



**UNIVERSIDAD  
DE GRANADA**

## **PROYECTO FIN DE GRADO**

**DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y  
ADMINISTRACIÓN Y DIRECCIÓN DE EMPRESAS**

# **Comparativa de consumo energético y calidad de soluciones en algoritmos evolutivos**

**Autor**

Francisco Javier Luque Hernández

**Directores**

Pablo García Sánchez

Sergio Iván Aquino Britez



Escuela Técnica Superior de Ingenierías  
Informática y de Telecomunicación



Departamento de Ingeniería de  
Computadores, Automática y Robótica

Granada, a 12 de junio de 2025

---

Yo, **D. Francisco Javier Luque Hernández**, alumno de la titulación **Doble Grado en Ingeniería Informática y Administración y Dirección de Empresas de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con **DNI 45925414A**, autorizo la ubicación de la siguiente copia de mi Proyecto Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo.: Francisco Javier Luque Hernández  
Granada, a 12 de junio de 2025

---

**Yo, D. Francisco Javier Luque Hernández, con DNI 45925414A, estudiante del Doble Grado en Ingeniería Informática y Administración y Dirección de Empresas de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada,**

**DECLARO**

**La originalidad del trabajo presentado y, en su caso, la veracidad de los méritos y evidencias alegados** en el mismo sin perjuicio de la posible comprobación, si procede, por parte de los tutores del Proyecto Fin de Grado.

Fdo.: Francisco Javier Luque Hernández  
Granada, a 12 de junio de 2025

---

D. **Pablo García Sanchez**, Profesor Titular del Departamento de Ingeniería de Computadores, Automática y Robótica de la Universidad de Granada.

D. **Sergio Iván Aquino Britez**, estudiante de doctorado de Tecnología de la Información y Comunicación.

**Informan:**

Que el presente proyecto, titulado ***Comparativa de consumo energético y calidad de soluciones en algoritmos evolutivos***, ha sido realizado bajo su supervisión por **Francisco Javier Luque Hernández**, y autorizan la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada, a 12 de junio de 2025.

**Los directores:**

**D. Pablo García Sánchez**

**D. Sergio Iván Aquino Britez**

## AGRADECIMIENTOS

Habiendo llegado al final de esta etapa tan especial y cargada de especiales recuerdos y experiencias con la presentación de este Proyecto Fin de Grado en Ingeniería Informática como parte de mi Doble Grado en Ingeniería Informática y Administración y Dirección de Empresas, quiero expresar mi más profundo agradecimiento a todas aquellas personas que han estado a mi lado a lo largo de este proceso.

A mis padres, Inmaculada y Francisco Javier, por ser el pilar al que aferrarme cuando las cosas han tambaleado y el impulso necesario para seguir adelante. Gracias por enseñarme que la constancia y el esfuerzo dan fruto, por celebrar mis éxitos, por sostenerme cuando aparecían las dudas, y, sobre todo, por enseñarme a creer en mí mismo.

A mis hermanos, Alberto y Alejandro, por ayudarme a buscar siempre mi superación personal, a no conformarme con poco y hacerme entender que las adversidades que se presentan no son más que un motivo y una gran oportunidad para crecer y mejorar.

A mi novia, María Teresa, por sus palabras de aliento y tranquilidad cuando las cosas no parecían seguir el rumbo que debieran seguir. Por su acompañamiento y apoyo incondicional en las innumerables horas frente al ordenador, escuchar mis frustraciones cuando parecía que no iba a lograr mis metas, y por celebrar siempre cada logro alcanzado.

A mis tutores, Pablo y Sergio, quienes han sido guías excepcionales durante todo el proceso de investigación para realizar este proyecto. Su dedicación, experiencia y paciencia han sido esenciales para transformar una idea inicial en este proyecto. Gracias por sus valiosas aportaciones, por resolver mis dudas con tan buena disposición y por confiar en mis capacidades desde el primer momento.

A Dios, por ser el faro que ha guiado los pasos que he dado para llegar hasta el final de esta etapa. Por enseñarme que, desde la humildad y la entrega desinteresada a los demás, se pueden alcanzar grandes metas, y que, con fe, esfuerzo y una confianza plena en Él, todo es posible.

Por último, a todas aquellas personas, amigos y conocidos, que de algún modo me han ayudado a culminar esta maravillosa etapa de mi vida, mi más sincero agradecimiento.

# Índice

<b>1. Introducción.....</b>	<b>11</b>
1.1. Contexto y motivación.....	11
1.2. Objetivos.....	12
1.3. Estructura del documento.....	12
<b>2. Revisión del estado del arte.....</b>	<b>13</b>
2.1. Descripción del dominio del problema.....	13
2.2. Metodologías potenciales a aplicar.....	14
2.2.1. Formulación multi-objetivo.....	14
2.2.2. Optimización mono-objetivo con penalización o restricción.....	15
2.2.3. Ajuste dinámico durante la ejecución.....	15
2.2.4. Implementaciones eficientes y elección del lenguaje.....	15
2.2.5. Modelos sustitutivos y reducción de evaluaciones.....	15
2.3. Tecnologías potenciales para usar.....	16
2.4. Trabajos relacionados.....	17
<b>3. Metodología.....</b>	<b>19</b>
3.1. Selección y justificación de los problemas <i>benchmark</i> .....	19
3.2. Selección y justificación de los <i>frameworks</i> utilizados.....	20
3.2.1. DEAP.....	21
3.2.2. Inspyred.....	21
3.2.3. ECJ.....	22
3.2.4. ParadisEO.....	22
3.3. Configuración del algoritmo genético y plan experimental.....	23
3.4. Entornos de ejecución e instrumentación energética.....	25
<b>4. Resultados experimentales.....</b>	<b>27</b>
4.1. OneMax.....	27
4.1.1. Consumo energético en el portátil.....	27
4.1.2. Emisiones totales de CO <sub>2</sub> en el portátil.....	29
4.1.3. Evolución del <i>fitness</i> en el portátil.....	30
4.1.4. Consumo energético y emisiones en servidor.....	33
4.1.5. Emisiones totales de CO <sub>2</sub> en servidor.....	35
4.1.6. Evolución del <i>fitness</i> en servidor.....	37
4.1.7. Consumo energético en el portátil frente al servidor.....	39
4.1.8. Emisiones totales de CO <sub>2</sub> en el portátil frente al servidor.....	41
4.1.9. Análisis la evolución del <i>fitness</i> en el portátil frente al servidor.....	43
4.1.10. Análisis de la eficiencia energética en el portátil.....	44
4.1.11. Análisis de la eficiencia energética en el servidor.....	46
4.1.12. Análisis de la eficiencia energética del portátil frente al servidor.....	48
4.1.13. Distribución de eficiencia energética y de variación de <i>fitness</i> .....	49
4.2. Sphere.....	51

4.2.1. Consumo energético en el portátil.....	51
4.2.2. Emisiones totales de CO <sub>2</sub> en el portátil.....	53
4.2.3. Evolución del <i>fitness</i> en el portátil.....	54
4.2.4. Consumo energético en servidor.....	57
4.2.5. Emisiones totales de CO <sub>2</sub> en servidor.....	59
4.2.6. Evolución del <i>fitness</i> en servidor.....	61
4.2.7. Consumo energético en el portátil frente al servidor.....	65
4.2.8. Emisiones totales de CO <sub>2</sub> en el portátil frente al servidor.....	66
4.2.9. Análisis la evolución del <i>fitness</i> en el portátil frente al servidor.....	68
4.2.10. Análisis de la eficiencia energética en el portátil.....	70
4.2.11. Análisis de la eficiencia energética en el servidor.....	72
4.2.12. Análisis de la eficiencia energética del portátil frente al servidor.....	73
4.2.13. Distribución de eficiencia energética y de variación de <i>fitness</i> .....	75
4.3. Rosenbrock.....	76
4.3.1. Consumo energético en el portátil.....	76
4.3.2. Emisiones totales de CO <sub>2</sub> en el portátil.....	78
4.3.3. Evolución del <i>fitness</i> en el portátil.....	80
4.3.4. Consumo energético en servidor.....	83
4.3.5. Emisiones totales de CO <sub>2</sub> en servidor.....	85
4.3.6. Evolución del <i>fitness</i> en servidor.....	87
4.3.7. Consumo energético en el portátil frente al servidor.....	91
4.3.8. Emisiones totales de CO <sub>2</sub> en el portátil frente al servidor.....	93
4.3.9. Análisis la evolución del <i>fitness</i> en el portátil frente al servidor.....	95
4.3.10. Análisis de la eficiencia energética en el portátil.....	97
4.3.11. Análisis de la eficiencia energética en el servidor.....	99
4.3.12. Análisis de la eficiencia energética del portátil frente al servidor.....	101
4.3.13. Distribución de eficiencia energética y de variación de <i>fitness</i> .....	104
4.4. Schwefel.....	106
4.4.1. Consumo energético en el portátil.....	106
4.4.2. Emisiones totales de CO <sub>2</sub> en el portátil.....	108
4.4.3. Evolución del <i>fitness</i> en el portátil.....	110
4.4.4. Consumo energético en servidor.....	113
4.4.5. Emisiones totales de CO <sub>2</sub> en servidor.....	115
4.4.6. Evolución del <i>fitness</i> en servidor.....	117
4.4.7. Consumo energético en el portátil frente al servidor.....	121
4.4.8. Emisiones totales de CO <sub>2</sub> en el portátil frente al servidor.....	123
4.4.9. Análisis la evolución del <i>fitness</i> en el portátil frente al servidor.....	125
4.4.10. Análisis de la eficiencia energética en el portátil.....	127
4.4.11. Análisis de la eficiencia energética en el servidor.....	129
4.4.12. Análisis de la eficiencia energética del portátil frente al servidor.....	131
4.4.13. Distribución de eficiencia energética y de variación de <i>fitness</i> .....	134

<b>5. Conclusiones.....</b>	<b>137</b>
5.1. Revisión de objetivos, hipótesis y alcance.....	137
5.2. Hallazgos cuantitativos.....	138
5.3. Comparativa con la literatura.....	146
5.4. Limitaciones.....	147
5.5. Contribuciones.....	147
5.6. Síntesis final.....	149
5.6.1. Hallazgos principales.....	149
5.6.2. Validación empírica de hipótesis.....	149
5.6.3. Cuantificación del impacto energético.....	150
5.6.4. La métrica $\eta$ como muestra de la eficiencia.....	150
5.6.5. Robustez de los hallazgos.....	150
<b>6. Trabajos futuros.....</b>	<b>151</b>
6.1. Funciones de coste elevado.....	151
6.2. Uso de GPU.....	151
6.3. Criterios de parada multi objetivo.....	152
6.4. Estrategias adaptativas en tiempo real.....	152
<b>7. Referencias.....</b>	<b>154</b>
<b>8. Anexos.....</b>	<b>158</b>
8.1. Anexo I: Glosario de términos.....	158
8.2. Anexo II: Implementaciones.....	164
8.2.1. OneMax.....	164
8.2.1.1. DEAP.....	164
8.2.1.2. Inspyred.....	167
8.2.1.3. ECJ.....	169
8.2.1.4. ParadisEO.....	171
8.2.2. Sphere.....	174
8.2.2.1. DEAP.....	174
8.2.2.2. Inspyred.....	177
8.2.2.3. ECJ.....	181
8.2.2.4. ParadisEO.....	184
8.2.3. Rosenbrock.....	187
8.2.3.1. DEAP.....	187
8.2.3.2. Inspyred.....	190
8.2.3.3. ECJ.....	193
8.2.3.4. ParadisEO.....	195
8.2.4. Schwefel.....	198
8.2.4.1. DEAP.....	198
8.2.4.2. Inspyred.....	201
8.2.4.3. ECJ.....	204
8.2.4.4. ParadisEO.....	207

# **Comparativa de consumo energético y calidad de soluciones en algoritmos evolutivos**

Francisco Javier Luque Hernández

## **Palabras clave:**

Eficiencia energética; algoritmos evolutivos; *frameworks*; DEAP; ECJ; ParadisEO; Inspyred; OneMax; Sphere; Rosenbrock; Schwefel; *green computing*.

## **Resumen:**

Los algoritmos evolutivos se utilizan de forma intensiva para resolver problemas de optimización. Este Proyecto Fin de Grado compara, desde una perspectiva medioambiental, la ejecución de cuatro *frameworks* populares (ParadisEO (C++), ECJ (Java), DEAP e Inspyred (Python)) sobre dos arquitecturas opuestas: un portátil (procesador Intel i7-7500U) y un servidor (procesador Intel i9-12900KF). El estudio sigue un diseño que combina tres tamaños de población ( $2^6$ ,  $2^{10}$ ,  $2^{14}$  individuos) y tres probabilidades de cruce (0,01; 0,20; 0,80) aplicadas a cuatro *benckmarks* (OneMax, Sphere, Rosenbrock y Schwefel). Cada configuración se repite diez veces, cada una durante dos minutos.

El consumo eléctrico se mide con contadores RAPL y se convierte a emisiones de CO<sub>2</sub> mediante CodeCarbon; la productividad algorítmica se expresa con la métrica unificada  $\eta = \text{fitness}/\text{kWh}$ . Los resultados muestran que el servidor acelera el número de generaciones en un factor en torno a 2,5 pero multiplica el consumo entre 4 y 7 más, de modo que la eficiencia energética del portátil es, de media, cinco veces mayor. Las conclusiones confirman que la potencia de cómputo no garantiza sostenibilidad y que el tamaño de población es la palanca clave para equilibrar calidad y energía.

El trabajo aporta una matriz experimental diferente, la implementación homogénea de  $\eta$  en cuatro *frameworks*. Además, propone líneas futuras de investigación que incluyen, entre otras, el uso de GPU o criterios de parada multi-objetivo.

# **Comparison of energy consumption and solution quality in evolutionary algorithms**

Francisco Javier Luque Hernández

## **Keywords:**

Energy efficiency; evolutionary algorithms; frameworks; DEAP; ECJ; ParadisEO; Inspyred; OneMax; Sphere; Rosenbrock; Schwefel; green computing.

## **Abstract:**

Evolutionary algorithms are extensively used to solve optimization problems. This Final Degree Project compares, from an environmental perspective, the execution of four popular frameworks (ParadisEO (C++), ECJ (Java), DEAP and Inspyred (Python)) on two opposing architectures: a laptop (Intel i7-7500U processor) and a server (Intel i9-12900KF processor). The study follows a design that combines three population sizes ( $2^6$ ,  $2^{10}$ ,  $2^{14}$  individuals) and three crossover probabilities (0.01; 0.20; 0.80) applied to four benchmarks (OneMax, Sphere, Rosenbrock and Schwefel). Each configuration is repeated ten times, each one for two minutes.

Electrical consumption is measured with RAPL counters and converted to CO<sub>2</sub> emissions using CodeCarbon; algorithmic productivity is expressed with the unified metric  $\eta = \text{fitness}/\text{kWh}$ . The results show that the server accelerates the number of generations by a factor of around 2.5 but multiplies consumption between 4 and 7 times more, so that the laptop's energy efficiency is, on average, five times higher. The conclusions confirm that computing power does not guarantee sustainability and that population size is the key lever for balancing quality and energy.

The work contributes a different experimental matrix, the homogeneous implementation of  $\eta$  across four frameworks. Additionally, it proposes future research lines that include, among others, the use of GPU or multi-objective stopping criteria.

# 1. Introducción

La mejora constante del rendimiento ha sido, durante décadas, el motor principal de la investigación en algoritmos evolutivos (AE). Los trabajos se han evaluado, normalmente, con dos métricas: la calidad de la solución y el tiempo de convergencia. Sin embargo, el contexto tecnológico y social ha cambiado. Los centros de datos y las instalaciones de computación de alto rendimiento contribuyen de manera sustancial al cambio climático, con 100 megatonnes de emisiones de CO<sub>2</sub> por año, similar a la aviación comercial estadounidense (Lannelongue *et al.*, 2021, p. 1).

Aunque los AE no son tan masivos como los modelos de lenguaje de gran tamaño, se emplean de forma intensiva en optimización de redes, ajuste de hiperparámetros, diseño de fármacos y simulación industrial. Ejecutar miles de ejecuciones sin medir la energía que exigen no sólo encarece los experimentos, sino que puede contradecir los objetivos de sostenibilidad que muchas universidades y empresas han empezado a fijar.

Los dispositivos portátiles como Raspberry Pi y tablets requieren un orden de magnitud menos energía para ejecutar algoritmos evolutivos en comparación con computadoras estándar como laptops e iMacs, mientras que el tamaño de población también influye significativamente en los requerimientos energéticos (Fernández de Vega *et al.*, 2016, pp. 553-554). A la luz de estos datos, resulta imprescindible replantear la forma en que se diseñan, miden y comparan algoritmos evolutivos: la eficiencia energética debe pasar a ser una variable de primer orden, junto al tiempo y la precisión.

## 1.1. Contexto y motivación

Con las demandas computacionales y energéticas de los métodos modernos de *machine learning* creciendo exponencialmente, los sistemas de ML tienen el potencial de contribuir significativamente a las emisiones de carbono, aunque la mayoría de papers de investigación en ML no reportan regularmente métricas de energía o emisiones de carbono (Henderson *et al.*, 2020, p. 1).

Dentro de este marco, los algoritmos evolutivos presentan dos características que empeoran el problema: su naturaleza estocástica obliga a realizar muchas réplicas para obtener resultados robustos, y la exploración de hiperparámetros (población, cruce, mutación) multiplica el número de ejecuciones. Además, en los últimos años ha crecido la tentación de usar en todo momento hardware cada vez más potente sin preguntarse, normalmente, cuánta energía adicional cuesta ese salto. La motivación principal de este Proyecto Fin de Grado contribuir a dar una respuesta a esa pregunta ofreciendo a la comunidad científica datos comparables para decidir cuándo conviene escalar, cuándo basta con un equipo ligero y qué configuraciones internas maximizan la relación entre solución obtenida y vatios consumidos.

Así, la aportación es doble: académica, porque introduce la variable energética en la evaluación rigurosa de los AE, y práctica, porque ayuda a reducir costes y emisiones sin sacrificar el rendimiento algorítmico.

## 1.2. Objetivos

El objetivo general de este Proyecto Fin de Grado es cuantificar la eficiencia energética de varios *benchmarks* usando diferentes *frameworks* de algoritmos evolutivos en hardware de distinta escala y extraer recomendaciones que equilibren calidad, tiempo y consumo. Para alcanzarlo se plantean cuatro metas específicas, cada una con su pregunta subyacente:

- Medición comparativa en dos arquitecturas: ¿hasta qué punto varía el consumo cuando el mismo algoritmo pasa de un portátil a un servidor? Se analizarán ParadisEO (C++), ECJ (Java), DEAP e Inspyred (ambos en Python) ejecutados durante un tiempo fijo en un portátil con un procesador Intel i7-7500U y en un servidor con un procesador Intel i9-12900KF.
- Definición y validación de la métrica  $\eta = \text{fitness}/\text{kWh}$ : ¿es posible condensar en un único valor la relación entre solución encontrada y energía gastada? La métrica se aplicará de forma homogénea a todos los ensayos para facilitar la comparación cruzada.
- Estudio del impacto de la configuración interna: ¿cómo influyen el tamaño de población y la probabilidad de cruce en la eficiencia? Se cubrirá un rango de tres tamaños ( $2^6, 2^{10}, 2^{14}$ ) y tres tasas de cruce (0,01; 0,20; 0,80), con diez ejecuciones cada una, para encontrar tendencias estables usando la media de esas 10 ejecuciones.

## 1.3. Estructura del documento

El documento se ha organizado para conducir al lector desde los fundamentos hasta las conclusiones prácticas:

- El capítulo 2 revisa la literatura sobre consumo energético en algoritmos evolutivos y técnicas de medición, situando el proyecto dentro del estado del arte.
- El capítulo 3 describe la metodología: hardware, software, medidor de emisiones y consumo, métrica  $\eta$  y diseño factorial del experimento.
- El capítulo 4 presenta los resultados y su análisis estadístico, incluyendo gráficos de consumo, *fitness* y eficiencia.
- El capítulo 5 sintetiza las conclusiones, discute su relación con trabajos previos y detalla las aportaciones originales.
- El capítulo 6 expone las líneas de investigación futura, como la inclusión de GPU o criterios de parada multi-objetivo.
- El capítulo 7 reúne las referencias en las que se sustenta la teoría aportada en este proyecto.
- El capítulo 8 contiene dos anexos, siendo el primero un glosario de términos y el segundo una explicación de las implementaciones de los benchmarks.

## **2. Revisión del estado del arte**

Antes de entrar en la metodología experimental, es necesario situar el problema dentro de la literatura reciente. Este capítulo expone, en primer lugar, los trabajos que han medido el consumo energético de algoritmos evolutivos en distintos contextos, desde dispositivos de bajo consumo hasta clústeres de alto rendimiento (Fernández de Vega *et al.*, 2016, pp. 551-552). A continuación, se resumen los avances en métricas y herramientas de monitorización que permiten medir y estimar la huella de carbono, destacando propuestas como los “*carbon impact statements*” para experimentos de *machine learning* (Henderson *et al.*, 2020, p. 7) y la calculadora *Green Algorithms* que considera la intensidad de carbono según la ubicación geográfica para estimar la huella de carbono computacional (Lannelongue *et al.*, 2021, pp. 1-2). Por último, se revisan las estrategias que la comunidad ha planteado para reducir el consumo, desde ajustes dinámicos de población y frecuencia de la CPU hasta el uso de aceleradores GPU y criterios de parada multi-objetivo.

### **2.1. Descripción del dominio del problema**

El impacto energético de la computación ha dado origen al concepto de green computing, término que surgió en la última década para enfocar adecuadamente el problema del consumo energético asociado a la computación, particularmente en las grandes instalaciones como centros de datos (Fernández de Vega *et al.*, 2016, p. 368). Este creciente interés es especialmente potente en infraestructuras a gran escala, como centros de datos y nubes, donde el aumento de recursos físicos conlleva un mayor consumo energético (Maryam *et al.*, 2018, p. 1).

Dentro de este contexto, los algoritmos evolutivos destacan por su versatilidad y la variabilidad de su demanda energética al poder ser configurados en función de las diferentes necesidades para las que su uso suponga una solución. Ensayos *cross-platform* han mostrado diferencias de un orden de magnitud en la energía usada para alcanzar soluciones cuando se comparan dispositivos portátiles con computadoras estándar y sistemas blade (Fernández de Vega *et al.*, 2016, pp. 553-554). Estas evidencias han llevado a desarrollar dos destacables líneas que mantienen una estrecha simbiosis en este ámbito: cuantificar de forma sistemática el consumo y diseñar metaheurísticas que incluyan la energía como criterio de optimización de forma explícita (Cotta y Martínez-Cruz, 2025, p. 237).

Esta cuantificación requiere un uso de herramientas adecuadas para ello, como es el caso de bibliotecas de uso general como CodeCarbon o contadores de bajo nivel (Intel RAPL o EnergiBridge, entre otros), los cuales permiten obtener trazas y mediciones de potencia y emisiones, teniendo en cuenta que la exactitud depende de que la porción instrumental represente una fracción significativa del tiempo de CPU; de lo contrario, el “ruido” del sistema puede ocultar diferencias sutiles. consecuentemente, gran parte de la literatura recomienda implementar funciones de prueba de coste moderado (como es el caso de las usadas en este Proyecto Fin de Grado: OneMax, Sphere, Rosenbrock y Schwefel) y fijar un

número máximo de evaluaciones para comparar algoritmos y configuraciones de manera reproducible (Cotta y Martínez-Cruz, 2025, p.231).

Diversos estudios previos ilustran la repercusión real que los AE pueden tener sobre la eficiencia energética:

- Carga óptima de enfriadoras (*chillers*): un algoritmo de *Differential Evolution* identificó combinaciones de parcialización que, en cargas bajas (40%), reducen el consumo energético hasta un 23.4% frente al método lagrangiano (Lee *et al.*, 2011, p. 605).
- Centros de datos en la nube: repertorios de AE (GA, PSO y ACO) asignan máquinas virtuales buscando reducir al máximo posible la energía sin violar SLA, con unas reducciones entorno al 44 % o 60 % respecto a políticas deterministas (Matyam *et al.*, 2018, p. 8).
- Minería *blockchain*: el problema de selección de mineros en *blockchain* puede reformularse como un problema de optimización que balancea el consumo energético con la confianza y descentralización del sistema (Alofi *et al.*, 2022, p. 913).

En suma, el dominio del problema a tratar en este Proyecto Fin de Grado se sitúa en el centro de la intersección entre optimización evolutiva y la eficiencia energética: no es suficiente con tener unas soluciones matemáticamente aceptables y deseables, sino que habrá que hacerlo persiguiendo el mínimo impacto medioambiental posible.

## 2.2. Metodologías potenciales a aplicar

Para intentar abordar, de una manera rigurosa, la dualidad calidad de la solución ↔ coste energético existen diferentes líneas metodológicas que, combinadas, permiten diseñar experimentos reproducibles y algoritmos evolutivos energy-aware, de entre las que cabe destacar:

### 2.2.1. Formulación multi-objetivo.

La estrategia más directa consiste en incorporar la energía consumida (o las emisiones de CO<sub>2</sub>) como objetivo adicional junto al fitness buscado (bien sea el máximo o bien sea el mínimo). Algoritmos como NSGA-II, SPEA2, IBEA o NSGA-III facilitan la búsqueda de fronteras de Pareto que revelan los intercambios entre los criterios usados (Alofi *et al.*, 2022, pp. 913-916). Esto es especialmente útil cuando el impacto energético y la calidad de la solución son magnitudes medibles en conflicto (por ejemplo, en la selección de mineros *blockchain* o en la planificación de tareas en la nube) y permite al decisor elegir soluciones “conscientes” de dicho impacto.

### **2.2.2. Optimización mono-objetivo con penalización o restricción.**

Cuando se trata de mantener un único objetivo, la energía puede introducirse como un criterio para la terminación del algoritmo o como criterio de penalización. Los estudios sobre sistemas HVAC han demostrado que la programación evolutiva puede lograr ahorros energéticos significativos mediante la optimización de parámetros operativos. El enfoque de simulación-optimización utilizando programación evolutiva mostró un potencial de ahorro de aproximadamente 7% comparado con las configuraciones operacionales existentes (Fong *et al.*, 2006, p. 11).

### **2.2.3. Ajuste dinámico durante la ejecución.**

Además de la configuración inicial del problema, ciertos parámetros pueden ajustarse de manera dinámica. Se ha observado que el tamaño de población puede influir paradójicamente en el consumo energético, donde poblaciones más grandes ocasionalmente requieren menos energía total debido a ajustes automáticos en la frecuencia del procesador (Fernández de Vega *et al.*, 2016, p. 555). Los compiladores modernos pueden aplicar optimizaciones dinámicas que van desde la eliminación de código no ejecutado hasta la combinación compleja de secuencias de instrucciones y bucles, mejorando el rendimiento y reduciendo el consumo energético de manera que no es obvio anticipar solo observando el código fuente (Merelo-Guervós *et al.*, 2025, p. 241).

### **2.2.4. Implementaciones eficientes y elección del lenguaje.**

El mismo algoritmo puede representar diferencias energéticas notables según el lenguaje y las buenas prácticas de programación. Medir las funciones *BBOB* en C++ evidenció que el uso de estructuras de datos de tamaño fijo como arrays consume aproximadamente un octavo de la energía comparado con estructuras de tamaño variable como vectores (Merelo-Guervós *et al.*, 2025, p. 246). Además, experimentos con transprecisión computing utilizando formatos personalizados de 8 y 16 bits redujeron el consumo energético hasta un 30% (Merelo-Guervós *et al.*, 2025, p. 242).

