface

**CPU** Central Processing Unit

**GB** Giga-Byte

**RAM** Random Access Memory

**CO<sub>2</sub>** Carbon Dioxide

**HPC** High Performance Computing

**SPMD** Single Program Multiple Data

**RL** Reinforcement Learning

**MDP** Markov Decision Process

**DQN** Deep Q-Network

**KNN** K-Nearest Neighbors

**PEV** Programación EVolutiva