### **2.2.5. Modelos sustitutivos y reducción de evaluaciones.**

El empleo de *surrogate models*, como redes neuronales o *Gaussian Processes*, para aproximar funciones de coste elevado disminuye el número de evaluaciones reales y, por ende, la energía. En la optimización de cachés para sistemas empotados, una combinación de NSGA-II con el simulador Dinero IV logró recortes medios del 92 % en kWh respecto a la búsqueda exhaustiva (Díaz Álvarez *et al.*, 2016, p. 200).

En conjunto, estas metodologías proporcionan un marco flexible que permite combinar técnicas de optimización, medición y análisis estadístico para avanzar hacia algoritmos evolutivos sostenibles.

## 2.3. Tecnologías potenciales para usar

La evaluación conjunta de calidad de solución y consumo energético conlleva a tener en consideración tanto la propia programación del problema como la infraestructura de medida y ejecución.

De entre los entornos más citados para investigación reproducible caben destacar DEAP e Inspyred (Python), ECJ (Java) y ParadisEO (C++) dado que son los seleccionados para la elaboración de este Proyecto Fin de Grado. Su idoneidad se debe a que exponen operadores genéticos altamente configurables y facilitan la integración con bibliotecas externas de monitorización. No obstante, el lenguaje subyacente influye de manera crítica en la energía consumida: los resultados muestran que no existe una categoría clara de lenguajes superiores, ya que el rendimiento varía según la operación específica, aunque Java casi siempre se encuentra entre los más rápidos junto con C o Go (Merelo-Guervós *et al.*, 2017, p. 700). Los experimentos revelan que lenguajes interpretados como PHP pueden superar a C++ en ciertas operaciones, difuminando la distinción entre lenguajes compilados rápidos y lenguajes de scripting lentos (Merelo-Guervós *et al.*, 2016, p. 1607). Esta disparidad justifica la elección para este PFG de combinar varios lenguajes de programación para contrastar su impacto energético.

En el plano del *monitoring* se pueden diferenciar dos tipos de elecciones:

- a) Contadores de hardware, como los sensores RAPL, ofrecen mediciones de consumo energético a nivel de paquete (PKG) que incluye CPU y memoria, con una precisión que puede ser de alrededor del 5% (Merelo-Guervós *et al.*, 2025, p. 251).
- b) Bibliotecas de usuario, de las que Codecarbon destaca por aportar, además del consumo en kWh, la conversión directa a kg CO<sub>2</sub> y factores de región. Su integración en Python es trivial y puede extenderse a otros lenguajes mediante *wrappers* o scripts. El propio método empleado por Merelo-Guervós *et al.* (2025) para las funciones *BBOB* demuestra que utilizando la herramienta *pinpoint*, que accede a la API RAPL, es posible obtener mediciones consistentes del consumo energético del sistema mientras un proceso está ejecutándose (p. 243).

Cuando los AE se ejecutan en infraestructuras compartidas, simuladores como CloudSim son ampliamente utilizados como herramienta de simulación para evaluar algoritmos de consolidación de máquinas virtuales y optimización energética (Maryam *et al.*, 2018). Estos entornos, además, proporcionan entornos controlados donde repetir las pruebas con los mismos patrones de carga.

Para reducir el time-to-solution sin disparar desorbitadamente el consumo eléctrico, se aconseja explotar el paralelismo a nivel de población mediante frameworks de computación distribuida como MapReduce, los cuales permiten parallelizar la evaluación de individuos y la aplicación de operadores genéticos entre múltiples procesos, logrando mejoras significativas en el rendimiento para problemas de gran dimensión (Salto *et al.*, 2018, p. 266). En cambio, el análisis energético no debe considerar únicamente el tiempo de ejecución, sino también el

consumo energético durante la búsqueda de soluciones, ya que ambos elementos son necesarios para evaluar el costo total del algoritmo (Fernández de Vega *et al.*, 2016, p. 549). Los experimentos muestran que es posible reducir el consumo de potencia de las redes neuronales artificiales mediante técnicas de neuroevolución; se han observado reducciones de hasta un 29.2 % en el consumo de potencia manteniendo un rendimiento predictivo similar (Cortés *et al.*, 2024, pp. 76-87).

Finalmente, sintetizar cada experimento en contenedores Docker elimina las variaciones nacidas del *software stack*. Esta práctica se ha propuesto explícitamente para garantizar la reproducibilidad de los *benchmarks* de algoritmos evolutivos, facilitando que cualquier investigador descargue la imagen y repita las pruebas bajo la misma pila de software (Merelo-Guervós *et al.*, 2016, p. 1610).

## 2.4. Trabajos relacionados

El interés por una computación "*energy-proportional*" nace en el ámbito de los centros de datos y se extiende gradualmente a otros contextos computacionales. En el ámbito de los algoritmos evolutivos, Fernández de Vega *et al.* (2016) plantearon que el consumo energético debe considerarse junto al tiempo de cómputo cuando se busca una solución, no solo este último (p. 549). Estudios han demostrado que el tamaño de la población influye significativamente en el consumo energético, observándose una tendencia general de incremento del costo energético con poblaciones más grandes (Fernández de Vega *et al.*, 2016, pp. 554-555), fenómeno que se atribuye tanto a la arquitectura del hardware como a la gestión de la jerarquía de memoria (Fernández de Vega *et al.*, 2016, p. 374).

Como es lógico pensar, la cuantificación sistemática del consumo ha sido mejorada con el tiempo. Cotta y Martínez-Cruz (2025) analizaron cómo la ejecución de lotes altera el perfil térmico y, por ende, la potencia instantánea, subrayando la necesidad de protocolos experimentales reproducibles (pp. 227-233). En el plano micro, Merelo-Guervós *et al.* (2025) midieron la energía de las funciones *BBOB* implementadas en C++ para detectar “puntos calientes” a nivel de operador y precisión numérica (p. 241). Estas aportaciones han facilitado la adopción de bibliotecas como Codecarbon o contadores RAPL como estándares de facto.

Como se ha comentado anteriormente, la propia implementación también importa. Experimentos comparativos revelan notables diferencias de la misma implementación en diferentes lenguajes: un mismo algoritmo escrito en C++ puede consumir hasta un 30 % menos de energía la CPU que su versión en Python o Java (Merelo-Guervós *et al.*, 2016, p. 704). Esto confirma que la eficiencia no radica sólo en el algoritmo, sino en las capas de compilación, ejecución y gestión de memoria.

En cuanto a la optimización explícita de la energía, la literatura muestra una clara evolución desde aplicaciones puntuales a enfoques multi-objetivo. Fong *et al.* (2006) desarrollaron un enfoque de simulación-optimización utilizando programación evolutiva para la gestión energética efectiva de sistemas HVAC, el cual puede manejar eficientemente problemas de optimización discretos, no lineales y altamente restringidos (p. 220), mientras

que Lee *et al.* (2011) establecieron la carga óptima de enfriadoras mediante algoritmos Differential Evolution, obteniendo mejores soluciones promedio que PSO (0.8-2.4 % de mejora) y superando significativamente al algoritmo genético (pp. 604-605). Más recientemente, Essiet *et al.* (2019) proponen un algoritmo *Differential Evolution* multi-archivo para la gestión de demanda residencial, pudiendo, al mismo tiempo, reducir el coste y aumentar el confort del usuario (pp. 361-363), y Alofi *et al.* (2022) incorporan la energía como criterio en la optimización de redes *blockchain* (pp. 913-919).

En suma, las investigaciones previas dejan claro que:

- La medición fina del consumo es condición *sine qua non* para comparar configuraciones y hardware.
- Las decisiones de ingeniería pueden modificar radicalmente el impacto medioambiental.
- Existe una transición desde estudios descriptivos a modelos multi-objetivo donde la energía es tratada como un parámetro optimizable al mismo nivel que la calidad de la solución.

Sin embargo, faltan comparativas que combinen varios lenguajes, *frameworks* y problemas *benchmark* bajo las mismas condiciones de evaluación, el vacío que pretende cubrir este Proyecto Fin de Grado.

### 3. Metodología

En este apartado se describe en detalle la metodología experimental seguida para comparar tanto la calidad de las soluciones como el consumo energético de los algoritmos genéticos implementados en los diferentes lenguajes y *frameworks* escogidos para desarrollar este Proyecto Fin de Grado.

Primero se justifica la selección de los problemas *benchmark* (OneMax, Sphere, Rosenbrock y Schwefel) y se explican los parámetros que los hacen comparables. A continuación se presentan las configuraciones del algoritmo, definidas por tamaños de población, porcentajes de cruce y mutación y los criterios de parada, y se detalla el entorno de ejecución en las dos máquinas empleadas (portátil i7-7500U y servidor i9-12900KF).

#### 3.1. Selección y justificación de los problemas *benchmark*

La comparativa de la huella energética de diferentes implementaciones de algoritmos genéticos requiere un conjunto de problemas que abarque un considerable abanico de dificultad topológica, manteniendo un coste computacional moderado y sea ampliamente usado en la bibliografía de eficiencia energética (Cotta y Martínez-Cruz, 2025, p. 232).

Siguiendo estos criterios, las funciones seleccionadas incluyen OneMax, Sphere,, Rosenbrock y Schwefel, elegidas como una representación del conjunto completo de funciones *BBOB* desde el punto de vista del consumo energético (Merelo-Guervós *et al.*, 2025, p. 247).

La primera función usada es el clásico OneMax, el cual opera en el espacio binario  $\{0,1\}^n$ . Para un individuo  $b = (b_1, \dots, b_n)$  el problema consiste en maximizar

$$f(x) = \sum_{i=1}^n b_i$$

donde  $b_i$  representa el  $i$ -ésimo bit y  $n$  representa la longitud del cromosoma. Así, el *fitness* coincide con el número de unos. Gracias a que este problema posee un carácter unimodal, se facilita aislar el consumo debido a los operadores genéticos y es la referencia habitual para comparar lenguajes y *frameworks* (Merelo-Guervós *et al.*, 2016, pp. 1620-1622).

La segunda función empleada es Sphere, la que introduce el caso continuo más sencillo. Para un vector real  $x = (x_1, \dots, x_n)$  dentro del dominio  $[l, u]^n$  de minimiza

$$f(x) = \sum_{i=1}^n x_i^2$$

siendo  $n$  la dimensión y  $x_i$  la  $i$ -ésima coordenada. La principal ventaja del uso de esta función es su convexidad y separabilidad, por lo que permite aislar el impacto energético de los operadores de cruce y mutación sobre variables reales (Merelo-Guervós *et al.*, 2025, p. 247). El óptimo, por tanto, se alcanza en  $x = 0$  con  $f = 0$ .

La tercera función implementada es Rosenbrock, la cual introduce un nuevo factor a tener en cuenta, el cual es que no aparece la separabilidad que se comentaba en las dos anteriores funciones. En el mismo dominio que Sphere,  $[l, u]^n$ , se minimiza

$$f(x) = \sum_{i=1}^{n-1} \left[ 100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2 \right]$$

donde  $x_i$  y  $x_{i+1}$  están acoplados. El valle curvado que conduce al óptimo (1,...,1) fuerza al algoritmo a equilibrar exploración y afinamiento, aumentando tanto el tiempo como la energía consumida.

Por último, la cuarta función analizada es Schwefel, la cual se caracteriza por ser multimodal. Para  $x \in [l, u]^n$  se intenta minimizar

$$f(x) = 418,9829n - \sum_{i=1}^n x_i \sin(\sqrt{|x_i|})$$

donde  $x_i$  es la coordenada  $i$ . Su considerable número de óptimos locales produce picos de uso de CPU que activan mecanismos *turbo-boost*, apareciendo así diferencias de consumo entre arquitecturas.

Tabla 1: Resumen de *benchmarks* utilizados

Función	Tipo de variable	Dominio	Rasgo topológico	Motivo principal de inclusión
OneMax	Enteros	$\{0,1\}^n$	Unimodal	Patrón básico para GA binarios y termómetro de eficiencia de implementación.
Sphere	Reales	$[l, u]^n$	Separable	Aísla el coste energético de funciones de fitness de punto flotante (Merelo-Guervós <i>et al.</i> , 2025, p. 247).
Rosenbrock	Reales	$[l, u]^n$	Valle curvado, no separable	Revela el impacto energético del acoplamiento entre variables (Cotta y Martínez-Cruz, 2025, p. 232).
Schwefel	Reales	$[l, u]^n$	Multimodal	Estresa la diversificación y expone picos de consumo (Merelo-Guervós <i>et al.</i> , 2025, p. 247).

Fuente: elaboración propia

### 3.2. Selección y justificación de los *frameworks* utilizados

La elección de *frameworks* determina tanto el rendimiento energético, dado que cada lenguaje y biblioteca compila e interpreta el código de manera diferente, como la reproducibilidad de los experimentos. Para este Proyecto Fin de Grado se han escogido cuatro entornos que

resultan idóneos para la cuestión que se aborda y que figuran entre los más citados en la literatura sobre eficiencia y algoritmos evolutivos.

### 3.2.1. DEAP

Este *framework* se estableció como uno de los pilares de este estudio porque combina simplicidad con transparencia algorítmica. Su diseño basado en "toolboxes" permite que los algoritmos sean explícitos y las estructuras de datos transparentes, facilitando la implementación de cualquier tipo de algoritmo evolutivo, como destacan Fortin *et al.* (2012, pp. 2171-2172). Esta brevedad gana importancia cuando se añade una medición energética: encapsular el bucle evolutivo con CodeCarbon requiere instanciar simplemente un *tracker* antes de eaSimple, con lo que la sobrecarga temporal es prácticamente nula y la medición del consumo se aúna en el código sin alterar la lógica del algoritmo.

Desde la perspectiva del consumo, se dice que Python introduce cierta penalización frente al código nativo, pero se matiza que el impacto real depende de la forma de manipulación de los datos. En el análisis del factor de mejora de Banijamali (2024, pp. 36-37), Codon demostró mejoras considerables en el consumo energético comparado con Python, aunque C++ generalmente mantuvo un rendimiento superior en la mayoría de las métricas evaluadas, eliminando así la penalización energética que suele asociarse a CPython (Stoico *et al.*, 2025, p. 6).

Por último, DEAP ofrece una trazabilidad completa en las mediciones y métricas usadas, lo que satisface la recomendación de “medición integral” que establece la guía *Green AI* para algoritmos evolutivos (Merelo-Guervós *et al.*, 2025, p. 241).

### 3.2.2. Inspyred

Elegir Inspyred como el segundo *framework* de Python tiene su razón en poder contrastar si la eficiencia energética depende, además del lenguaje, de la carga estructural que añade cada biblioteca. A diferencia de DEAP, Inspyred está desarrollado con un sistema de “observadores” que permite al programador utilizar una lógica propia en puntos concretos del ciclo. Esta característica implica menos objetos y, por tanto, menos presión sobre el medidor, tal y como muestran Los experimentos multilenguaje de Merelo-Guervós *et al.* (2016), dado que aunque los lenguajes compilados como Java y C generalmente obtienen el mejor rendimiento, algunos lenguajes interpretados como Python pueden alcanzar velocidades competitivas en ciertas operaciones específicas (p. 38).

Así, Inspyred favorece la reproducibilidad. Esto se debe a que, al exponer el generador de números aleatorios y las listas de operadores como parámetros de la propia configuración, el código resultante deja evidencias de las decisiones experimentales, siguiendo el principio de "*implementation matters*" establecido por Merelo-Guervós *et al.* (2016), donde se demuestra que prestar atención a la forma particular de implementar un algoritmo puede resultar en mejoras de velocidad que superan las obtenidas usando el lenguaje a priori más rápido (p. 28).

Desde el punto de vista del consumo, Inspyred facilita pruebas donde la sobre-capa de la biblioteca es casi inexistente. Cotta y Martínez-Cruz (2025) demostraron que la introducción de pausas cortas entre ejecuciones puede reducir el consumo energético hasta un 9% en el *benchmark* considerado (p. 227).

En síntesis, Inspyred ofrece el contrapunto minimalista de DEAP: mismo lenguaje, misma instrumentación, pero una arquitectura más ligera, lo que permite analizar si “menos código” implica necesariamente “menos julios”.

### 3.2.3. ECJ

Se incorpora ECJ a este proyecto por su ecosistema de *byte-code* compilado sobre la JVM, un punto intermedio entre el código interpretado de Python y el binario nativo de C++. Su trayectoria de más de dos décadas como uno de los *toolkits* más usados en investigación evolutiva y la arquitectura ortogonal que ofrece para combinar operadores mono- y multi-objetivo, coevolución y aprendizaje automático lo sitúan como estándar de facto dentro del universo Java (Scott y Luke, 2019, pp. 1391-1392).

Cuando algoritmos evolutivos se ejecutan en dispositivos de bajo consumo como Raspberry Pi o tabletas, la energía total necesaria es aproximadamente un orden de magnitud menor que en portátiles o equipos de sobremesa (Fernández de Vega *et al.*, 2016, pp. 553-554). Segundo, Cotta y Martínez-Cruz (2025) constataron que Java consume menos energía que C++ en dimensionalidades menores, aunque este efecto se revierte conforme aumenta la dimensionalidad debido al uso más intensivo del recolector de basura (p. 236).

Por último, ECJ permite estudiar la influencia de la JVM en la sostenibilidad. Al incluir a ECJ, este proyecto podrá verificar si ese ajuste dinámico sigue siendo ventajoso cuando el número de evaluaciones es similar y la calidad de la solución viene controlada por los mismos operadores genéticos.

### 3.2.4. ParadisEO

Este último *framework* utilizado representa el extremo de la eficiencia compilada de este PFG. Desarrollado originalmente por le grupo INRIA Lille, este *framework* desarrollado en C++ es de caja blanca basado en plantillas que separan de forma explícita los componentes del algoritmo evolutivo de los componentes dependientes del problema, con lo que el usuario apenas necesita implementar la representación y la función de evaluación mientras se reutiliza el resto de código (Liefooghe *et al.*, 2011, p. 108).

Desde el punto de vista del consumo energético, los trabajos comparativos de Alba *et al.* (2007) destacan que el tiempo (y, por ende, la energía de la CPU) requerido para ejecutar un algoritmo genético se reduce significativamente cuando la población se organiza como un array, en comparación con la versión que utiliza vectores en Java. El segundo grupo consume un 100% más de tiempo que su equivalente en el primero, lo que se traduce en un consumo energético aproximadamente del doble (p. 768). Este ahorro por instrucción se puede ampliar al activar la evaluación paralela de la población: ParadisEO permite distribuir las soluciones

entre hilos para evaluarlas en paralelo sin necesidad de modificar el código principal del algoritmo, logrando así una alta escalabilidad sin introducir la sobrecarga de colas de mensajes (Liefooghe *et al.*, 2011, p. 110), aunque esta paralelización no se utilice en este PFG.

Desde el punto de vista de la implementación, las decisiones de programación pueden tener un impacto significativo en el consumo energético. Cotta y Martínez-Cruz (2025) compararon implementaciones gemelas de la misma librería de algoritmos evolutivos en Java y C++, revelando que esta última escala mejor en términos de eficiencia energética y tiempo de ejecución en problemas de alta dimensionalidad. Para la implementación en C++ se emplearon punteros inteligentes (`std::unique_ptr` y `std::shared_ptr`) para la gestión automática de memoria del heap, evitando deliberadamente el uso de asignadores de memoria. Las mediciones de energía se realizaron utilizando la herramienta Intel Power Gadget, determinando la contribución real de cada algoritmo al restar el consumo basal del sistema de las mediciones durante la ejecución del algoritmo evolutivo (p. 232).

Además, la portabilidad del binario ofrece, a su vez, un escenario controlado: el mismo ejecutable se lanza en el portátil y en el servidor sin recompilar, de modo que cualquier diferencia de consumo procede únicamente de la arquitectura física y no de cambios en *flags* del compilador, una recomendación de Merelo-Guervós *et al.* (2025) al advertir de que cada optimización activada con `-O` o `-march` altera el perfil de potencia y debe mantenerse constante para comparar los diferentes lenguajes (pp. 244).

Por último, este framework mantiene ejemplos y contribuciones disponibles cuyo código se distribuye bajo la licencia libre CeCILL, lo que facilita la reproducibilidad y el intercambio de configuraciones (Liefooghe *et al.*, 2011, pp. 108, 111). Con ello se obtiene un punto de referencia del mejor caso en impacto energético: en un GA One-Max las versiones que usan estructuras de datos optimizadas (`arrays`) frente a las menos eficientes (`Vector`) pueden reducir el consumo energético aproximadamente a la mitad, ya que los experimentos demuestran que las implementaciones con `Vector` consumen más del 100% del tiempo necesario comparado con las implementaciones basadas en `arrays` (Alba *et al.*, 2007, p. 768).

Si las implementaciones Python o Java se acercan a esa marca, podrá afirmarse que la capa de interpretación ya no es el factor decisivo; si la brecha continúa, quedará demostrado el coste de la facilidad de uso frente a la eficiencia compilada, respaldando la hipótesis de "*no fast lunch*" para la implementación de problemas de optimización evolutiva (Merelo-Guervós *et al.*, 2017, p. 689).

### **3.3. Configuración del algoritmo genético y plan experimental**

Para poder comparar de una manera justa la calidad de las soluciones alcanzadas y el consumo energético de los cuatro *frameworks* empleados se ha fijado una configuración genética común en cada problema analizado, establecida como constantes al comienzo de cada script o fichero `.params`. Así, cualquier diferencia en el tiempo de ejecución será debida

exclusivamente al hardware, ya que las operaciones ejecutadas en alto nivel son exactamente las mismas (Fernández de Vega *et al.*, 2016, p. 370).

Cada ejecución la cual está siempre limitada a 2 minutos de tiempo máximo de ejecución, comienza con una población de tamaño  $N$ , el cual varía entre tres posibles tamaños:  $2^6$ ,  $2^{10}$ ,  $2^{14}$ . Los individuos de OneMax se representan como una cadena binaria de longitud  $n = 1024$ , mientras que en Sphere, Rosenbrock y Schwefel se emplea un vector de valores reales de longitud  $n = 1024$ . Este parámetro  $n$ , el cual representa el tamaño del individuo, es el mismo que puede identificarse en las fórmulas del apartado 3.1. Para garantizar una comparación justa entre lenguajes de programación, Cotta y Martínez-Cruz (2025) advirtieron sobre la necesidad de que las implementaciones consideradas sean lo más similares posible en términos lógicos, de manera que cualquier diferencia se deba a cuestiones profundamente arraigadas en el lenguaje de programación y no a decisiones de implementación de alto nivel (p. 230).

La fase de selección utiliza torneo binario sin reemplazo por dos principales motivos: mantiene una presión selectiva estable, independientemente de  $N$ , y minimiza las operaciones aritméticas necesarias para calcular probabilidades acumuladas. Los experimentos revelan que la metodología puede fallar cuando los compiladores avanzados examinan el código globalmente y generan el mejor código máquina posible, llegando incluso a combinar la función bajo medición con el bucle de generación de cromosomas, haciendo imposible separar el consumo energético de ambas operaciones (Merelo-Guervós *et al.*, 2025, p. 251).

Para introducir variabilidad controlada en las mediciones, en Sphere, Rosenbrock y Schwefel los padres seleccionados se recombinan mediante *Simulated Binary Crossover (SBX)* con un único índice de distribución  $\eta_c = 0,1$  y tres probabilidades de cruce,  $p_c = 0,01$ ,  $p_c = 0,2$  y  $p_c = 0,8$ , las cuales cubren desde un régimen casi asexual hasta una recombinación altamente intensa. La ontología MOODY de Aldana-Martín *et al.* (2024) detalla las reglas SWRL que validan configuraciones de parámetros como el índice de distribución del operador SBX y previenen su uso con operadores incompatibles como BLX\_Alpha, subrayando su papel para reducir el número de configuraciones que se estiman producen malos resultados y acelerar la búsqueda al evitar evaluaciones innecesarias (p. 8).

Tras esto, al descendiente generado se le aplica una mutación polinómica con la probabilidad por gen  $p_m = 1/L$  con el parámetro  $\eta_m$  constante también.

El enfoque adoptado utiliza la arquitectura generacional de ECJ, que proporciona subclases *EvolutionState* predeterminadas para métodos de optimización generacional, *steady-state* o *single-state (hill-climbing)*, permitiendo flexibilidad en la estrategia evolutiva implementada (Scott y Luke, 2019, p. 1392).

De igual forma, para controlar la variabilidad estocástica, cada configuración se ejecuta, de manera independiente, 10 veces, lo que hace un total de 90 ejecuciones (3 tamaños de población x 3 probabilidades de cruce x 10 ejecuciones) por cada *framework* y *benchmark*. Así, los datos analizados se obtienen de calcular la media de cada 10 ejecuciones de cada parámetro sometido a estudio.

### 3.4. Entornos de ejecución e instrumentación energética

Como se ha mencionado anteriormente, las ejecuciones de este proyecto se han realizado en dos máquinas diferentes, elegidas de manera intencionada: un ordenador portátil y un servidor. Esto se debe, en primer lugar, a que se permite observar la elasticidad energética del algoritmo genético frente a arquitecturas diferentes, dado que estas dos máquinas presentan unos perfiles térmicos distintos: el i7-7500U está diseñado para un TDP de 15 W<sup>1</sup> mientras que el i9-12900KF admite valores de más de 200 W<sup>2</sup> gracias a su disipación y separación entre núcleos de rendimiento y eficiencia. Esas diferencias pueden traducirse en consumos energéticos diferentes de un orden de magnitud entre dispositivos portátiles y computadoras estándar, aún ejecutando exactamente el mismo algoritmo (Fernández de Vega *et al.*, 2016, pp. 553-554).

Tabla 2: Resumen de las características de los entornos de ejecución

Portátil		Servidor
Modelo de CPU	Intel Core i7-7500U	Intel Core i9-12900KF
Arquitectura física	2 núcleos / 4 hilos	16 núcleos / 24 hilos
Frecuencia base / turbo	2,7 GHz / 3,5 GHz	3,2 GHz / 5,2 GHz
Potencia de diseño	TDP 15 W	Máxima 241 W / Base 125 W
Memoria RAM disponible	7,7 GiB	125 GiB
Sistema Operativo	Ubuntu 22.04.3 LTS	Ubuntu 22.04.5 LTS
Kernel	6.8.0-59 - generic	6.8.0-59 - generic

Fuente: elaboración propia

En segundo lugar, el contraste entre las dos máquinas permite poner a prueba la portabilidad real de los *frameworks*, ya que, tanto Python como Java, trabajan con intérpretes y máquinas virtuales que optimizan el código “en caliente”, mientras que C++ produce un binario único que depende de las decisiones del compilador. Evaluar estas diferencias expone si la ventaja tradicional del código nativo permanece intacta cuando la JVM se ajusta dinámicamente o cuando Python delega el trabajo en librerías vectorizadas. Este enfoque sigue la metodología de Merelo-Guervós *et al.* (2016), quienes indican que los benchmarks deben incluir una amplia variedad de tamaños porque el tiempo necesario para realizar operaciones particulares no siempre depende linealmente del tamaño, ya que el rendimiento

<sup>1</sup> Para más información, se puede acceder al siguiente enlace:

<https://www.intel.es/content/www/xl/es/products/sku/95451/intel-core-i77500u-processor-4m-cache-up-to-3-50-ghz/specifications.html>

<sup>2</sup> Para más información, se puede acceder al siguiente enlace:

<https://www.intel.es/content/www/xl/es/products/sku/134600/intel-core-i912900kf-processor-30m-cache-up-to-5-20-ghz/specifications.html>

también se ve afectado por detalles técnicos como la implementación de bucles y gestión de memoria (p. 28).

En tercer lugar, se expone el efecto del subsistema de memoria. El i7-7500U cuenta con 7,7 GiB frente a los 125 GiB del servidor, aunque los algoritmos y configuraciones usadas en el proyecto no agotan esos límites, el acceso a DRAM y los fallos de caché repercuten en la potencia instantánea registrada. Díaz Álvarez *et al.* (2016) indican que la memoria caché puede representar hasta la mitad del consumo energético en sistemas embebidos, citando estudios previos que demuestran el impacto significativo del subsistema de memoria como cuello de botella energético (p. 201).

Por último, para aumentar al máximo la reproducibilidad, sendas máquinas comparten la misma distribución (Ubuntu 22.04 LTS) y un kernel idéntico (6.8.0-59-generic). Esto reduce la varianza entre la propia gestión del algoritmo, cumpliendo la exigencia de Cotta y Martínez-Cruz (2025) de minimizar las diferencias existentes que no estén relacionadas con el lenguaje antes de comparar los resultados (p. 230).

## 4. Resultados experimentales

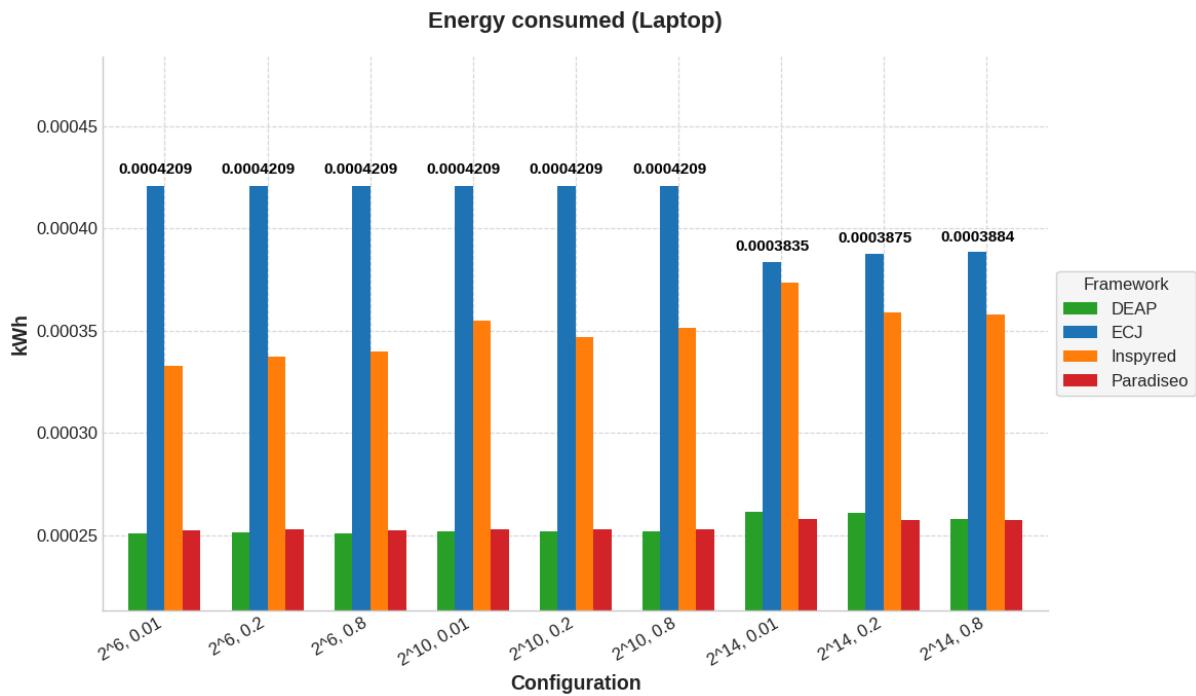
A continuación se estudiarán los resultados experimentales de las ejecuciones realizadas de los diferentes *benchmarks* y *frameworks* utilizados. Para mostrar estos resultados se seguirá el siguiente orden: en primer lugar, las estadísticas recogidas por CodeCarbon en el portátil; en segundo lugar, las estadísticas de calidad de la solución recogidas en el portátil; en tercer lugar, las estadísticas recogidas por CodeCarbon en el servidor; en cuarto lugar, las estadísticas de calidad de la solución recogidas en el servidor; en quinto lugar, una comparativa de portátil y servidor en las estadísticas recogidas por CodeCarbon; en sexto lugar, una comparativa del portátil y el servidor en las estadísticas de calidad de la solución recogidas; y en séptimo lugar, un análisis de la eficiencia energética tanto en el portátil como en el servidor y una comparación entre ambos.

### 4.1. OneMax

En este apartado se analizarán los resultados obtenidos en las métricas mencionadas anteriormente en relación al *benchmark* OneMax.

#### 4.1.1. Consumo energético en el portátil

Figura 1: energía consumida por el portátil en las ejecuciones de OneMax

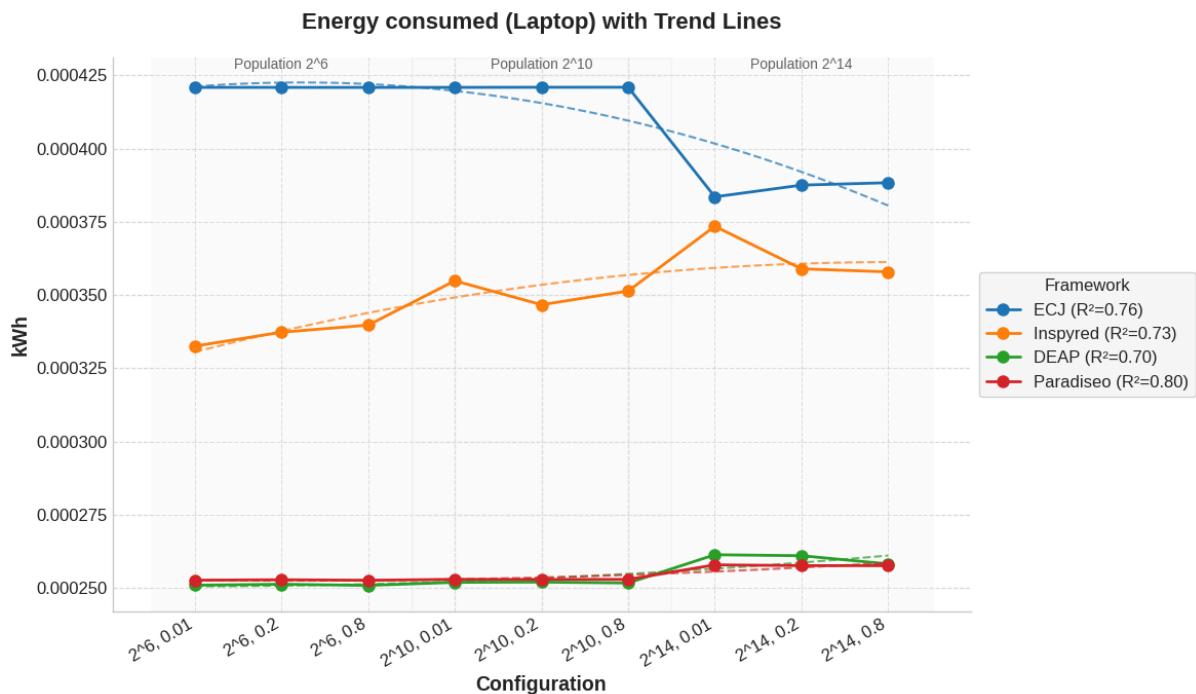


Fuente: elaboración propia

Tal y como se puede observar en la figura 1, se muestra en la misma el orden jerárquico de consumo, el cual se mantiene den las nueve configuraciones evaluadas, las cuales eran combinaciones de tamaño de población  $N \in \{2^6, 2^{10}, 2^{14}\}$  y probabilidad de cruce  $p_c \in \{0.01, 0.2, 0.8\}$ . Así, se puede dilucidar que:

- ECJ ocupa el extremo superior con valores casi constantes de 0,000421 kWh mientras que  $N \leq 2^{10}$ , y sólo cuando la población se eleva a  $2^{14}$  el consumo disminuye a 0,000384 kWh aproximadamente (-8,92 %). Este patrón concuerda con los costes de *warm-up* de la JVM: ECJ fue diseñado desde el inicio para proporcionar no solo alta calidad sino también experimentos replicables, lo cual se deriva de varios factores incluyendo las facilidades consistentes de bajo nivel de ECJ y el hecho de que está dirigido a Java (Scott y Luke, 2019, p. 1392).
- Inspyred aparece, de manera sistemática, en segunda posición. Entre  $N = 2^6$  y  $N = 2^{10}$ , la energía consumida crece de 0,000333 kWh a 0,000355 kWh, lo que supone un aumento del 6,6 %. Con  $N = 2^{14}$  aumenta hasta el 0,000361 kWh aproximadamente. Esto corrobora la observación de que las decisiones de implementación son cruciales, donde incluso en lenguajes interpretados diferentes versiones y bibliotecas pueden mostrar variaciones significativas en el rendimiento (Merelo-Guervós *et al.*, 2016, p. 34).
- DEAP y ParadisEO comparten una cantidad de energía consumida bastante menos: 0,00025 kWh para las primeras 6 configuraciones con un pequeño repunte a 0,000263 kWh en las tres últimas configuraciones. Esto demuestra que ParadisEO ofrece el “piso energético” que se puede esperar de un código desarrollado en C++ optimizado con -O3.

Figura 2: tendencia de la energía consumida por el portátil en las ejecuciones de OneMax



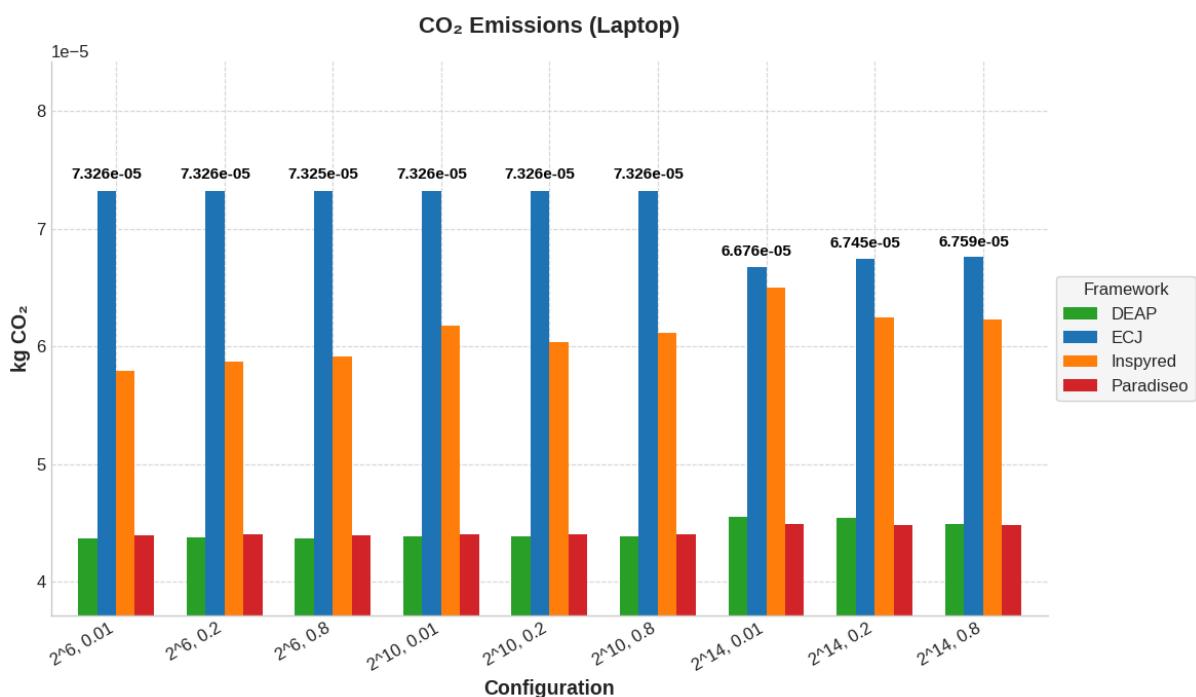
Fuente: elaboración propia

En la figura 2 se pueden observar las líneas de tendencia por cada *framework*, las cuales revelan que:

- ECJ ( $R^2 = 0,76$ ) e Inspyred ( $R^2 = 0,73$ ) mantienen una tendencia hacia reducir el consumo.
- DEAP ( $R^2 = 0,7$ ) mantiene una pendiente casi nula hasta  $N = 2^{10}$ , aunque el pequeño aumento final se puede atribuir al uso de arrays NumPy más grandes, aumentando, lógicamente, los accesos a memoria y, aunque es menester señalar que las CPU Intel de la 11.<sup>a</sup> generación en adelante han eliminado las lecturas de energía DRAM del dominio RAPL en procesadores que no son de grado servidor (Stoico *et al.*, 2025, p. 5).
- ParadisEO ( $R^2 = 0,8$ ) confirma que el consumo se mantiene estable cuando la gestión de memoria es estática y el compilador elimina las llamadas virtuales.

#### 4.1.2. Emisiones totales de CO<sub>2</sub> en el portátil

Figura 3: emisiones totales de CO<sub>2</sub> por el portátil en las ejecuciones de OneMax



Fuente: elaboración propia

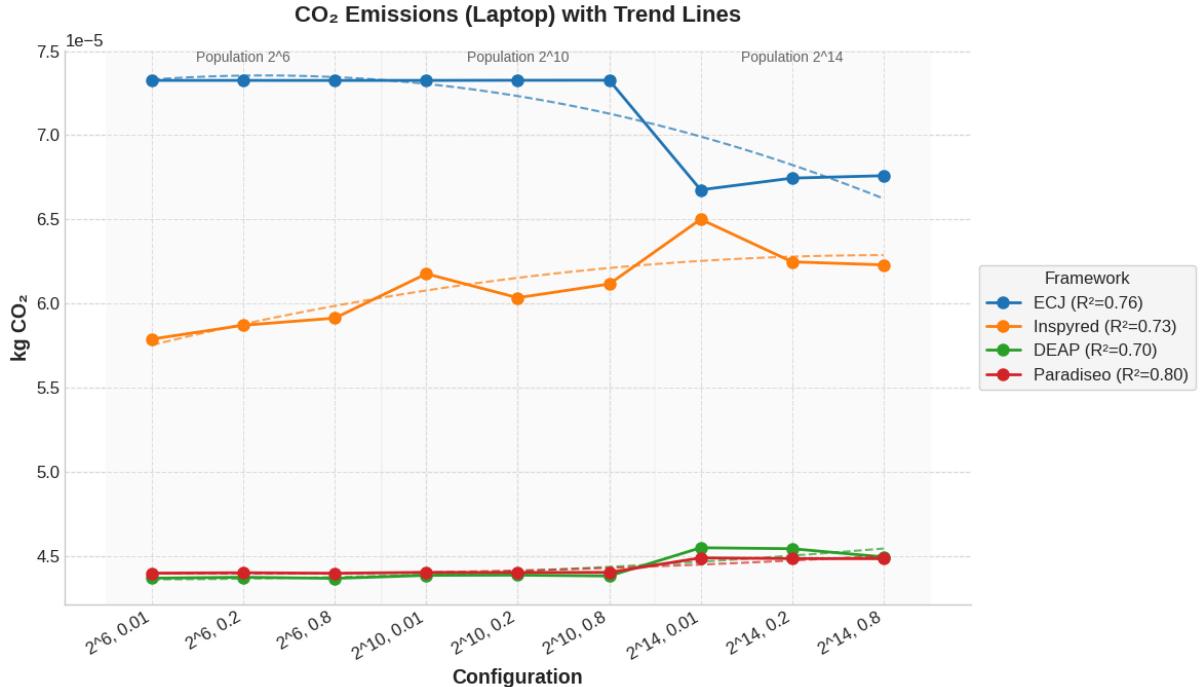
Según se puede observar en la figura 3, las emisiones de carbono coinciden con la energía consumida vista en la sección anterior.

Por un lado, ECJ lidera, de manera considerable, las emisiones en todas las configuraciones. En las seis primeras configuraciones se aprecia unas emisiones cuasimonótonas de 0,0000732 kg CO<sub>2</sub>. El descenso observable al llegar a un tamaño de población de  $N = 2^{14}$  se puede ver tanto en la figura 3 como en la figura 4: el descenso se produce hasta la cantidad de 0,0000668 kg CO<sub>2</sub> ( $\approx -8,8\%$ ).

Por otro lado, Inspyred se mantiene en segunda posición en la cantidad de carbono emitido, con un incremento progresivo desde 0,000058 kg CO<sub>2</sub> hasta 0,0000612 kg CO<sub>2</sub> a

medida que el tamaño de la población aumenta. Puede, igualmente, observarse un leve pico en la configuración  $N = 2^{14}$ ,  $p_c = 0,01$  de 0,000065 kg CO<sub>2</sub>, también visto en el apartado anterior.

Figura 4: tendencia de las emisiones totales de CO<sub>2</sub> por el portátil en las ejecuciones de OneMax



Fuente: elaboración propia

Por último, DEAP y ParadisEO constituyen las opciones menos contaminantes, con unas emisiones controladas que no superan los 0,0000457 kg CO<sub>2</sub> en el caso de DEAP y 0,000045 kg CO<sub>2</sub> en el caso de ParadisEO.

#### 4.1.3. Evolución del *fitness* en el portátil

Para representar el comportamiento evolutivo y la calidad de las soluciones se han propuesto cuatro métricas a evaluar: *fitness* inicial, variación del *fitness*, *fitness* máximo alcanzado y el número de generaciones alcanzadas. Estas métricas pueden observarse en las figuras 5 a 8.

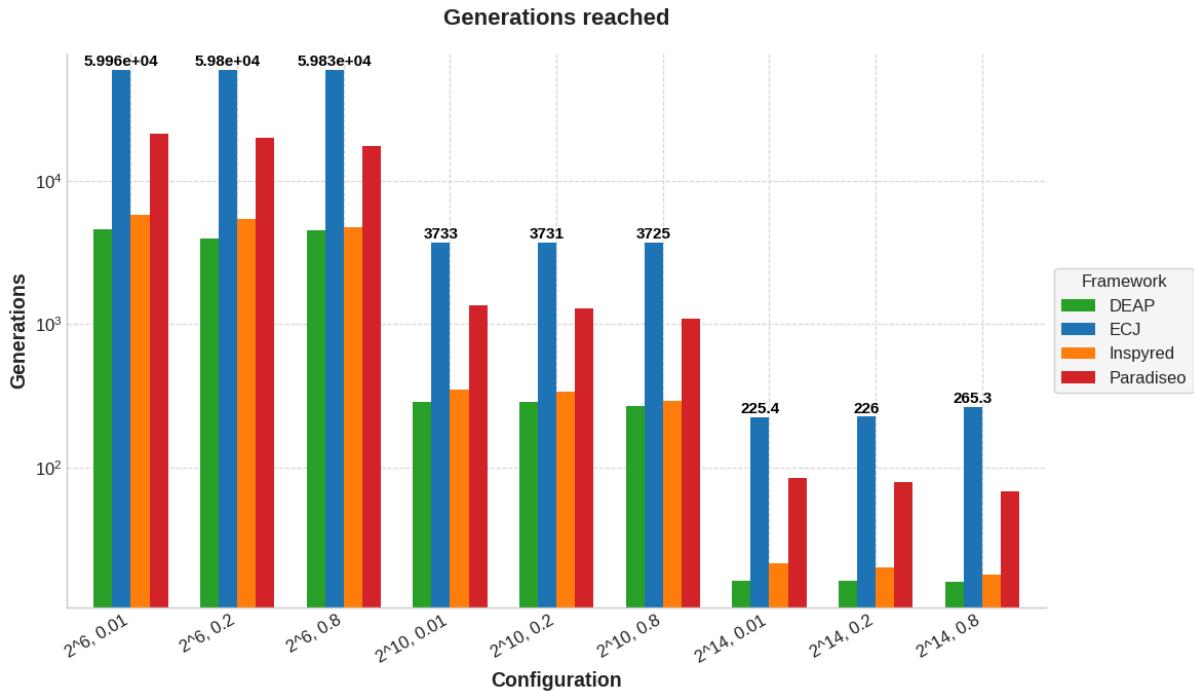
En primer lugar, se exponen en la figura 5 tres órdenes de magnitud de diferencia en el número de generaciones alcanzado:

- Con  $N = 2^6$  los *frameworks* alcanzan casi 60.000 generaciones en el caso de ECJ o casi 5.000 generaciones en el caso de DEAP e Inspyred, estando ParadisEO entre ambos extremos. Estas generaciones se alcanzan antes de cumplir los dos minutos de ejecución, condición de parada expuesta con anterioridad.
- Al crecer la población a  $N = 2^{10}$  el número máximo de generaciones alcanzado es de 3.700 para ECJ y de 300 aproximadamente para DEAP e Inspyred.

- Al llegar la población a  $N = 2^{14}$  el número de generaciones se reduce hasta las 230 aproximadamente en el caso de ECJ y a 16 en el caso de DEAP, siendo ambos, respectivamente, el extremo máximo y mínimo de número de generaciones alcanzadas en todas las configuraciones.

Esta reducción tan drástica confirma una “ley inversa” entre generaciones alcanzadas y tamaño de la población en la cual explican que para una condición de parada similar, los mecanismos de variación operan menos veces cuanto mayor sea el tamaño de la población.

Figura 5: número máximo de generaciones alcanzadas de OneMax



Fuente: elaboración propia

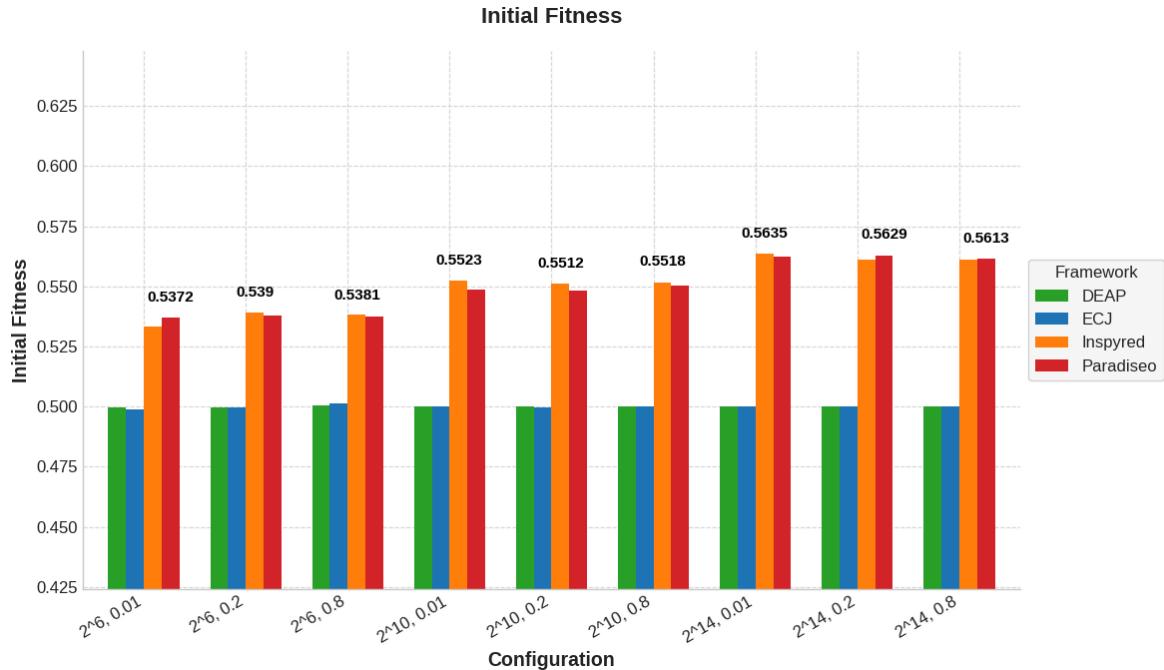
En segundo lugar, en la figura 6 se muestra que Inspyred y ParadisEO parten con un *fitness* inicial promedio de 0,54 (matizando que, de ahora en adelante, las medidas del *fitness* se expresan en tanto por uno, siendo el peor *fitness* posible un 0 y el mejor *fitness* posible un 1), mientras que DEAP y ECJ parten de un *fitness* promedio un poco menor, rondando el 0,5.

En tercer lugar, en la figura 7 se muestra la variación del *fitness*, donde es lógico pensar que, con un mayor porcentaje de probabilidad de cruce ( $p_c$ ), la variación tenderá a ser mayor. Esto se puede observar en las tres configuraciones que tienen esa probabilidad, viendo que dentro del mismo tamaño de población, donde se alcanza mayor variación es cuando se tiene una  $p_c = 0,8$ . Así, se puede destacar que:

- Inspyred sobresale cuando la población es de tamaño medio, dado que su política de mutación explora rápido la diversidad inicial y alcanza la mayor subida absoluta, de casi 0,45 puntos. Sin embargo, cuando la población alcanza un  $N = 2^{14}$  la variación es prácticamente nula con un  $p_c = 0,01$ .

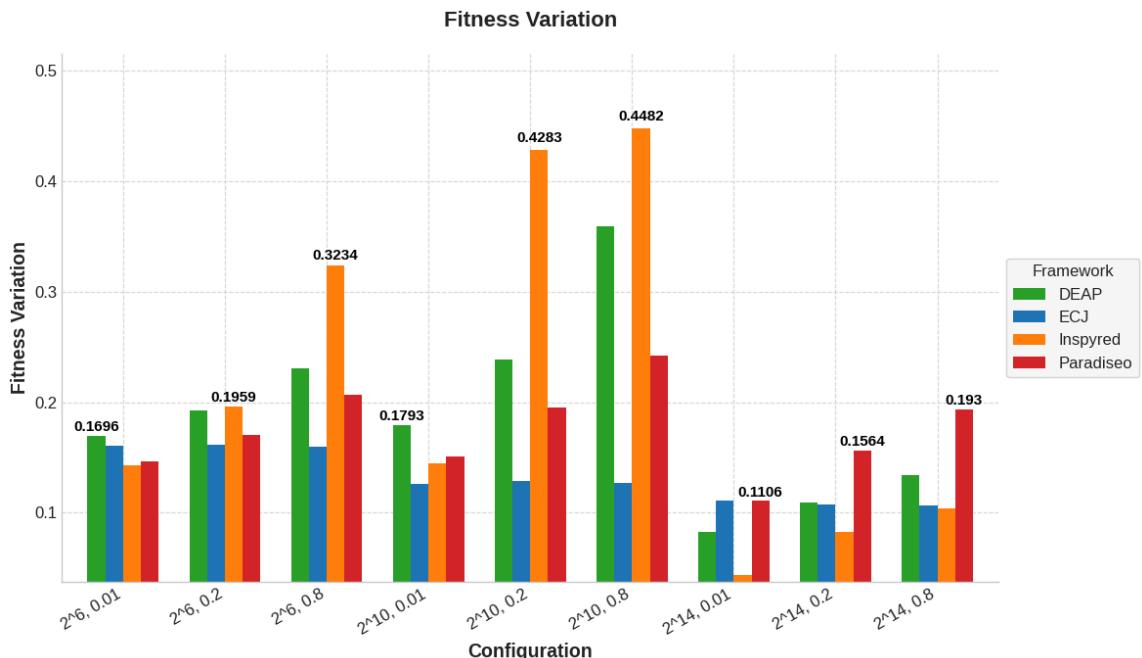
- ECJ mejora aunque se mantiene casi constante independientemente de las configuraciones; el ajuste lineal ( $R^2 = 0,9$ ) muestra que su variación se reduce casi monótonamente con el tamaño de la población.
- ParadisEO mantiene un patrón de ganancias regular gracias a su operador de mutación gaussiano superando al resto de *frameworks* en una población de  $N = 2^{14}$ , donde el número de generaciones alcanzado es mínimo, según se ha visto en la figura 7.

Figura 6: *fitness* inicial promedio de OneMax



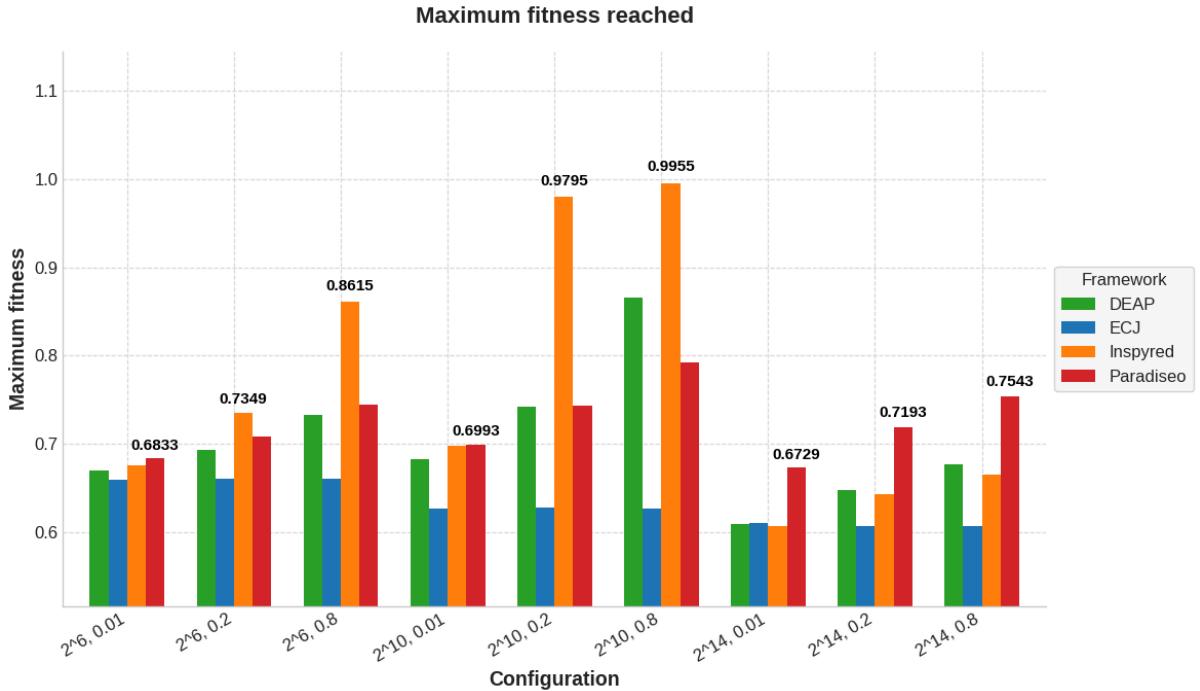
Fuente: elaboración propia

Figura 7: variación promedio del *fitness* de OneMax



Fuente: elaboración propia

Figura 8: *fitness* máximo promedio alcanzado de OneMax



Fuente: elaboración propia

Por último, en la figura 8 se arroja el la calidad de la solución alcanzada, la cual, en general, se alcanza con una población de tamaño  $2^{10}$  y con un índice de probabilidad de cruce de 0.8.

En este punto, Inspyred es el que alcanza un *fitness* máximo mayor, muy próximo a 1. Es menester mencionar que este *framework* supera al resto en las primeras seis configuraciones, sólo quedando atrás en las configuraciones con un tamaño de población de  $N = 2^{14}$ , donde es ParadisEO el que alcanza una calidad mejor. Esto es lógico al comparar las figuras 7 y 8 y lo mencionado anteriormente, gracias a lo cual, al ser ParadisEO el que tiene una variación mayor en esta franja, será el que mayor *fitness* consiga.

Además, cabe mencionar que ECJ se estanca en un 0,64 aproximadamente, debido a que su motor de cruce binario sin elitismo resulta conservador.

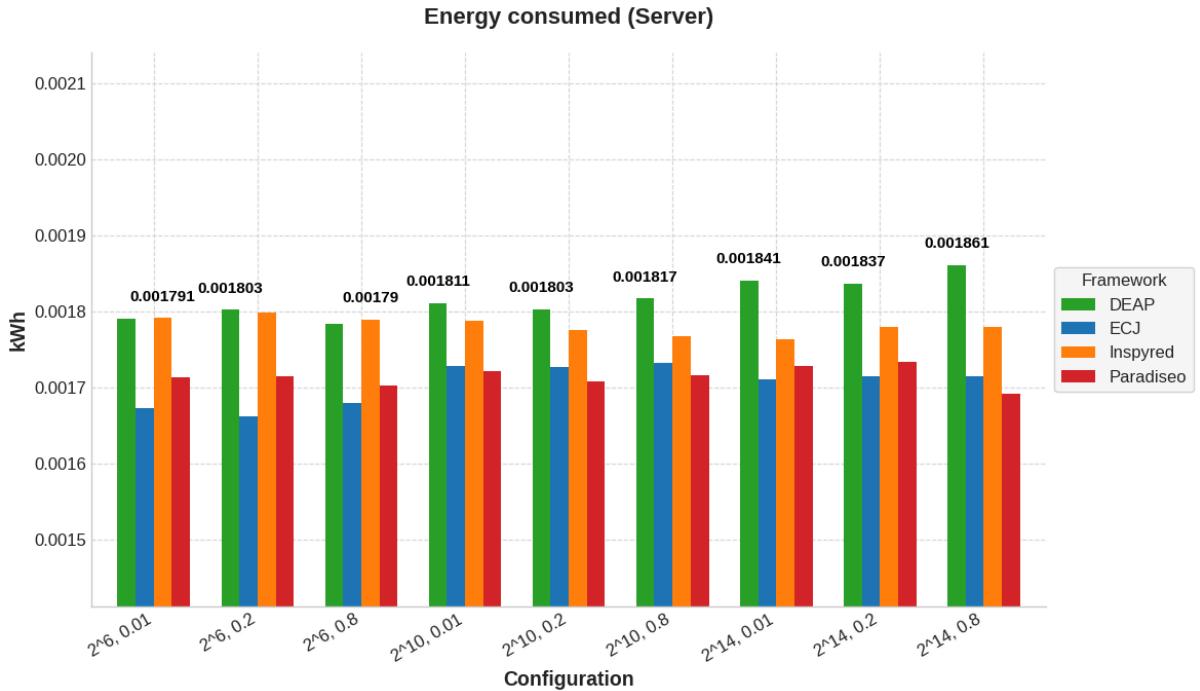
#### 4.1.4. Consumo energético y emisiones en servidor

Como se puede apreciar en la figura 9, el orden jerárquico de consumo en el servidor se mantiene estable en las nueve configuraciones ejecutadas ( $N \in \{2^6, 2^{10}, 2^{14}\}$  y probabilidad de cruce  $p_c \in \{0.01, 0.2, 0.8\}$ ), pero de manera contraria al portátil:

- DEAP ocupa ahora el extremo superior con valores comprendidos entre 0,001791 kWh y 0,001861 kWh. En la figura 10 se puede ver cómo la tendencia es claramente de crecimiento casi lineal conforme aumenta la población.
- Inspyred se sitúa inmediatamente por debajo: comienza 0,0018 kWh y oscila levemente descendiendo hasta un mínimo relativo en  $N = 2^{10}$  y  $p_c = 0,8$  con valor aproximado de 0,001625 kWh.

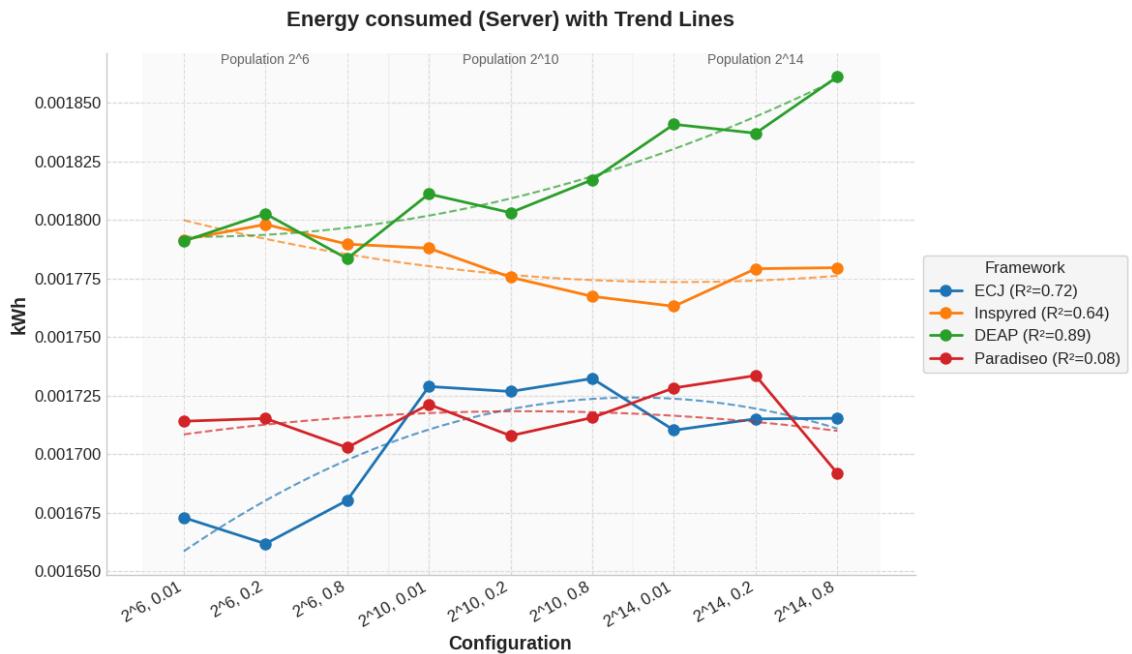
- ParadisEO es el *framework* cuyo consumos son más constantes en todas las configuraciones, encontrándose entre los 0,001707 kWh y los 0,001729 kWh, consolidándose como el tercero que más consume.
- ECJ pasa ahora a ser el menos costoso, teniendo consumos entre los 0,001657 kWh y los 0,001729 kWh.

Figura 9: energía consumida por el servidor en las ejecuciones de OneMax



Fuente: elaboración propia

Figura 10: tendencia de la energía consumida por el servidor en las ejecuciones de OneMax



Fuente: elaboración propia

Si se analiza la diferencia absoluta entre el mejor (ECJ) y el peor (DEAP) es de apenas 0,00011 kWh, lo que supone aproximadamente un 6 %, estando muy lejos de la brecha del 40 % vista en las mediciones del portátil. Los resultados muestran cómo las diferencias en la arquitectura del hardware influyen en el consumo energético, donde dispositivos con múltiples núcleos presentan comportamientos energéticos distintos según su capacidad de procesamiento (Fernández de Vega *et al.*, 2016, p. 551).

Además, centrando la atención en la figura 10, se puede observar que:

- DEAP ( $R^2 = 0,89$ ) tiene una pendiente positiva pronunciada, lo que se puede traducir en que la copia de *arrays* NumPy de mayor tamaño penaliza cuando el servidor ejecuta cada generación, por lo que el coste de memoria vuelve a hacerse visible.
- Inspyred ( $R^2 = 0,64$ ) dibuja una pendiente ligeramente negativa hasta que comience la población mayor; su motor de mutación reduce la energía por evaluación aunque este efecto se pierda en el último trío de configuraciones.
- ECJ ( $R^2 = 0,72$ ) mantiene una línea casi horizontal una vez que la JVM se estabiliza, ejecutando código compilado a una velocidad cercana al C++ nativo, y el consumo queda dominado por la propia función evaluadora.
- ParadisEO ( $R^2 = 0,09$ ) se estabiliza en un consumo casi plano desde el principio.

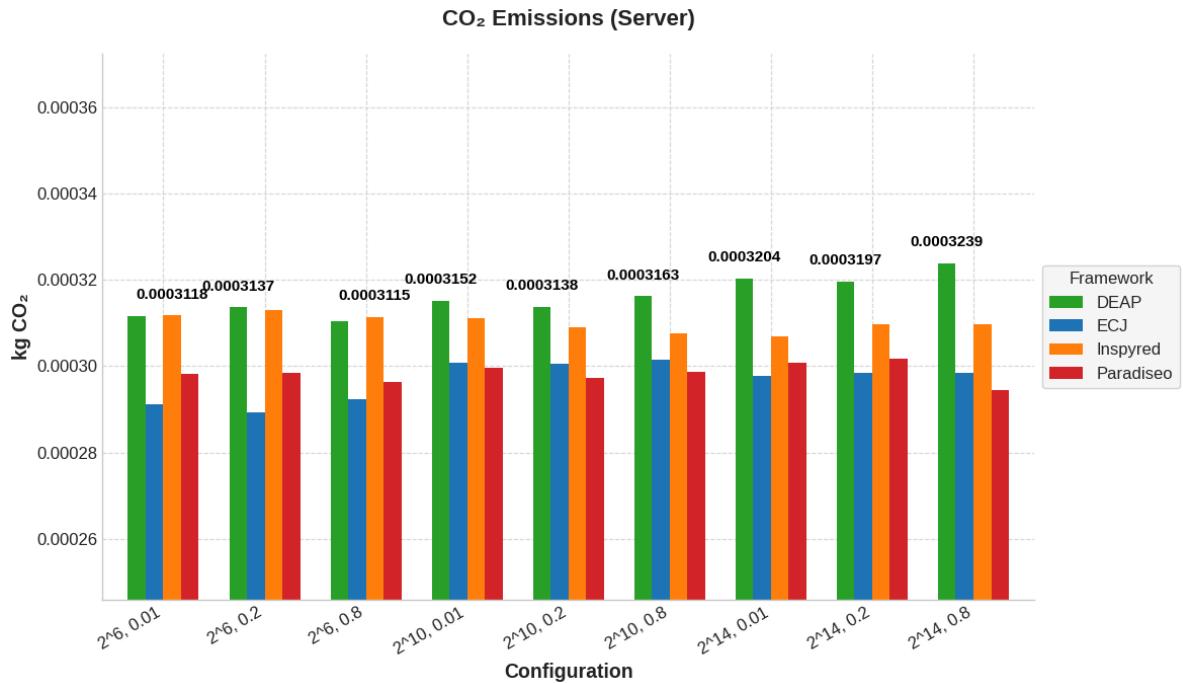
#### 4.1.5. Emisiones totales de CO<sub>2</sub> en servidor

En la figura 11 se expone el orden jerárquico de emisiones de CO<sub>2</sub> en el servidor, pudiendo afirmar que se invierte por completo respecto al portátil. En las nueve configuraciones ejecutadas ( $N \in \{2^6, 2^{10}, 2^{14}\}$  y probabilidad de cruce  $p_c \in \{0.01, 0.2, 0.8\}$ ), se observa que:

- DEAP ocupa el primer puesto en siete de las nueve configuraciones, con valores que crecen de manera constante pero suavemente desde los 0,0003118 kg CO<sub>2</sub> hasta los 0,0003239 kg CO<sub>2</sub>. La pendiente positiva que se observa en la figura 12, con un  $R^2 = 0,89$ , confirma que el consumo aumenta cuando la población obliga a copiar arrays NumPy más grandes; Stoico *et al.* (2025) documentan esta limitación que afecta las mediciones energéticas (p. 5).
- Inspyred queda en segundo lugar, descendiendo las emisiones ligeramente hasta la configuración  $N = 2^{14}$  y  $p_c = 0,01$ , repuntando en las dos siguientes configuraciones. La tendencia inicial negativa, con un  $R^2 = 0,64$ , avala la hipótesis de amortización de *overhead* cuando el número de generaciones baja, mientras que el aumento final indica que la mutación pierde eficiencia en poblaciones muy grandes.
- ParadisEO expone una banda estrecha de valores de emisiones, situándose entre los 0,000296 kg CO<sub>2</sub> y los 0,0003239 kg CO<sub>2</sub>, con una línea de tendencia prácticamente horizontal.

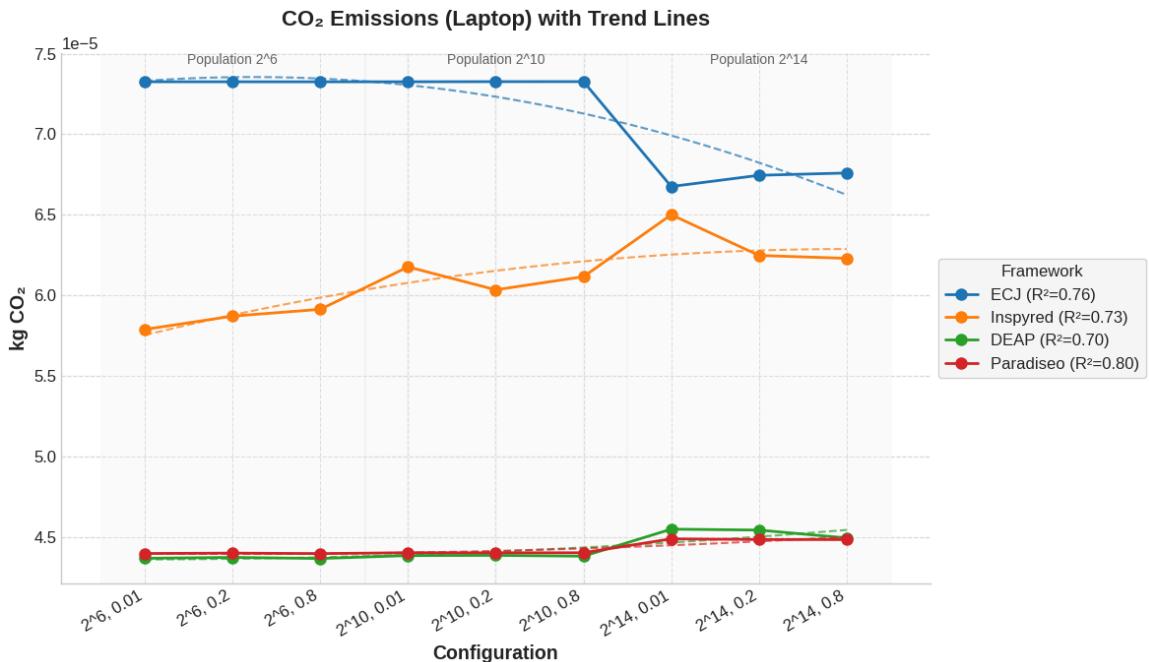
- ECJ para ser, de media, el que menos emisiones genera, ya que oscila entre los 0,0002905 kg CO<sub>2</sub> y los 0,000301 kg CO<sub>2</sub>, con una pendiente suave y un  $R^2 = 0,72$ . El *warm-up* JVM apenas tiene un peso considerable frente a la potencia sostenida del chip (241 W), de modo que parte de la penalización en el portátil se diluye, tal como indican Fernández de Vega *et al.* (2016) al estudiar la elasticidad energética en arquitecturas (p. 553).

Figura 11: emisiones totales de CO<sub>2</sub> por el servidor en las ejecuciones de OneMax



Fuente: elaboración propia

Figura 12: tendencia de las emisiones totales de CO<sub>2</sub> por el servidor en las ejecuciones de OneMax



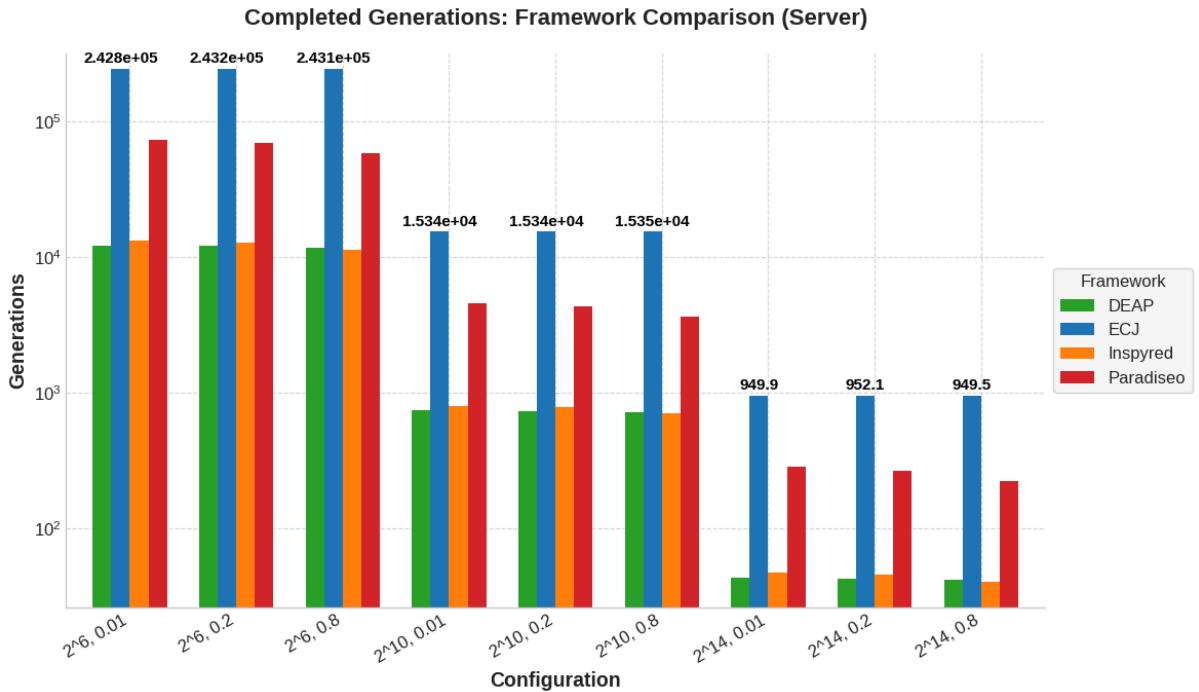
Fuente: elaboración propia

Por último, la brecha absoluta entre el peor (DEAP) y el mejor (ECJ) es de 0,00033 kg CO<sub>2</sub>, lo que supone un 10 % del valor máximo, estando muy lejos del casi 60 % registrado en el portátil. Sin embargo, en magnitud de cantidad de emisiones, se encuentra bastante por encima de los valores que se registraron en el portátil.

#### 4.1.6. Evolución del *fitness* en servidor

En la figura 13 se observa cómo en el servidor, gracias a su alta capacidad de cómputo, permite alcanzar aproximadamente 243.000 generaciones en ECJ cuando  $N = 2^6$ , Inspyred y DEAP se encuentran en torno a las 12.000 generaciones y ParadisEO las 60.000.

Figura 13: número máximo de generaciones alcanzadas de OneMax



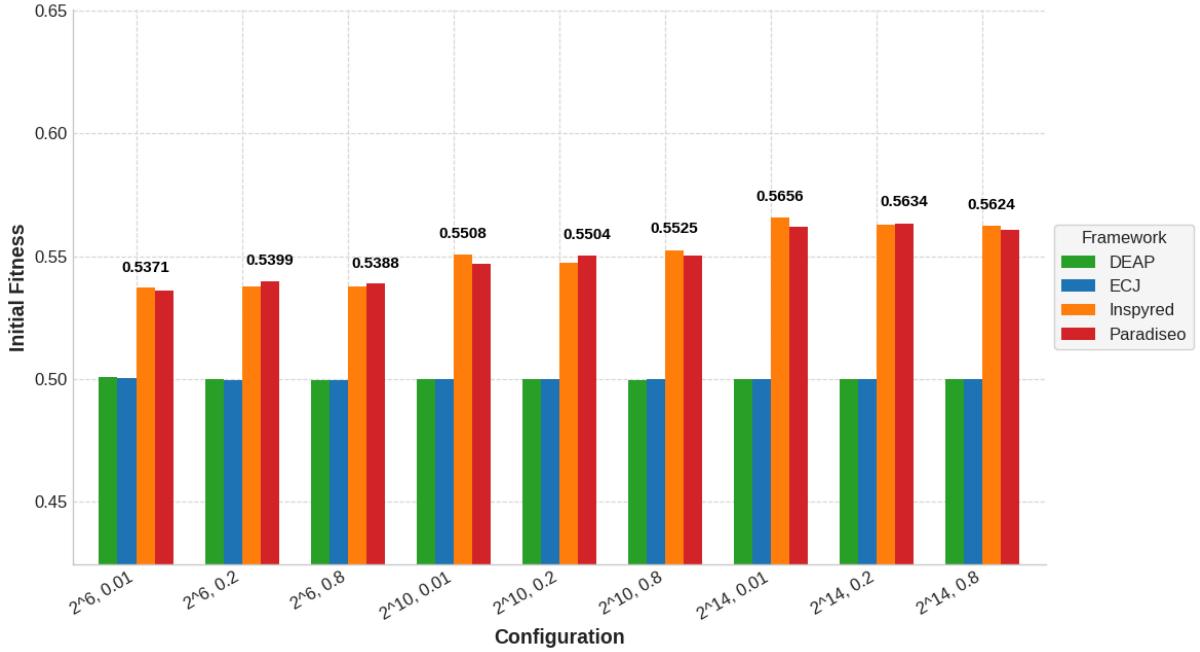
Fuente: elaboración propia

Al aumentar la población a  $2^{10}$ , ECJ cae hasta las 15.300 generaciones aproximadamente, mientras que Inspyred y DEAP alcanzan las 850, reflejo de la misma ley inversa entre población y generaciones. Finalmente, con un tamaño de población de  $2^{14}$ , el número de generaciones de ECJ vuelve a caer drásticamente llegando a ejecutar 950 generaciones, y tanto Inspyred como DEAP evalúan tan sólo 50 generaciones, confirmando que las diferencias en consumo energético pueden deberse a la utilización de la jerarquía de memoria, siendo menos eficientes los sistemas más potentes (Fernández de Vega *et al.*, 2016, p. 374).

A la hora de iniciar la evaluación, el escenario presentado en la figura 14 es similar al ya visto en el portátil. En este caso se observa que Inspyred y ParadisEO comienzan con un *fitness* inicial promedio de 0,55 mientras que DEAP y ECJ parten de un *fitness* inicial promedio un poco menor, rondando el 0,5.

Figura 14: *fitness* inicial promedio de OneMax

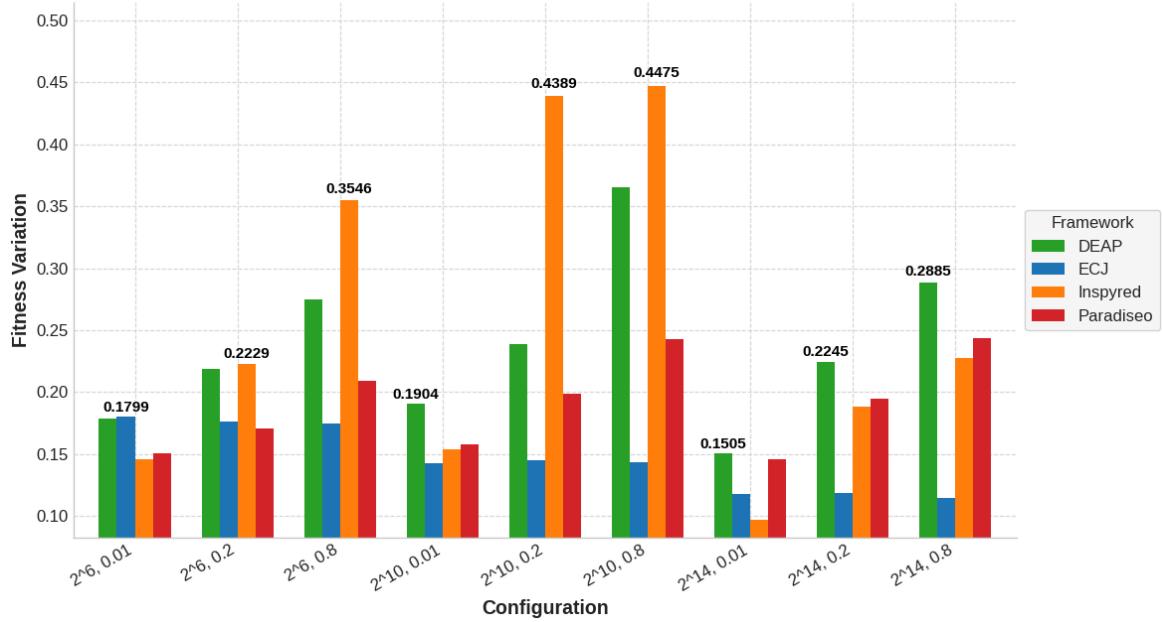
Initial Fitness: Framework Comparison (Server)



Fuente: elaboración propia

Figura 15: variación promedio del *fitness* de OneMax

Fitness Variation: Framework Comparison (Server)

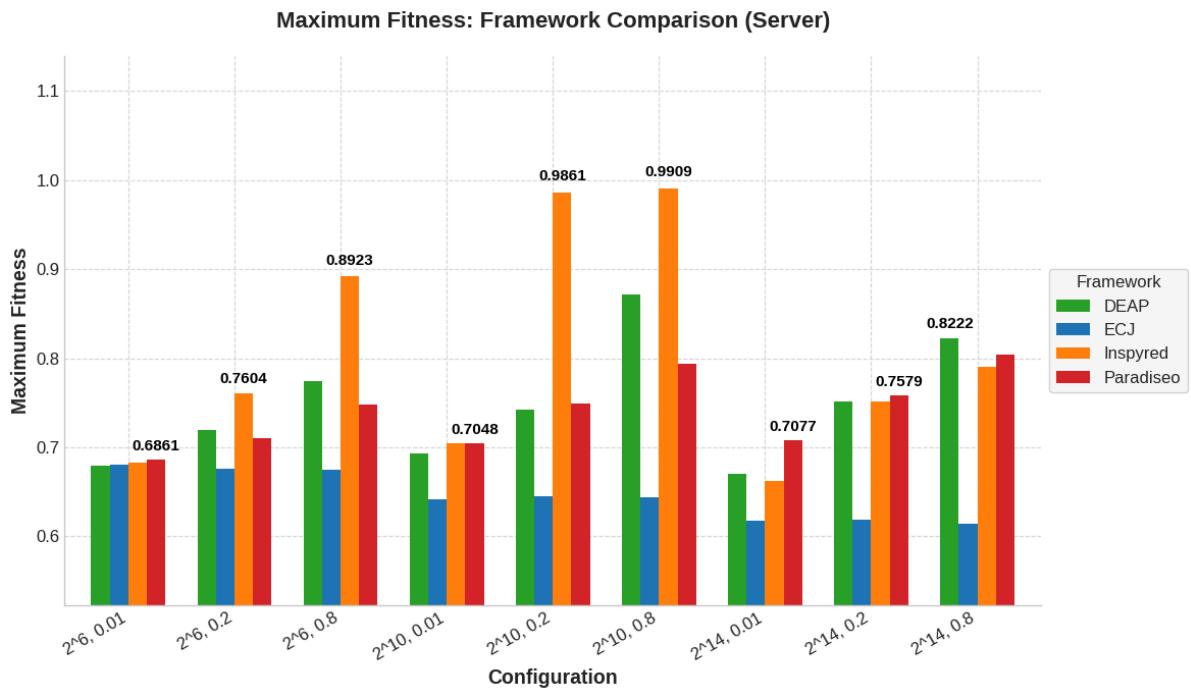


Fuente: elaboración propia

En la figura 15 se muestra que, en momentos puntuales, más concretamente en cuatro de las nueve configuraciones, Inspyred alcanza la mayor ganancia media, siendo esta de aproximadamente 0,45 cuando  $N = 2^{10}$  y  $p_c = 0,8$ . DEAP obtiene mejoras mayores en otras cuatro configuraciones, señalando que DEAP es el que alcanza una mayor variación cuando la población es considerablemente grande. Por el contrario, tanto ECJ se puede ver que

mantiene un ratio de mejora muy limitado, con una media de 0,15. Y de igual forma ocurre con ParadisEO, el cual mantiene un patrón de mejora dentro de cada tamaño de población, haciendo especial mención en que la mayor diferencia de variación entre las tres posibles probabilidades de cruce se da en la población mayor. Así, todo esto confirma que, en entornos de alto coste por generación, los *frameworks* más ligeros en llamadas a biblioteca escalan mejor la mejora por evaluación.

Figura 16: *fitness* máximo promedio alcanzado de OneMax



Fuente: elaboración propia

Por último, según se la figura 16, el mejor *fitness* alcanzado aparece para Inspyred en  $N = 2^{10}$  y  $p_c = 0,8$ : 0,991. DEAP logra un 0,873, ParadisEO un 0,795 y ECJ tan solo un 0,645.

Un punto importante a destacar es la brecha entre Inspyred y ECJ, dado que supera el 35 %, mucho mayor que la diferencia de consumo, lo que subraya que la calidad de la solución no siempre acompaña la frugalidad energética y que la capa de biblioteca (más que la del intérprete) define la capacidad de exploración y explotación de la función objetivo.

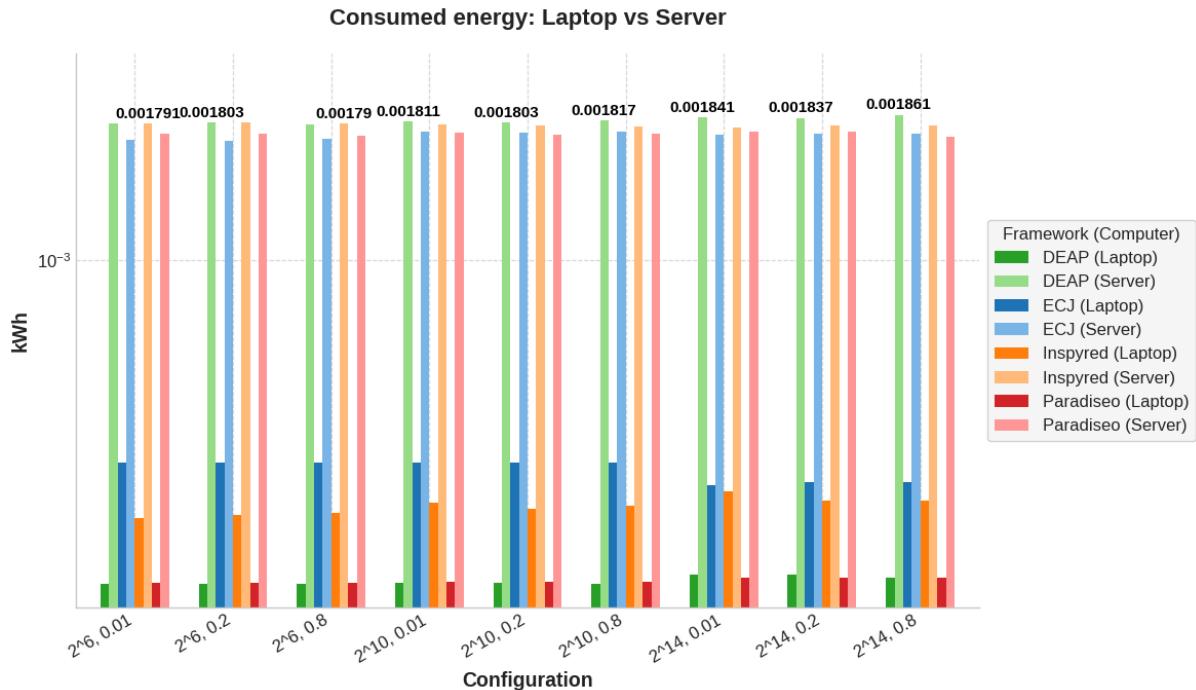
#### 4.1.7. Consumo energético en el portátil frente al servidor

Fernández de Vega *et al.* (2016) mencionan la necesidad de una computación *energy-proportional* y *green computing* para abordar los problemas de consumo energético en entornos computacionales (p. 549). Este enfoque, originalmente formulado en el contexto de centros de datos, se hace igualmente relevante al comparar arquitecturas muy dispares, como un portátil y un servidor, en la ejecución de algoritmos evolutivos.

Los datos muestran que, para configuraciones idénticas de tamaño de población y probabilidad de cruce, en el portátil el consumo total oscila entre 0,000252 kWh y 0,000388

kWh, mientras que en el servidor el consumo lo hace entre 0,001666 kWh y 0,001861 kWh. En todos los casos, el servidor requiere entre 4,3 y 7,4 veces más energía que el portátil para resolver el problema OneMax bajo las mismas condiciones de ejecución.

Figura 17: comparación de energía consumida de OneMax entre el portátil y el servidor

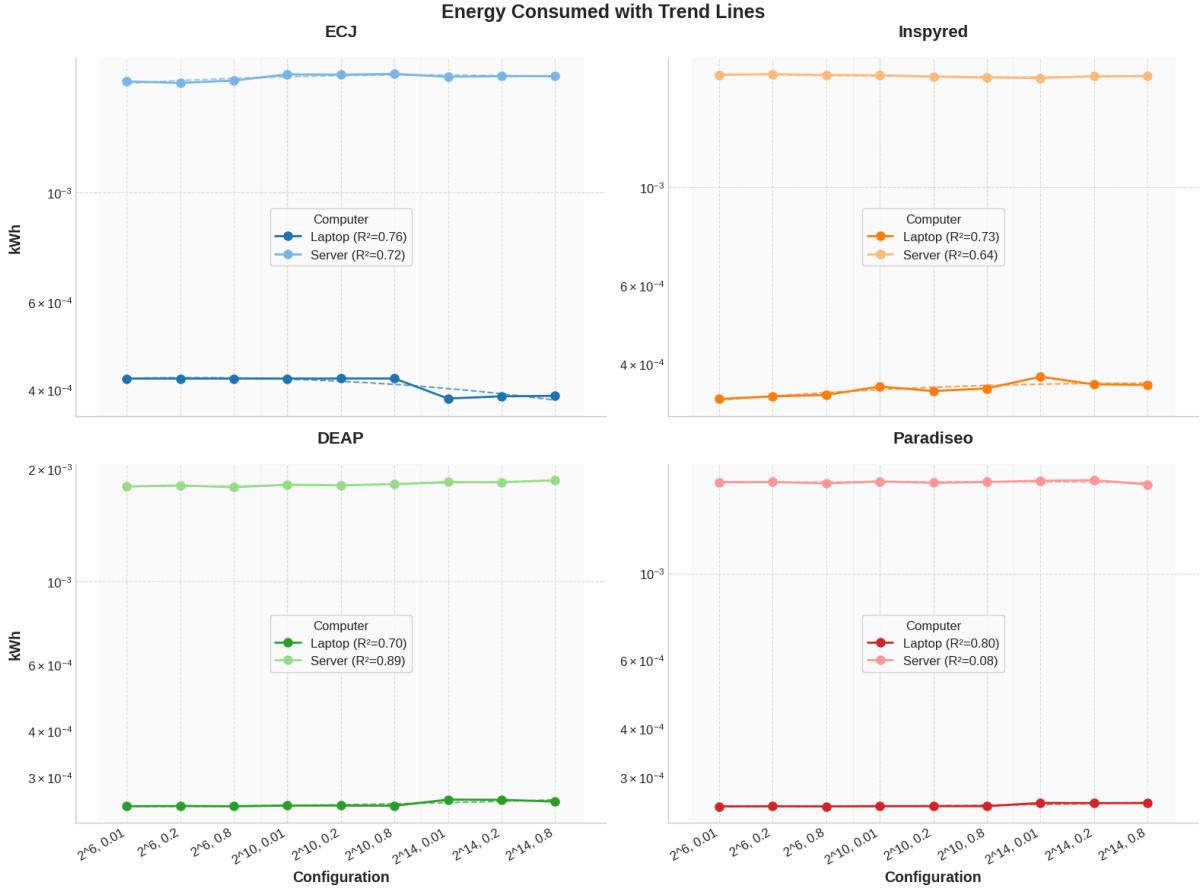


Fuente: elaboración propia

Al analizar la evolución del consumo en función del tamaño de la población (de  $2^6$  a  $2^{14}$  individuos) se observa que en el portátil, frameworks como ECJ e Inspyred presentan una disminución notable del consumo medio por generación (coeficientes de determinación  $R^2$  próximos a 0,75). Otros, como DEAP y Paradiseo, mantienen un consumo prácticamente constante ( $R^2$  entre 0,7 y 0,8).

En el servidor, DEAP muestra un incremento marcado del consumo con el aumento de población ( $R^2$  en torno a 0,89), probablemente debido a la sobrecarga de gestión de hilos y sincronización en entornos masivamente paralelos; ECJ y Paradiseo varían de forma moderada ( $R^2$  aproximado de 0,72 y 0,08), mientras que Inspyred tiende a una ligera reducción ( $R^2 \approx 0,64$ ).

Figura 18: comparación desglosada de energía consumida de OneMax entre el portátil y el servidor



Fuente: elaboración propia

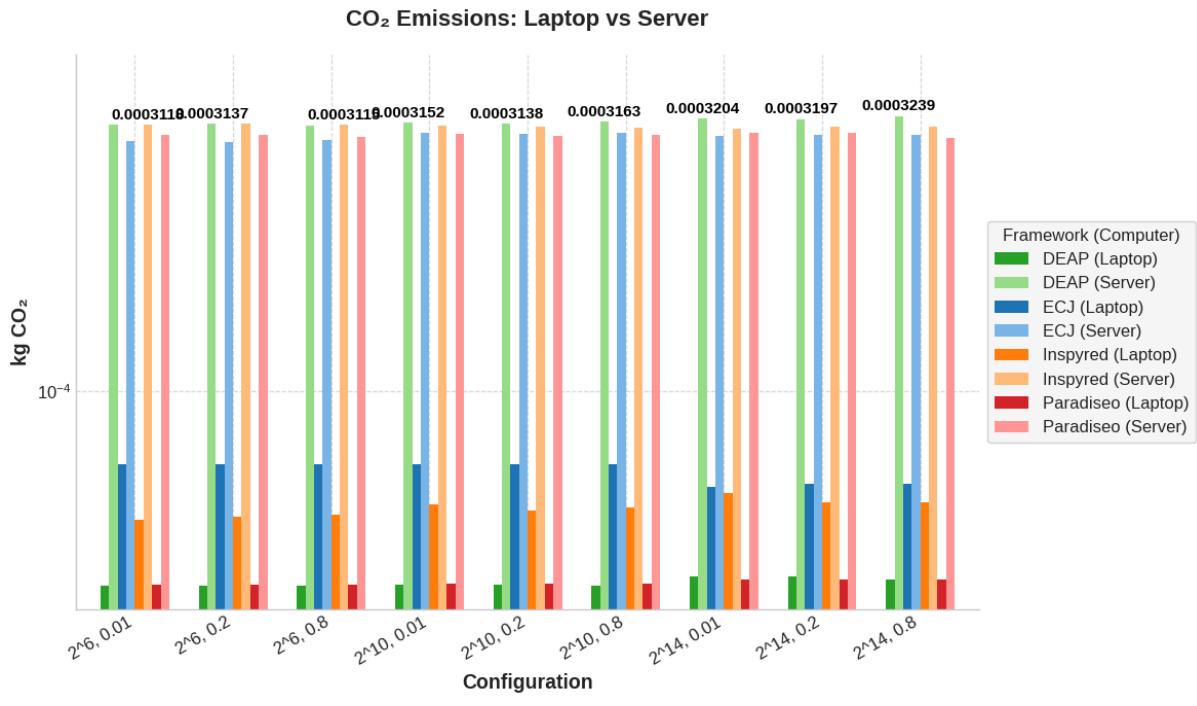
#### 4.1.8. Emisiones totales de CO<sub>2</sub> en el portátil frente al servidor

Para comparar de manera directa las emisiones entre el portátil y el servidor, se toman como referencia las figuras 19 y 20, las cuales son las concernientes a las emisiones generadas.

Por un lado, el portátil presenta unas emisiones medias totales que abarcan desde 0,0000438 kg CO<sub>2</sub> (DEAP, población  $2^6$ ,  $p_c=0,01$ ) hasta 0,0000733 kg CO<sub>2</sub> (ECJ, población  $2^6$ ,  $p_c=0,01$ ). Para las configuraciones con población  $2^{14}$ , las emisiones oscilan entre 0,0000456 kg CO<sub>2</sub> (DEAP) y 0,0000676 kg CO<sub>2</sub> (ECJ). Además, frameworks como ECJ e Inspyred presentan una reducción de emisiones por ejecución al incrementar la población de  $2^6$  a  $2^{14}$  (coeficientes de determinación  $R^2 \approx 0,75$ ), sugiriendo que el paralelismo interno amortigua el impacto por iteración. DEAP y Paradiseo, en cambio, mantienen emisiones casi constantes.

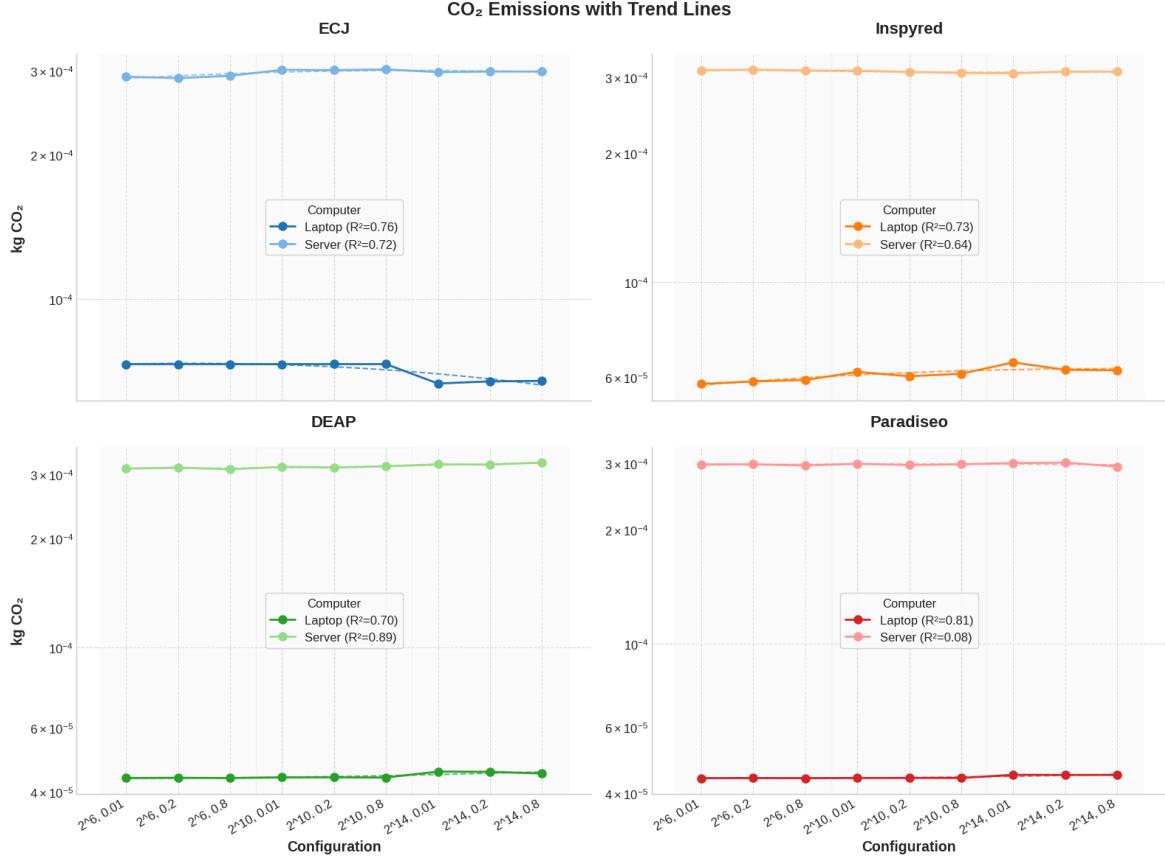
Por otro lado, en el servidor los valores van de 0,0003118 kg CO<sub>2</sub> (DEAP, población  $2^6$ ,  $p_c=0,01$ ) a 0,0003239 kg CO<sub>2</sub> (DEAP, población  $2^{14}$ ,  $p_c=0,8$ ). En suma, DEAP muestra un aumento marcado de emisiones con la población ( $R^2 \approx 0,89$ ), debiéndose posiblemente por la sobrecarga de gestión de hilos y sincronización en arquitecturas masivamente paralelas. ECJ y Paradiseo oscilan de modo moderado ( $R^2 \approx 0,72$  y  $0,08$ ), mientras que Inspyred tiende a una ligera disminución ( $R^2 \approx 0,64$ ).

Figura 19: comparación de emisiones de CO<sub>2</sub> de OneMax entre el portátil y el servidor



Fuente: elaboración propia

Figura 20: comparación desglosada de emisiones de CO<sub>2</sub> de OneMax entre el portátil y el servidor



Fuente: elaboración propia

Así, el servidor emite entre 4,3 y 7,4 veces más CO<sub>2</sub> que el portátil para resolver el mismo problema con idénticas configuraciones.

En conclusión, el análisis de las gráficas de emisiones totales de CO<sub>2</sub> demuestra que el portátil ofrece una ventaja clara en eficiencia y emisiones frente al servidor.

#### 4.1.9. Análisis la evolución del *fitness* en el portátil frente al servidor

Ambas máquinas comparten el mismo proceso de inicialización aleatoria, por lo que los valores promedio de *fitness* al arranque se sitúan cerca de 0,50 para DEAP y ECJ en todas las configuraciones. Sin embargo, en Inspyred y Paradiseo, el portátil arranca ligeramente por encima que en el servidor especialmente en poblaciones grandes ( $2^{14}$ ), donde la dispersión del muestreo aleatorio aumenta la varianza del promedio. Esta leve diferencia puede atribuirse simplemente al muestreo estadístico en cada ejecución, sin impacto real en la comparación posterior.

La variación de *fitness*, calculada como la diferencia entre *fitness* máximo e inicial, muestra patrones parecidos en ambas plataformas:

- Inspyred es el *framework* que más gana en *fitness* cuando la población es intermedia ( $2^{10}$  con  $p_c=0,8$ ), alcanzando variaciones entre 0,43 y 0,45 en el portátil y entre 0,44 y 0,45 en el servidor.
- DEAP se queda en torno a 0,17 y 0,36 en el portátil para poblaciones pequeñas e intermedias, mientras que en el servidor sube hasta casi 0,29 en la máxima población.
- Paradiseo mantiene variaciones moderadas (0,15 a 0,24 en el portátil y 0,15 a 0,25 en el servidor), con un pico en configuraciones de población media.
- ECJ, por su parte, muestra la menor ganancia de *fitness* en ambas plataformas (aproximadamente de 0,12 a 0,18), reflejo de una estrategia de evolución más conservadora.

En términos generales, la curva de ganancia se superpone casi idéntica entre portátil y servidor, indicando que el paralelismo del *servidor* no altera la dinámica de búsqueda, sino únicamente la velocidad de ejecución.

El *fitness* perfecto (*fitness* = 1.0) no se llega a lograr, y los máximos reales dependen fuertemente de la población y del *framework*:

- Inspyred es el más exitoso, llegando a 0,9955 (portátil) y 0,9909 (servidor) en la configuración  $N = 2^{10}$  y  $p_c = 0,8$ .
- DEAP alcanza valores entre 0,67 ( $2^{14}$ ) y 0,87 ( $2^{10}$ ) en el portátil, mientras que en el servidor sube hasta 0,82 en la población más grande.
- Paradiseo presenta resultados intermedios, con un máximo de 0,79 (portátil,  $N = 2^{10}$ ) y 0,80 (servidor,  $N = 2^{14}$ ).

- ECJ se mantiene estable en torno a 0,61-0,65 en todas las configuraciones.

Aunque las diferencias absolutas son pequeñas entre plataformas, el portátil tiende a obtener ligeramente mejores óptimos en configuraciones medias ( $2^{10}$ ), quizás por una convergencia más estable ante recursos limitados.

Por último, el recuento de generaciones completadas antes de detenerse revela una clara ventaja de cálculo bruto del *servidor*, visible en la tabla 3:

- Para poblaciones pequeñas ( $2^6$ ), el servidor completa, aproximadamente, 240.000 generaciones, frente a las 60.000 en el portátil, lo que supone casi cuatro veces más.
- Con poblaciones intermedias ( $2^{10}$ ), esa relación se mantiene (15.000 vs. 3.700), igual que la magnitud de cuatro veces más.
- En poblaciones grandes ( $2^{14}$ ), ambas máquinas requieren muchas menos generaciones (unas 200 y 300 en el portátil, y casi 950 en el servidor), manteniéndose de nuevo una proporción de cuatro veces más.

Este patrón uniforme indica que el servidor acelera el ritmo de iteración en un factor cercano a cuatro, sin alterar el número de generaciones necesario para que el algoritmo alcance al criterio de parada.

Tabla 3: Resumen aproximado de las generaciones alcanzadas de OneMax

<i>N</i> y <i>p<sub>c</sub></i>	<i>portátil</i>	<i>servidor</i>	Razón <i>servidor/portátil</i>
$2^6, 0,01$	59.960	242.800	4,05
$2^6, 0,2$	59.800	243.200	4,07
$2^6, 0,8$	59.830	243.100	4,06
$2^{10}, 0,01$	3.733	15.340	4,11
$2^{10}, 0,2$	3.731	15.340	4,11
$2^{10}, 0,8$	3.725	15.350	4,12
$2^{14}, 0,01$	225	950	4,22
$2^{14}, 0,2$	226	952	4,21
$2^{14}, 0,8$	265	949	3,58

Fuente: elaboración propia

#### 4.1.10. Análisis de la eficiencia energética en el portátil

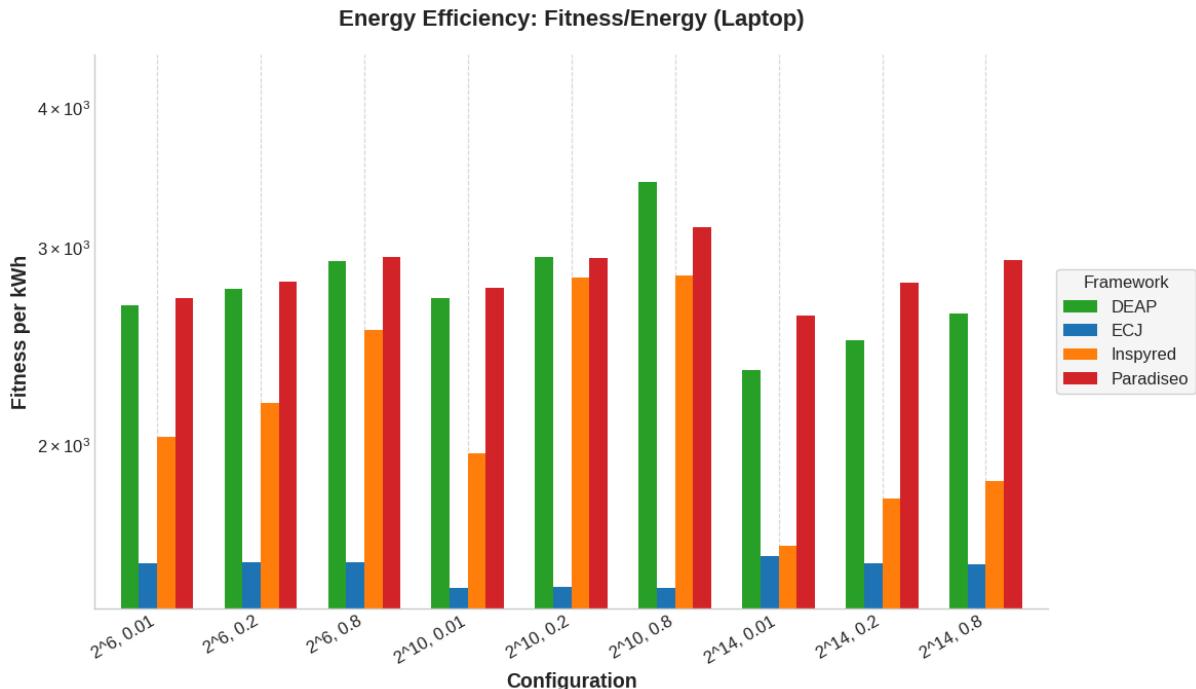
Para evaluar la eficiencia energética de cada *framework* en el portátil, se define la métrica

$$\eta = \frac{\text{Fitness máximo alcanzado}}{\text{Consumo (kWh)}}$$

que indica cuántas unidades de *fitness* se obtienen por cada kWh consumido. Así, gracias a la figura 21 se puede observar que:

- Paradiseo alcanza sistemáticamente la mayor eficiencia energética en todas las configuraciones, con valores de  $\eta$  en torno a 2.650 y 3.150. Esto indica una implementación particularmente ligera que aprovecha bien cada vatio consumido.
- DEAP ocupa el segundo puesto, con  $\eta$  oscilando entre 2.600 y 3.500 en las configuraciones intermedias. Destaca especialmente en la población con  $N = 2^{10}$  y  $p_c = 0,8$ , donde se registra el pico máximo de 3.480 *fitness/kWh*.
- Inspyred muestra una eficiencia más variable, con  $\eta$  comprendido entre 1.950 y 2.800, apuntando a que su estrategia de exploración trabaja bien ciertos tamaños de población pero resulta menos rentable energéticamente en poblaciones extremas.
- ECJ presenta los valores más bajos de  $\eta$ , estando entre 1.200 y 1.350, reflejando una trayectoria evolutiva robusta pero con bajo rendimiento por unidad de energía.

Figura 21: comparación entre *frameworks* del cálculo de  $\eta$  de OneMax en el portátil



Fuente: elaboración propia

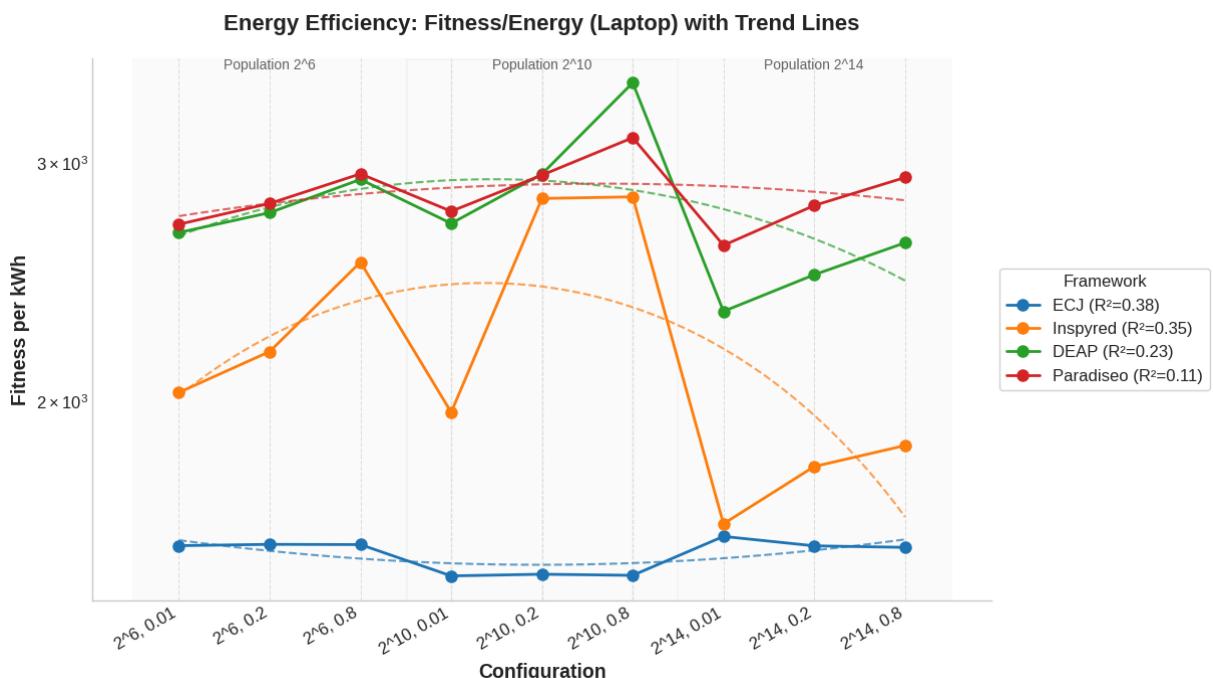
Las líneas de tendencia polinomiales revelan un comportamiento característico para cada *framework*:

- DEAP muestra una curva cóncava ( $R^2 = 0,23$ ), con eficiencia creciente al pasar de  $2^6$  a  $2^{10}$  y una posterior caída al llegar a  $2^{14}$ . Los resultados sugieren la necesidad de encontrar un equilibrio entre tiempo de ejecución y consumo energético al seleccionar parámetros del algoritmo (Fernández de Vega *et al.*, 2016, pp. 554-555).
- Paradiseo denota una tendencia casi plana ( $R^2 = 0,11$ ), lo que indica que su eficiencia se mantiene estable independientemente de la población, favoreciendo uniformidad en escenarios variables.

- Inspyred presenta un pico agudo en  $N = 2^{10}, p_c=0,8$  seguido de un fuerte descenso en  $2^{14}$  ( $R^2 = 0,35$ ), reflejo de que su modelo es muy sensible al balance entre exploración y explotación.
- ECJ ( $R^2 = 0,38$ ) casi no modifica su eficiencia con la población, lo que implica que su coste energético por iteración y su ganancia en *fitness* escalan de forma casi equivalente.

Tanto DEAP como Inspyred y Paradiseo alcanzan sus mayores  $\eta$  en poblaciones de tamaño  $2^{10}$ .

Figura 22: comparación entre tendencias de *frameworks* del cálculo de  $\eta$  de OneMax en el portátil



Fuente: elaboración propia

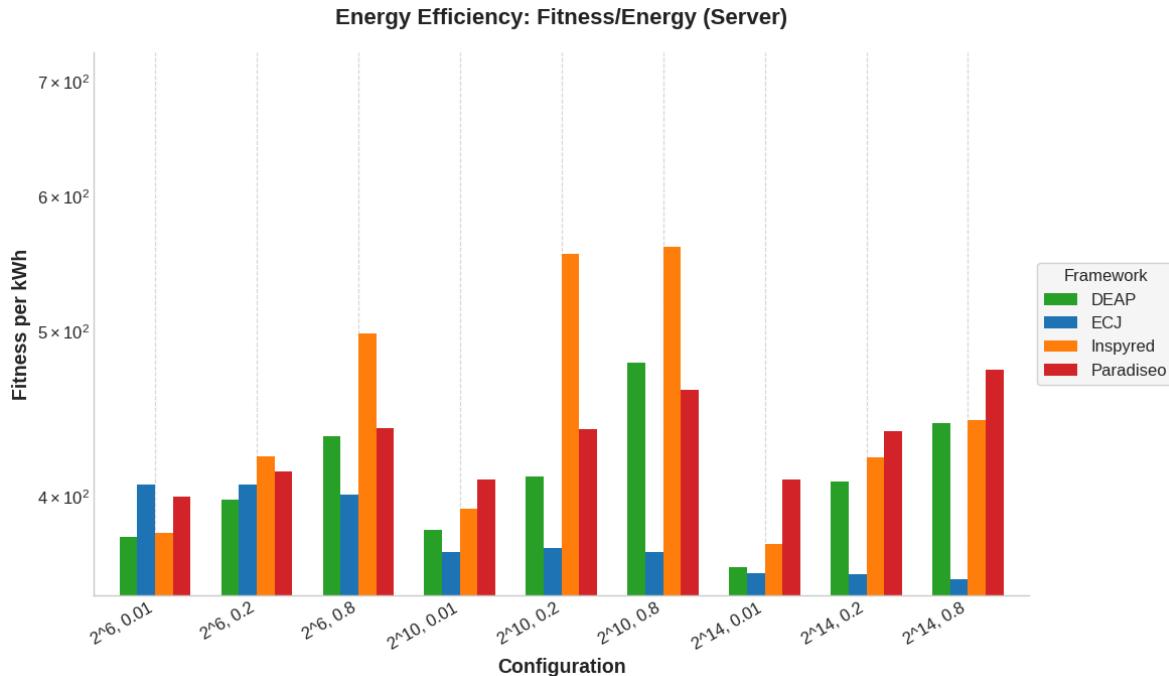
#### 4.1.11. Análisis de la eficiencia energética en el servidor

Usando la misma métrica que en el apartado anterior, se puede observar en las figuras 23 y 24 que:

- Inspyred alcanza la mayor eficiencia del servidor, con picos de  $560 \text{ fitness/kWh}$  en las configuraciones de población intermedia ( $2^{10}, p_c = 0,2$  y  $2^{10}, p_c = 0,8$ ). Este elevado  $\eta$  refleja una buena relación entre ganancia de *fitness* y gasto energético.
- Paradiseo ocupa el segundo lugar, con valores que oscilan entre 400 y 470  $\text{fitness/kWh}$ , mostrando una eficiencia relativamente constante y una tendencia polinómica moderada ( $R^2 = 0,42$ ).
- DEAP, con  $\eta$  entre 375 y 480, presenta una tendencia casi horizontal, lo que indica que su eficiencia no se ve muy afectada por el tamaño de la población, aunque mejora ligeramente en  $N = 2^{10}, p_c = 0,8$ .

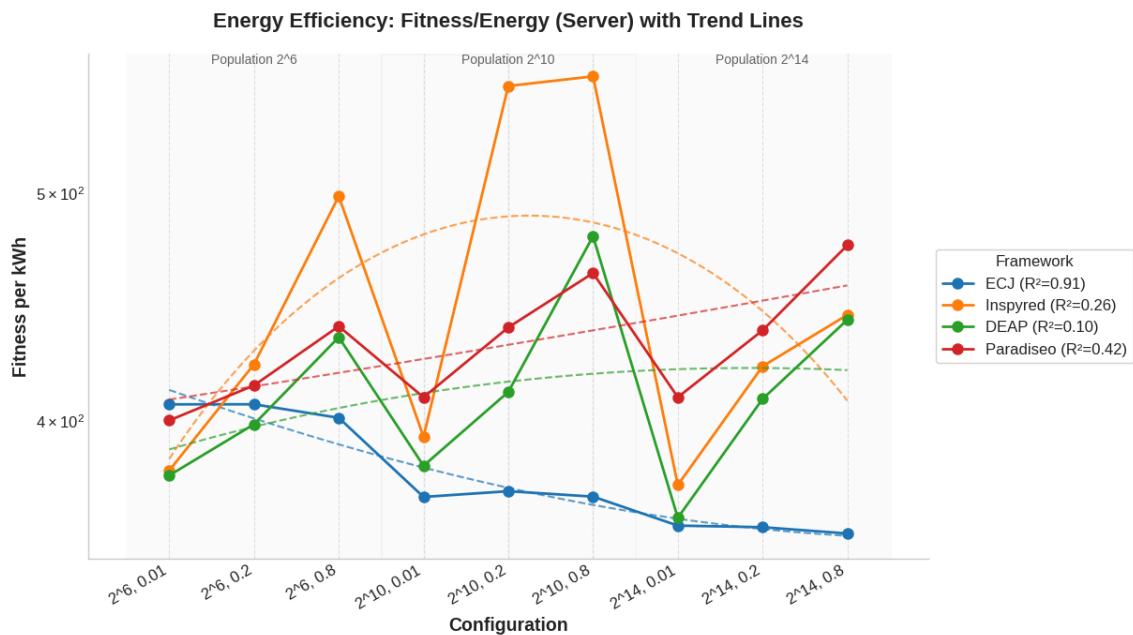
- ECJ, a pesar de completar más generaciones por unidad de tiempo, ofrece la eficiencia más baja ( $\eta \approx 385-410$ ), y muestra una clara caída de eficiencia al crecer la población, evidenciando que su consumo de energía aumenta más rápido que la ganancia de rendimiento.

Figura 23: comparación entre *frameworks* del cálculo de  $\eta$  de OneMax en el servidor



Fuente: elaboración propia

Figura 24: comparación entre tendencias de *frameworks* del cálculo de  $\eta$  de OneMax en el servidor



Fuente: elaboración propia

La línea de tendencia polinomial para Inspyred revela un incremento acentuado de eficiencia en la transición de poblaciones pequeñas ( $2^6$ ) a medianas ( $2^{10}$ ), seguido de un leve

descenso al llegar a  $2^{14}$ . Esto sugiere que, en el servidor, existe igualmente un tamaño poblacional óptimo donde el paralelismo disponible compensa con creces el coste de sincronización y comunicación interna. En contraste, ECJ sufre un decrecimiento lineal de  $\eta$  con el aumento de la población.

#### 4.1.12. Análisis de la eficiencia energética del portátil frente al servidor

A continuación, se comparan los resultados obtenidos en el portátil y en el servidor usando la misma métrica que la expuesta en el punto 4.1.10. Como se puede observar en las figuras 25 y 26, en todas las configuraciones y *frameworks*, el portátil multiplica por cinco o, incluso, por siete la eficiencia de la misma combinación en el servidor. Para población  $N = 2^{10}$  y  $p_c = 0,8$ :

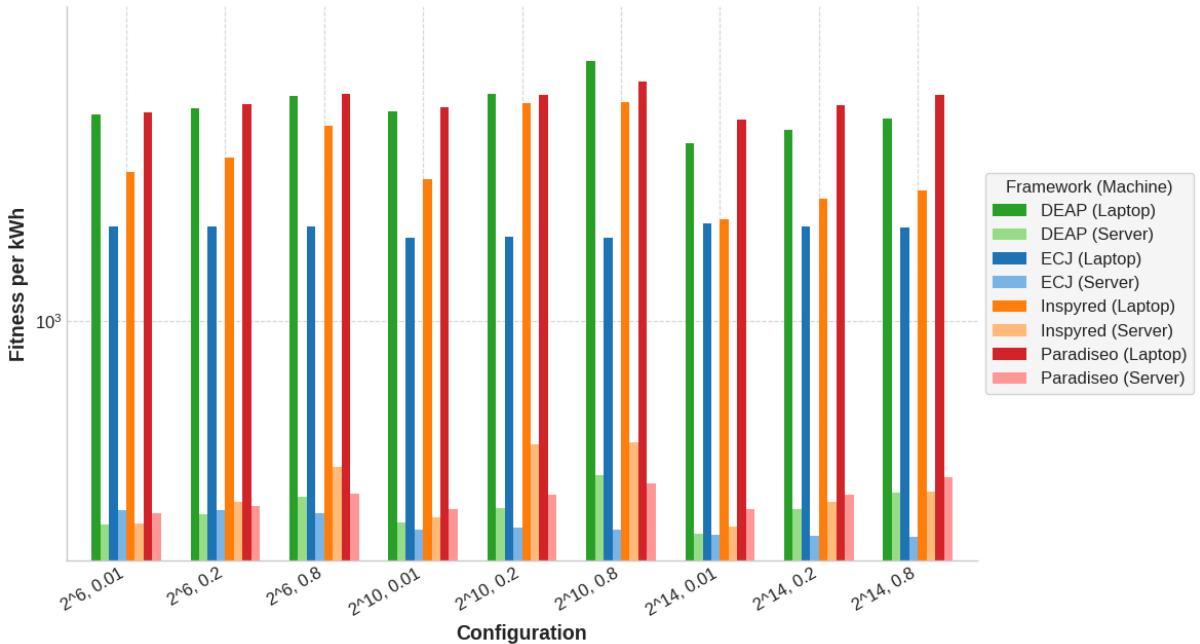
Tabla 4: Ejemplo de comparación de  $\eta$  de OneMax para  $N = 2^{10}$  y  $p_c = 0,8$  en el portátil y el servidor

<i>Framework</i>	$\eta$ (portátil)	$\eta$ (servidor)	Factor multiplicativo
DEAP	3.480	480	7,25
ParadisEO	2.950	460	6,41
Inspyred	2.800	560	5
ECJ	1.300	380	3,42

Fuente: elaboración propia

Figura 25: comparación entre *frameworks* del cálculo de  $\eta$  de OneMax en el portátil y el servidor

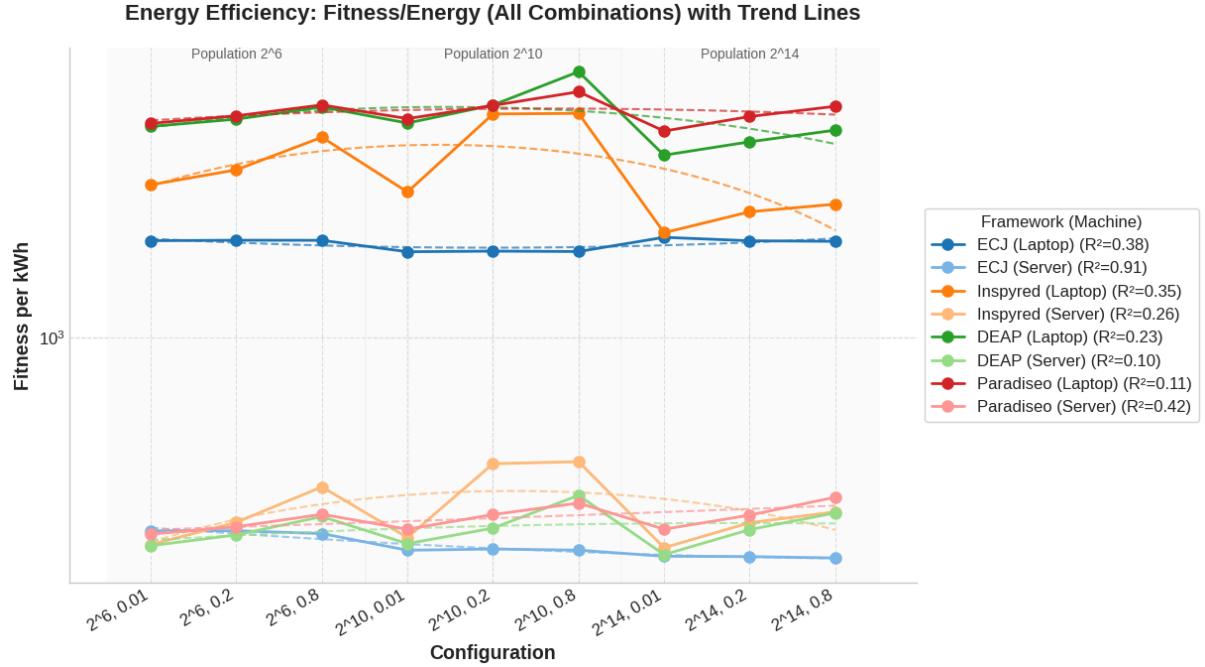
Energy Efficiency: Fitness/Energy (All Combinations)



Fuente: elaboración propia

Este salto refleja cómo la sobrecarga de gestión de paralelismo masivo en el servidor penaliza drásticamente la rentabilidad energética, pese a su mayor número de iteraciones por unidad de tiempo.

Figura 26: comparación entre tendencias de *frameworks* del cálculo de  $\eta$  de OneMax en el portátil y en el servidor



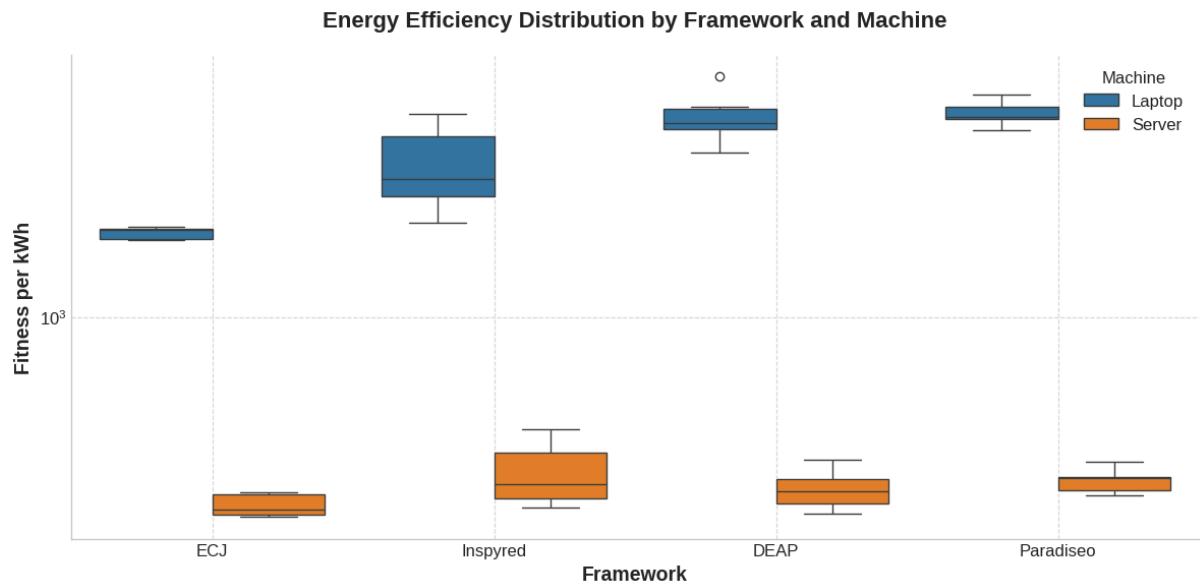
Fuente: elaboración propia

Las líneas de tendencia polinomiales indican que todas las curvas sitúan su máximo  $\eta$  en poblaciones  $N = 2^{10}$ .

Este análisis conjunto subraya que, aunque el servidor acelera el cómputo puro, no compensa el sobrecoste energético por generación. El portátil, en combinación con *frameworks* bien optimizados (especialmente DEAP y Paradiseo), ofrece la mejor eficiencia *fitness/kWh*, reforzando la necesidad de considerar métricas energéticas como criterio principal al diseñar y seleccionar plataformas y EA para problemas de optimización.

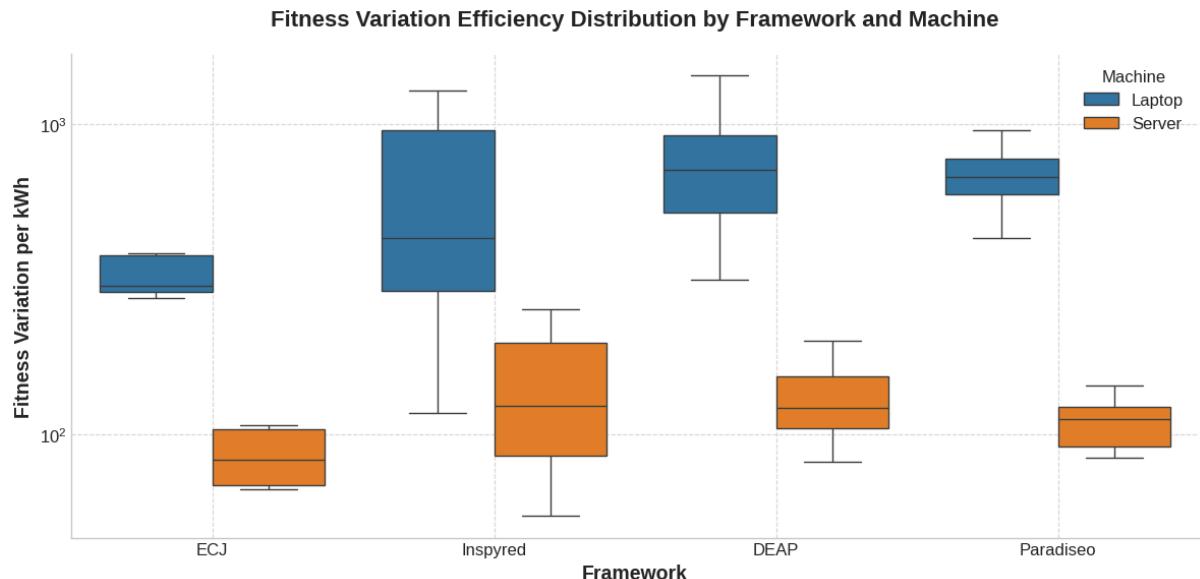
#### 4.1.13. Distribución de eficiencia energética y de variación de *fitness*

Figura 27: distribución de la eficiencia energética de OneMax en portátil y servidor



Fuente: elaboración propia

Figura 28: distribución de la variación del *fitness* de OneMax en portátil y servidor



Fuente: elaboración propia

Para completar el análisis de eficiencia, se examina la distribución de las métricas de *fitness* por kWh (figura 27) y de variación de *fitness* por kWh (figura 28) agrupadas por *framework* y por máquina (portátil frente servidor). Estas representaciones en “*boxplot*” permiten apreciar la consistencia y variabilidad de cada combinación a lo largo de todas las configuraciones evaluadas. Cabe así destacar, por *framework*:

- ECJ: En el portátil, muestra una caja muy estrecha alrededor de los 1.250 y los 1.350 *fitness/kWh*, con whiskers cortos, lo que indica que su eficiencia es prácticamente invariable entre poblaciones y probabilidades de cruce. En el servidor, la mediana ronda los 400 *fitness/kWh*, con variaciones igualmente pequeñas. Este bajo rango de

dispersión sugiere que la implementación de ECJ es estable pero poco adaptable a distintos ajustes, coherente con su escaso escalado energético observado previamente.

- Inspyred: presenta la caja más ancha en ambos entornos, especialmente en el portátil, donde la eficiencia oscila entre 2.000 y 2.900 *fitness/kWh*. En el servidor, su rango va de 350 a 580 *fitness/kWh*, reflejando un comportamiento muy sensible al tamaño de la población. Esta alta dispersión confirma que Inspyred alcanza picos de eficiencia en poblaciones intermedias pero sufre diámetros considerables en extremos poblacionales.
- DEAP: caja moderada en la portátil (en torno a los 2.600 y los 3.400 *fitness/kWh*) y en el servidor (aproximadamente unos 370 y 480 *fitness/kWh*), con whiskers medianos. Esto indica un buen equilibrio entre eficiencia y consistencia.
- Paradiseo: en el portátil se observa una distribución estrecha alrededor de 2.700 y 3.100 *fitness/kWh*, mientras que en el servidor oscila entre 400 y 480 *fitness/kWh*, mostrando la mayor homogeneidad tras ECJ.

## 4.2. Sphere

En esta subsección se repetirá la misma estructura de análisis del experimento OneMax, pero aplicándolo al *benchmark* Sphere.

### 4.2.1. Consumo energético en el portátil

Como se puede observar en la figura 29, la energía consumida entre los *frameworks* es diferente. Por un lado, DEAP oscila entre 0,000445 kWh ( $N = 2^6$ ,  $p_c = 0,2$ ) y 0,000478 kWh ( $N = 2^{10}$ ,  $p_c = 0,01$ ), situándose como el más costoso. Lo sigue Inspyred, el cual se mueve de 0,00032 kWh ( $N = 2^6$ ,  $p_c = 0,01$ ) a 0,00035 kWh ( $N = 2^{14}$ ,  $p_c = 0,8$ ). En tercer y cuarto lugar se sitúan ECJ y ParadisEO, con unos valores de consumo bastante estables, en torno a 0,000256 kWh y 0,000255 kWh, respectivamente.

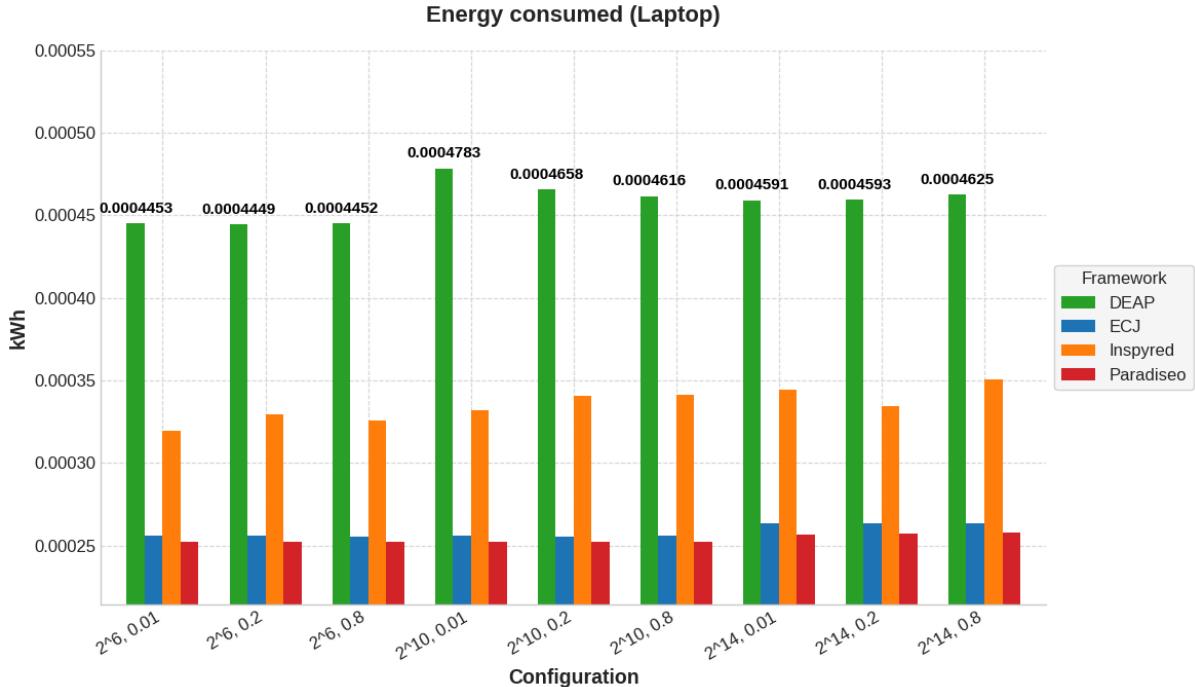
Esto revela que DEAP es el más costoso en cuanto la consumo se refiere, con picos por encima de 0,00047 kWh cuando la población es de tamaño medio, mientras que ECJ y ParadisEO se sitúan consistentemente por debajo de 0,00026 kWh, es decir, suponen casi la mitad del consumo de DEAP.

Poniendo la mirada en las líneas de tendencia polinomial sobre el consumo, visibles en la figura 30, se observa que:

- ECJ y ParadisEO registran un escalado casi lineal y suave, con un  $R^2 = 0,81$  y  $R^2 = 0,87$ , respectivamente, lo que sugiere variaciones en el consumo energético proporcionales al tamaño de población, atribuibles a la utilización de la jerarquía de memoria (Fernández de Vega *et al.*, 2016, p. 374).
- Inspyred muestra una pendiente positiva notable, con un  $R^2 = 0,77$ , con un consumo creciente conforme la población crece, lo que indica mayor sobrecarga al gestionar individuos más grandes.

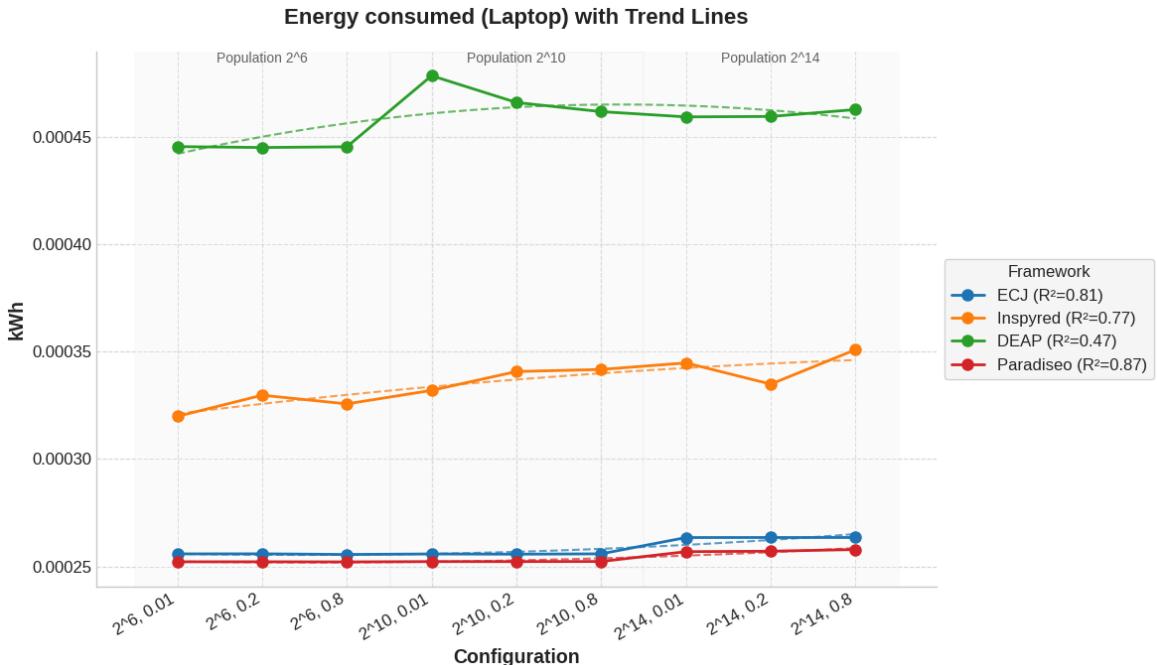
- DEAP exhibe una pendiente que alcanza su máximo en  $2^{10}$ , y descendiendo ligeramente en  $2^{14}$ , lo que podría atribuirse a una amortización de costes de arranque frente al tamaño poblacional.

Figura 29: energía consumida por el portátil en las ejecuciones de Sphere



Fuente: elaboración propia

Figura 30: tendencia de la energía consumida por el portátil en las ejecuciones de Sphere



Fuente: elaboración propia

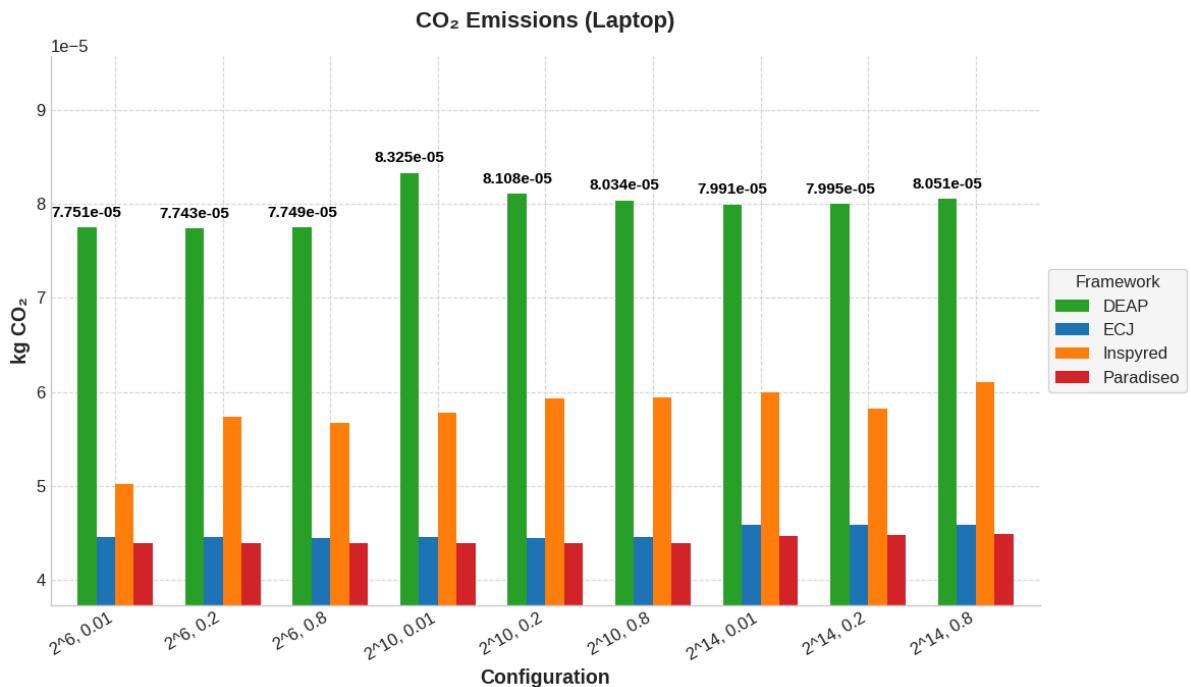
Para poblaciones pequeñas, de tamaño  $2^6$ , todos los *frameworks* consumen alrededor de 0,000445 kWh (DEAP), 0,000320 kWh (Inspyred), 0,000256 kWh (ECJ) y 0,000252 kWh

(ParadisEO). Al pasar a poblaciones intermedias, de tamaño  $2^{10}$ , DEAP aumenta aproximadamente un 7 % su consumo, Inspyred un 4 %, mientras que ECJ y ParadisEO apenas varían un 1 %. Al llegar a poblaciones más grandes, de tamaño  $2^{14}$ , DEAP decrece ligeramente hasta un 4 % respecto al pico, mostrando un efecto de amortiguación, mientras Inspyred se sitúa en su valor más alto y tanto ECJ como ParadisEO incrementa su consumo sólo un 3 %.

En conjunto, el portátil presenta un comportamiento muy cercano al paradigma de computación energéticamente proporcional, especialmente en ECJ y Paradiseo, mientras que Inspyred y DEAP muestran mayores sobrecostes al escalar población, con DEAP alcanzando el mayor consumo absoluto en configuraciones intermedias.

#### 4.2.2. Emisiones totales de CO<sub>2</sub> en el portátil

Figura 31: emisiones totales de CO<sub>2</sub> por el portátil en las ejecuciones de Sphere



Fuente: elaboración propia

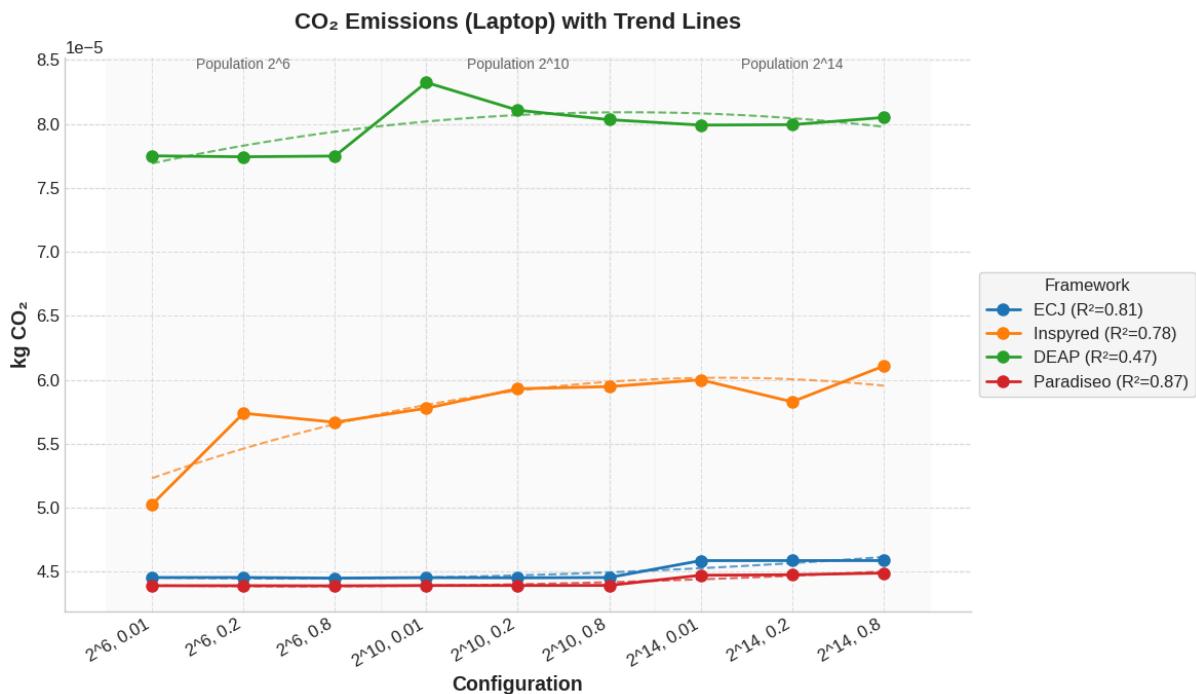
Según se puede observar en la figura 31, las emisiones de carbono coinciden con la energía consumida vista en la sección anterior.

Por un lado, DEAP lidera con diferencia las emisiones en todas las configuraciones. En las configuraciones con  $N = 2^6$  y  $N = 2^{11}$  sus emisiones oscilan, de forma casi constante, entre 0,00007743 kg CO<sub>2</sub> y 0,00008108 kg CO<sub>2</sub> con la salvedad de tener un pico en la primera de las configuraciones de la población intermedia, alcanzando los 0,00008235 kg CO<sub>2</sub>. Al alcanzar el mayor tamaño de población, la emisión desciende hasta 0,00007991 kg CO<sub>2</sub>, lo que supone un descenso del 3,9 %, tal y como se aprecia también en la tendencia polinomial de la figura 32.

Por otro lado, Inspyred se mantiene en segunda posición en la cantidad de carbono emitido, con un incremento progresivo desde 0,00005001 kg CO<sub>2</sub> hasta 0,00006103 kg CO<sub>2</sub> a medida que el tamaño de la población aumenta. Puede, igualmente, observarse un leve pico en la configuración  $N = 2^{14}$ ,  $p_c = 0,01$  de 0,00005946 kg CO<sub>2</sub>, también visto en el apartado anterior, indicando que los accesos a memoria y los intérpretes Python se combinan de forma no lineal a mayor tamaño de la población.

Finalmente, ECJ y Paradiseo constituyen las opciones menos contaminantes, con emisiones muy controladas que oscilan entre 0,00004445 kg CO<sub>2</sub> y 0,0000446 kg CO<sub>2</sub> en el caso de ECJ, y entre 0,0000438 kg CO<sub>2</sub> y 0,000045 kg CO<sub>2</sub> en el de Paradiseo, evidenciando un escalado casi lineal y una baja variabilidad entre configuraciones.

Figura 32: tendencia de las emisiones totales de CO<sub>2</sub> por el portátil en las ejecuciones de Sphere



Fuente: elaboración propia

#### 4.2.3. Evolución del *fitness* en el portátil

Para representar el comportamiento evolutivo y la calidad de las soluciones se han propuesto cuatro métricas a evaluar: *fitness* inicial, variación del *fitness*, *fitness* máximo alcanzado y número de generaciones alcanzadas.

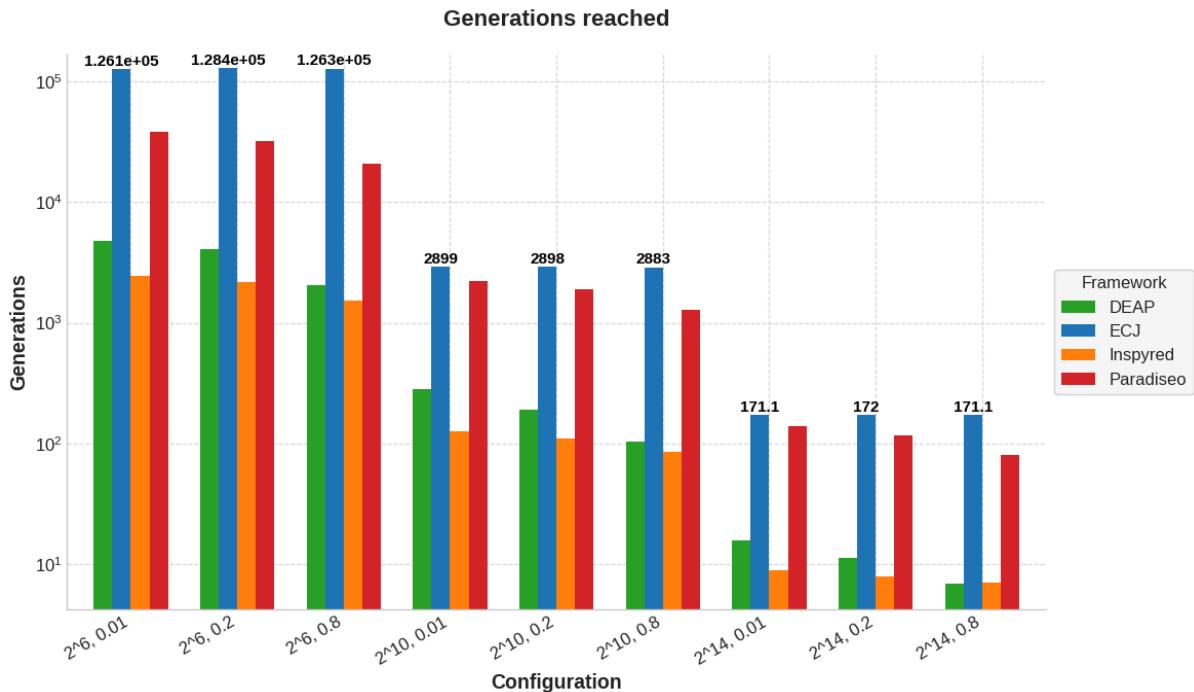
En primer lugar, se exponen en la figura 33 tres órdenes de magnitud de diferencia en el número de generaciones alcanzado:

- Con  $N = 2^6$  los *frameworks* alcanzan casi 126.000 generaciones en el caso de ECJ o casi 3000 y 1600 generaciones en el caso de DEAP e Inspyred, estando ParadisEO entre ambos extremos, alcanzando casi 27.000. Estas generaciones se alcanzan antes de cumplir los dos minutos de ejecución, condición de parada ya expuesta.

- Al crecer la población a  $N = 2^{10}$  el número máximo de generaciones alcanzado es de casi 2.900 para ECJ, de 200 y 120 aproximadamente para DEAP e Inspyred y de 1.200 para ParadisEO.
- Al llegar la población a  $N = 2^{14}$  el número de generaciones se reduce hasta las 170 aproximadamente en el caso de ECJ y a 9 en el caso de Inspyred, siendo ambos, respectivamente, el extremo máximo y mínimo de número de generaciones alcanzadas en todas las configuraciones.

Esta reducción tan drástica confirma una “ley inversa” existente entre generaciones alcanzadas y tamaño de la población en la cual explican que para una condición de parada similar, los mecanismos de variación operan menos veces cuanto mayor sea el tamaño de la población.

Figura 33: número máximo de generaciones alcanzadas de Sphere



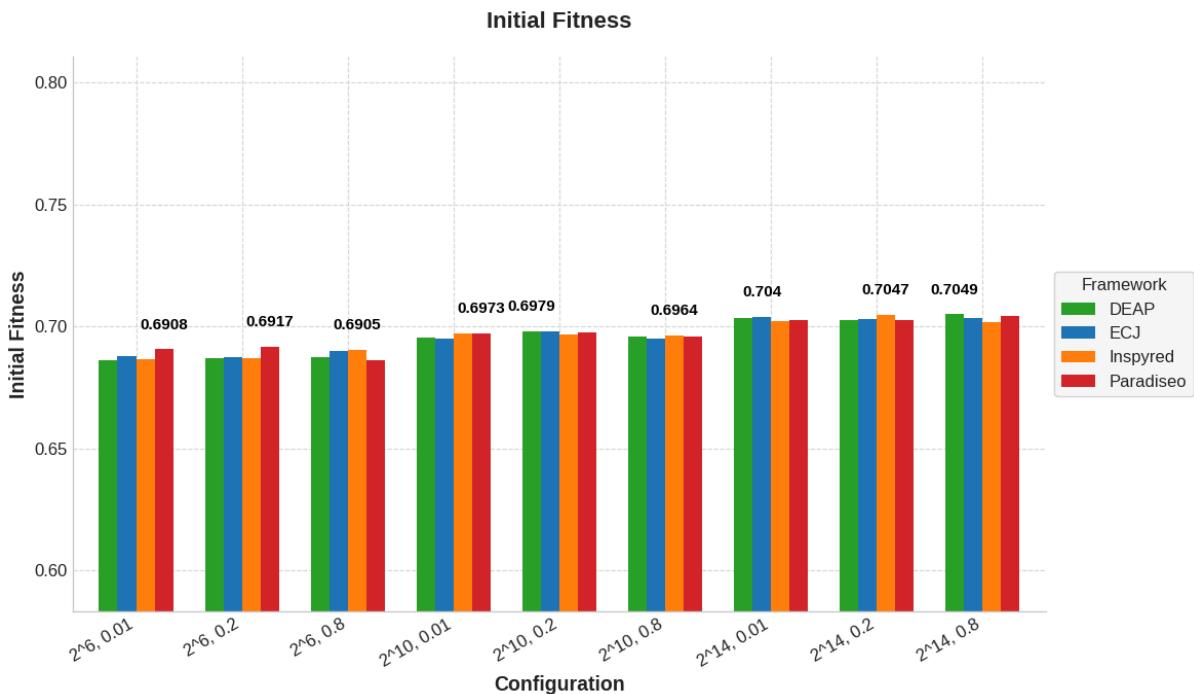
Fuente: elaboración propia

En segundo lugar, en la figura 34 se muestra que ECJ y ParadisEO parten con un *fitness* inicial promedio de 0,689 y 0,691 mientras que DEAP e Inspyred parten de un *fitness* inicial promedio un poco menor, rondando el 0,687.

En tercer lugar, en la figura 35 se muestra la variación del *fitness*, donde es lógico pensar que, con un mayor porcentaje de probabilidad de cruce ( $p_c$ ), la variación tenderá a ser mayor. Esto se puede observar en las tres configuraciones que tienen esa probabilidad, viendo que dentro del mismo tamaño de población, donde se alcanza mayor variación es cuando se tiene una  $p_c = 0,8$ . Así, se puede destacar que:

- Inspyred sobresale cuando la población es de tamaño medio, dado que su política de mutación explora rápido la diversidad inicial y alcanza la mayor subida, de casi 0,28 puntos. Aún así, cuando la población es  $N = 2^{14}$ , casi no hay variación con  $p_c = 0,01$ .
- ECJ mejora aunque se mantiene casi constante dentro de un mismo tamaño de población, independientemente de la probabilidad de cruce, reduciendo ligeramente la variación según aumenta el tamaño de la población, pero siendo el mejor *framework* para poblaciones grandes; el ajuste lineal ( $R^2 = 0,93$ ) muestra que su variación se reduce casi monótonamente con el tamaño de la población.
- DEAP sobresale cuando el tamaño de la población es más pequeño ( $N = 2^6$ ), siendo el que consigue mayor variación frente al resto de *frameworks*, alcanzando dicha variación en torno a 0,28 puntos.

Figura 34: *fitness* inicial promedio de Sphere

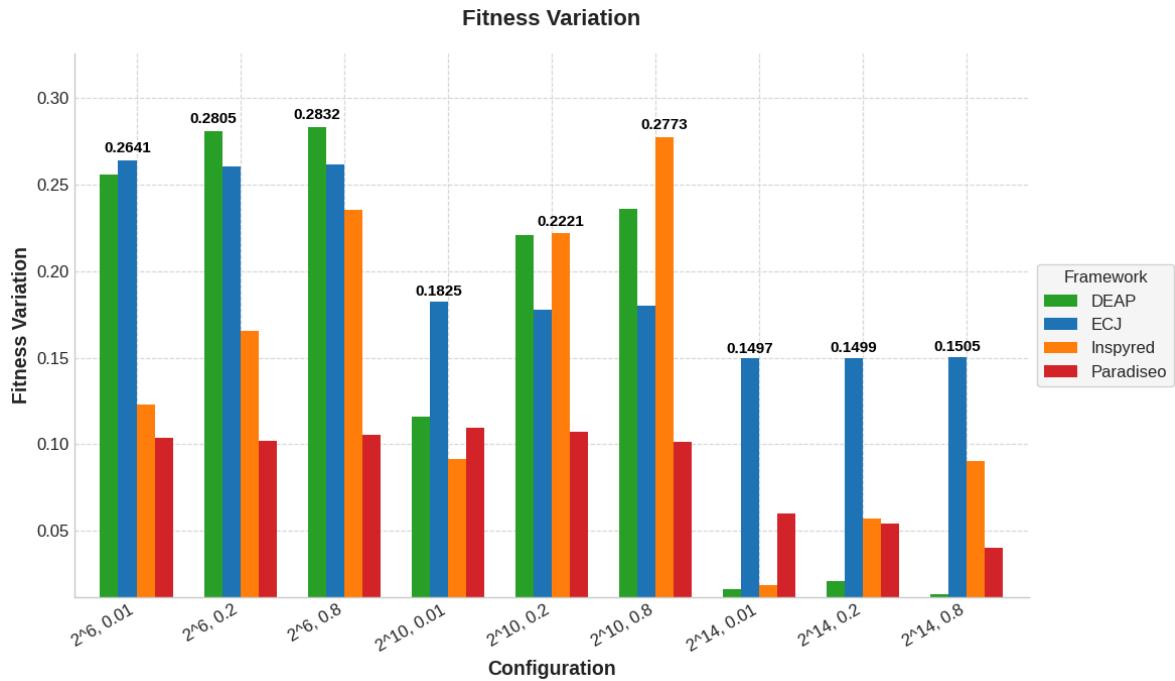


Fuente: elaboración propia

Por último, en la figura 36 se muestra la calidad de la solución alcanzada, que en general se maximiza con  $N = 2^{10}$  y  $p_c = 0,8$ :

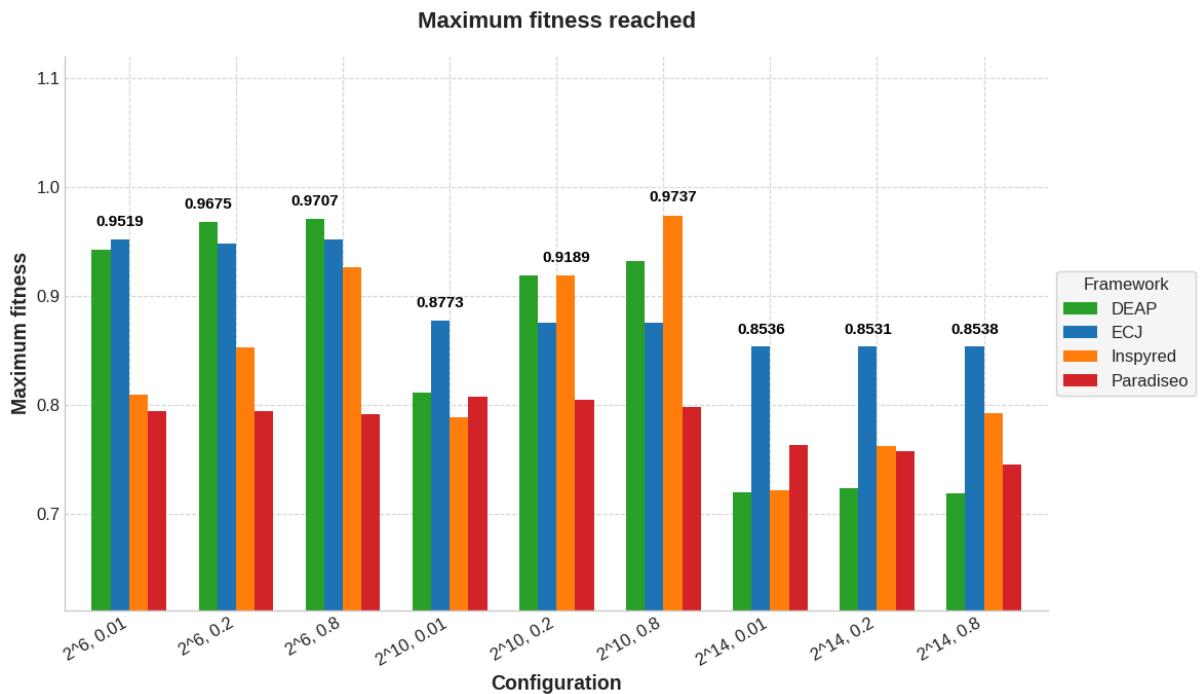
- Inspyred logra el *fitness* máximo más alto, 0,9737, muy próximo al óptimo teórico.
- DEAP le sigue con 0,9707 (en  $N = 2^6$ ,  $p_c = 0,8$ ) y ECJ alcanza 0,952 en esa misma configuración.
- En  $N = 2^{14}$ , ECJ es el líder relativo con 0,8538, mientras que Paradiseo consigue 0,7438, beneficiándose de la mayor variación residual en esa franja.

Figura 35: variación promedio del *fitness* de Sphere



Fuente: elaboración propia

Figura 36: *fitness* máximo promedio alcanzado de Sphere



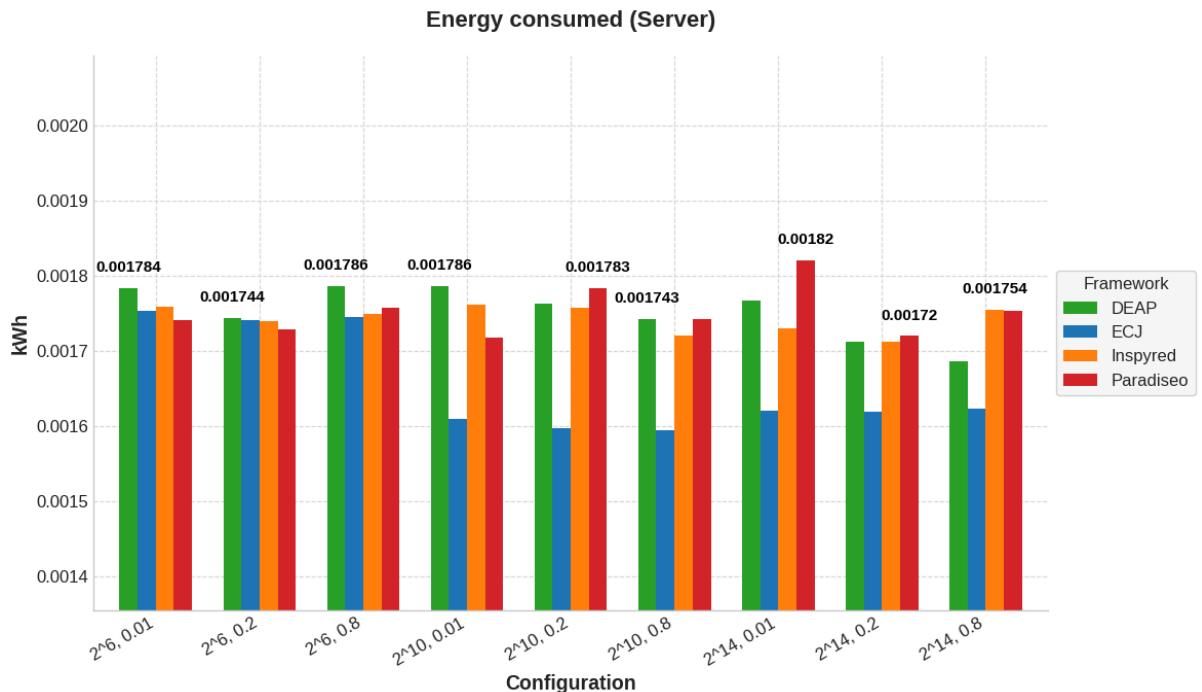
Fuente: elaboración propia

#### 4.2.4. Consumo energético en servidor

Como se puede apreciar en la figura 37, el orden jerárquico de consumo en el servidor se mantiene invariable a lo largo de las nueve configuraciones, aunque con claras diferencias respecto al portátil:

- DEAP es el que, en términos globales, consume más energía, con valores que oscilan de 0,001786 kWh ( $N = 2^6, p_c = 0,8$ ) a 0,001689 kWh ( $N = 2^{14}, p_c = 0,8$ ).
- Inspyred queda justo por debajo, arrancando en 0,001756 kWh ( $N = 2^6, p_c = 0,01$ ) y descendiendo hasta 0,001743 kWh ( $N = 2^{10}, p_c = 0,8$ ).
- Paradiseo muestra el consumo más estable, entre 0,001716 kWh ( $N = 2^{10}, p_c = 0,01$ ) y 0,001820 kWh ( $N = 2^{14}, p_c = 0,01$ ), con la salvedad de ser ese valor un visible pico.
- ECJ, por su parte, es el menos costoso, variando desde 0,001754 kWh ( $N = 2^6, p_c = 0,01$ ) hasta 0,001596 kWh ( $N = 2^{10}, p_c = 0,2$ ).

Figura 37: energía consumida por el servidor en las ejecuciones de Sphere



Fuente: elaboración propia

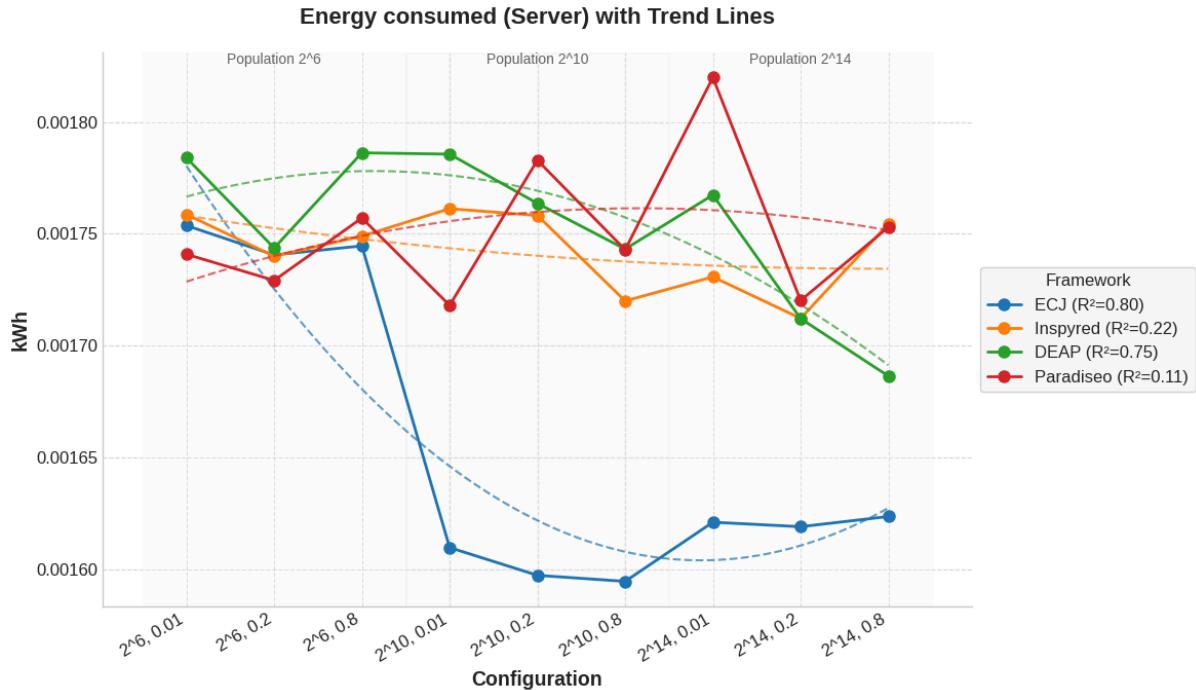
La diferencia absoluta entre el más eficiente ( $N = 2^{10}, p_c = 0,2$ ) y el más costoso (DEAP en  $N = 2^6, p_c = 0,8$ ) es de 0,000190 kWh, apenas un 13 %, muy lejos del casi 60 % visto en portátil. Esto refleja cómo el overhead de arranque de la JVM o de CPython se amortiza al distribuirse de manera diferente según el tamaño de la población: poblaciones más grandes pueden amortizar mejor el overhead computacional, distribuyendo el costo fijo entre más evaluaciones de fitness (Fernández de Vega *et al.*, 2016, pp. 554-555).

Además, en la figura 38 se aprecian las tendencias polinomiales:

- DEAP ( $R^2 = 0,75$ ) muestra una leve pendiente descendente al aumentar  $N$ , lo que indica que la sobrecarga inicial se diluye en menos generaciones cuando la población crece.
- Inspyred ( $R^2 = 0,22$ ) dibuja una curva casi plana, con un ligero descenso hasta la población intermedia y mínima variación en el tramo final.

- ECJ ( $R^2 = 0,8$ ) experimenta una caída marcada del consumo por generación desde  $2^6$  hasta  $2^{10}$ , recuperándose luego en  $2^{14}$ , lo que evidencia un escalado energético eficiente gracias al código compilado y a la amortización de la JVM.
- Paradiseo ( $R^2 = 0,11$ ) mantiene un comportamiento prácticamente horizontal, beneficiándose de un perfil de ejecución de bajo overhead desde el inicio.

Figura 38: tendencia de la energía consumida por el servidor en las ejecuciones de Sphere



Fuente: elaboración propia

En conjunto, estos resultados confirman que, en el servidor, la potencia de cálculo tiende a homogeneizar el consumo entre *frameworks*, reduciendo significativamente las brechas energéticas vistas en entornos portátiles.

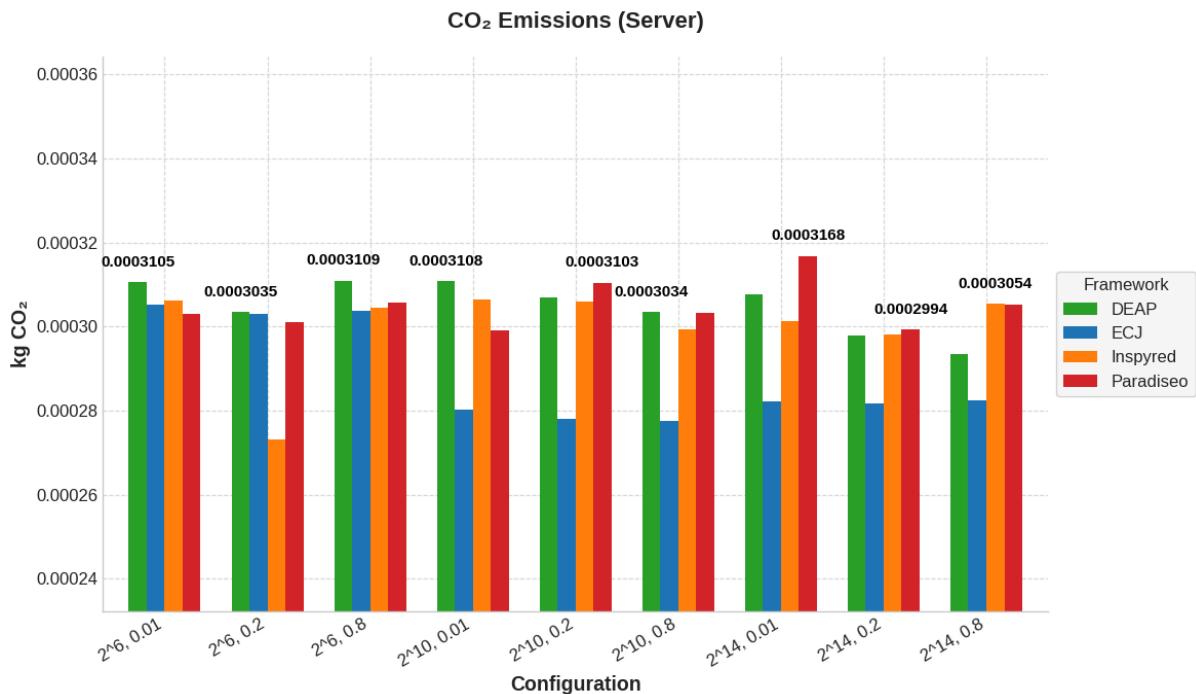
#### 4.2.5. Emisiones totales de CO<sub>2</sub> en servidor

En la figura 39 se expone el orden jerárquico de emisiones de CO<sub>2</sub> en el servidor. En las nueve configuraciones ejecutadas ( $N \in \{2^6, 2^{10}, 2^{14}\}$  y probabilidad de cruce  $p_c \in \{0.01, 0.2, 0.8\}$ ), se observa que:

- DEAP ocupa el primer puesto en siete de las nueve configuraciones, con emisiones que aumentan suavemente desde 0,0003105 kg CO<sub>2</sub> ( $N = 2^6, p_c = 0,01$ ) hasta 0,0002935 kg CO<sub>2</sub> ( $N = 2^{14}, p_c = 0,8$ ). La tendencia polinomial ( $R^2 = 0,75$ ) confirma que, al copiar arrays NumPy cada vez mayores, el servidor penaliza ligeramente en consumo energético y, por tanto, en emisiones.

- Inspyred queda en segundo lugar, partiendo de 0,0003055 kg CO<sub>2</sub> ( $N = 2^6$ ,  $p_c = 0,01$ ), cayendo hasta un mínimo de 0,0002735 kg CO<sub>2</sub> ( $N = 2^6$ ,  $p_c = 0,2$ ) y repuntando hasta 0,0003054 kg CO<sub>2</sub> ( $N = 2^8$ ,  $p_c = 0,8$ ). Su línea de tendencia es prácticamente plana ( $R^2 = 0,07$ ), lo que sugiere que el overhead de inicialización se amortiza cuando el número de generaciones decrece.
- Paradiseo exhibe la banda más estrecha de emisiones, entre 0,000301 y 0,0003168 kg CO<sub>2</sub>, con un  $R^2 = 0,11$ , que denota un escalado casi lineal y un perfil de ejecución de bajo *overhead*.
- ECJ es el menos emisivo, con valores entre 0,000278 y 0,0003035 kg CO<sub>2</sub>, y una pendiente moderada ( $R^2 = 0,8$ ) que refleja la amortización del warm-up de la JVM frente a la elevada potencia sostenible de la CPU i9.

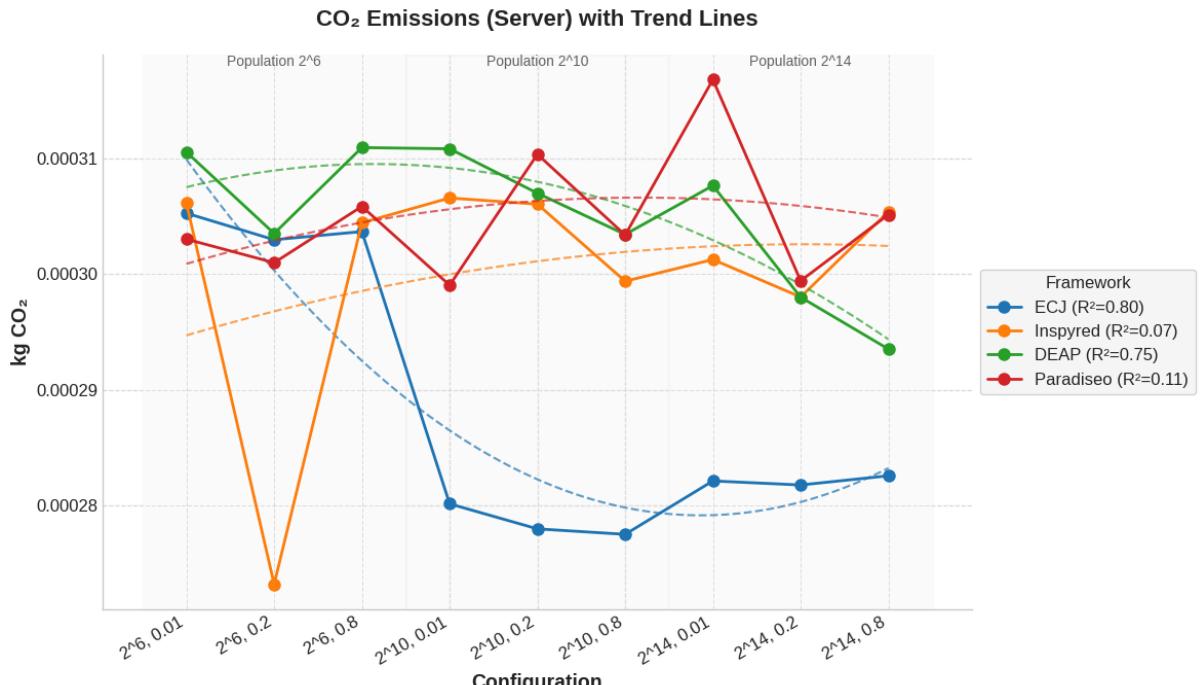
Figura 39: emisiones totales de CO<sub>2</sub> por el servidor en las ejecuciones de Sphere



Fuente: elaboración propia

La brecha absoluta entre el más contaminante (DEAP) y el más limpio (ECJ) es de 0,000325 kg CO<sub>2</sub>, aproximadamente un 14 % del valor máximo, muy lejos del casi 60 % observado en portátil, aunque en términos absolutos las emisiones del servidor duplican las de la plataforma portátil. Este estrechamiento relativo confirma que, a mayor paralelismo (24 hilos), el coste fijo de arranque y sincronización se reparte mejor, reduciendo la variabilidad energética entre *frameworks*.

Figura 40: tendencia de las emisiones totales de CO<sub>2</sub> por el servidor en las ejecuciones de Sphere

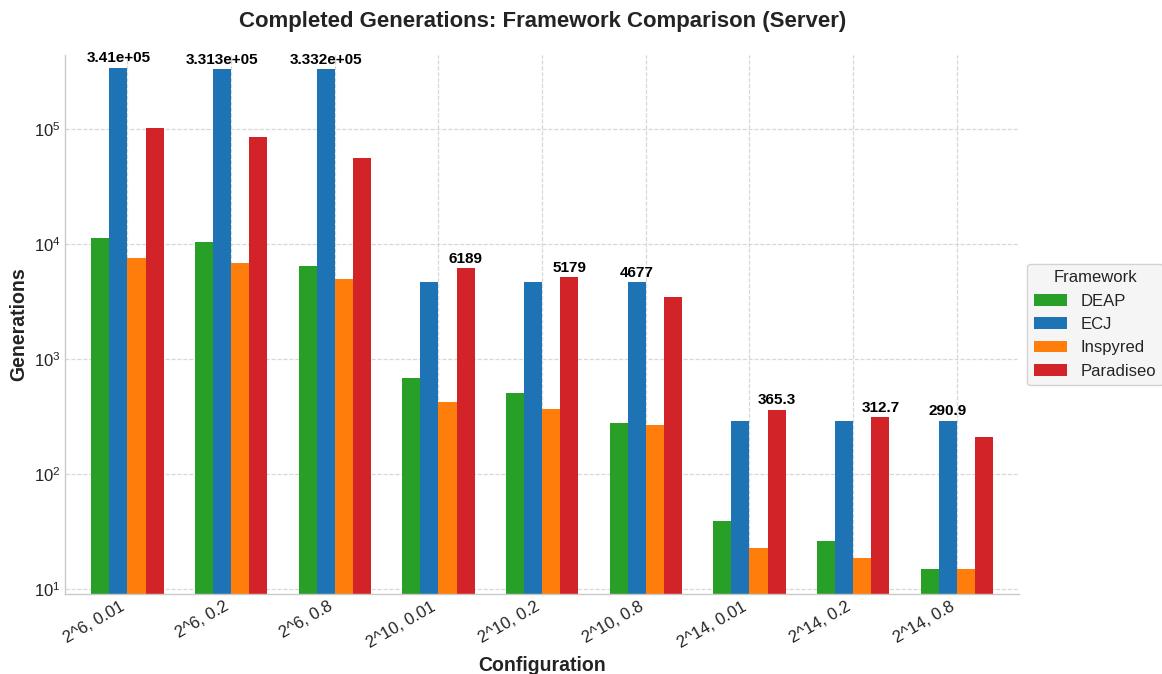


Fuente: elaboración propia

#### 4.2.6. Evolución del *fitness* en servidor

Para representar el comportamiento evolutivo y la calidad de las soluciones se han propuesto cuatro métricas a evaluar: *fitness* inicial, variación del *fitness*, *fitness* máximo alcanzado y número de generaciones alcanzadas.

Figura 41: número máximo de generaciones alcanzadas de Sphere



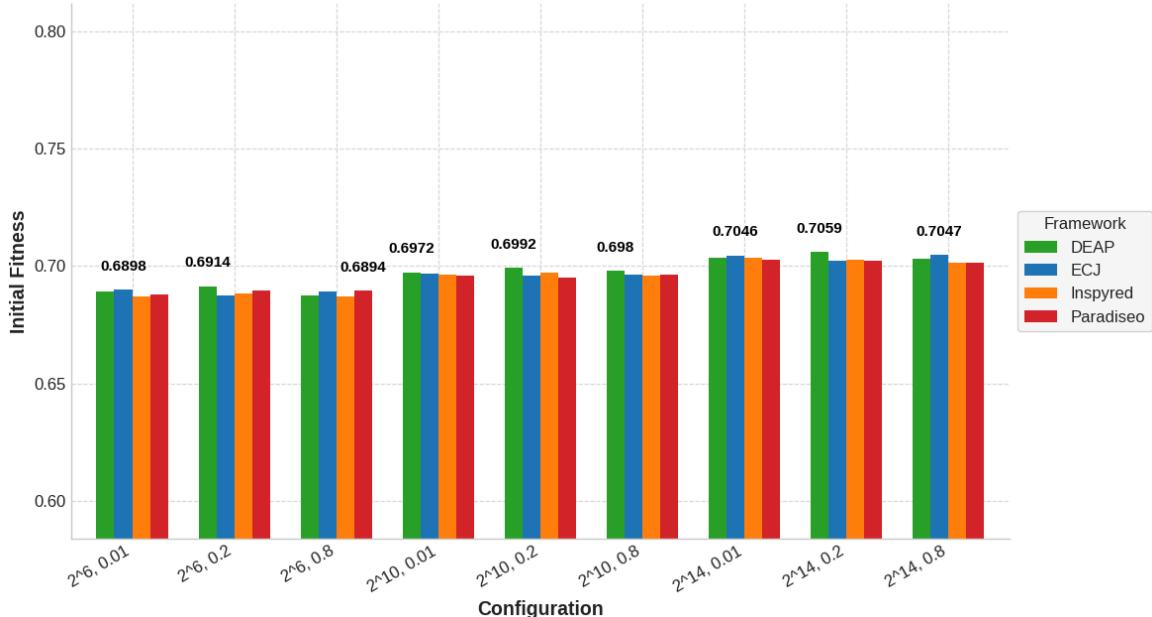
Fuente: elaboración propia

En primer lugar, se exponen en la figura 41 tres órdenes de magnitud de diferencia en el número de generaciones alcanzado:

- Con  $N = 2^6$  los *frameworks* alcanzan unas 330.000 generaciones aproximadamente en el caso de ECJ o casi 10.000 generaciones en el caso de DEAP e Inspyred, estando ParadisEO entre ambos, alcanzando casi 100.000. Estas generaciones se alcanzan antes de cumplir los dos minutos de ejecución, condición de parada expuesta.
- Al crecer la población a  $N = 2^{10}$  el número máximo de generaciones cae drásticamente hasta casi alcanzar 5.000 para ECJ, entre 800 y 500 aproximadamente para DEAP e Inspyred y 6.000 para ParadisEO. En este tamaño de población se observa cómo ECJ deja de ser el *framework* dominante para darle paso a ParadisEO en dos de las tres posibles configuraciones.
- Al llegar la población a  $N = 2^{14}$  el número de generaciones se reduce hasta las 300 aproximadamente en el caso de ECJ y a 30 en el caso de Inspyred, siendo ambos, respectivamente, de media, el extremo máximo y mínimo de número de generaciones alcanzadas en todas las configuraciones, aunque, en dos de las tres configuraciones, ParadisEO vuelve a sobrepasar a ECJ.

Figura 42: *fitness* inicial promedio de Sphere

Initial Fitness: Framework Comparison (Server)

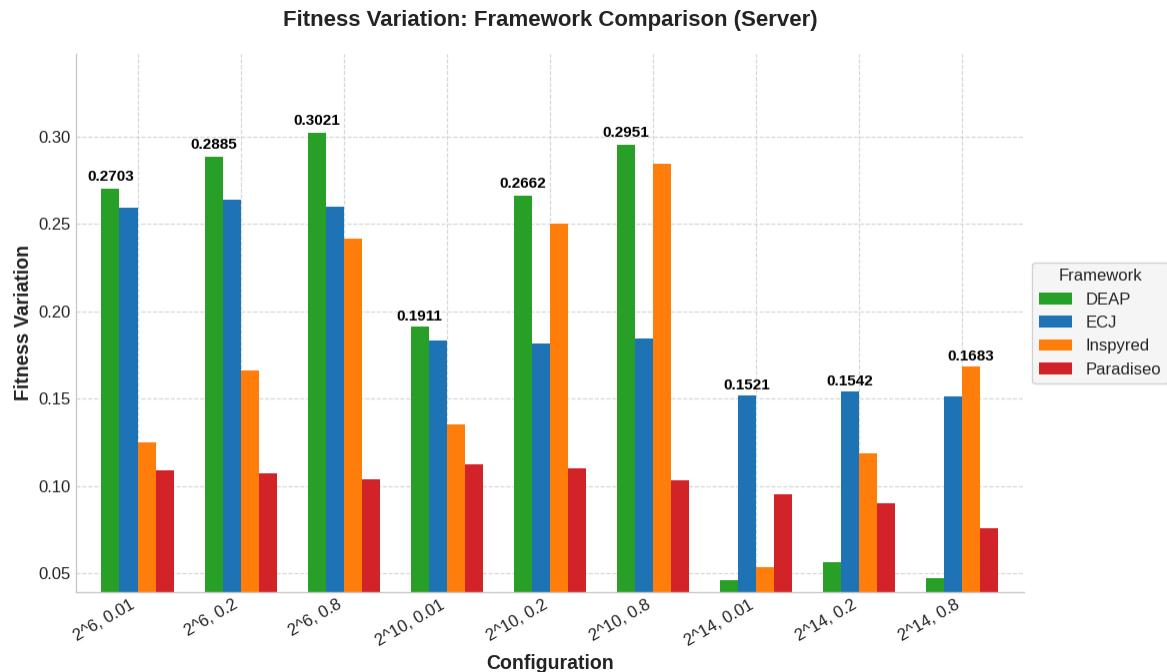


Fuente: elaboración propia

Esta reducción tan drástica confirma una “ley inversa” existente entre generaciones alcanzadas y tamaño de la población en la cual explican que para una condición de parada similar, los mecanismos de variación operan menos veces cuanto mayor sea el tamaño de la población.

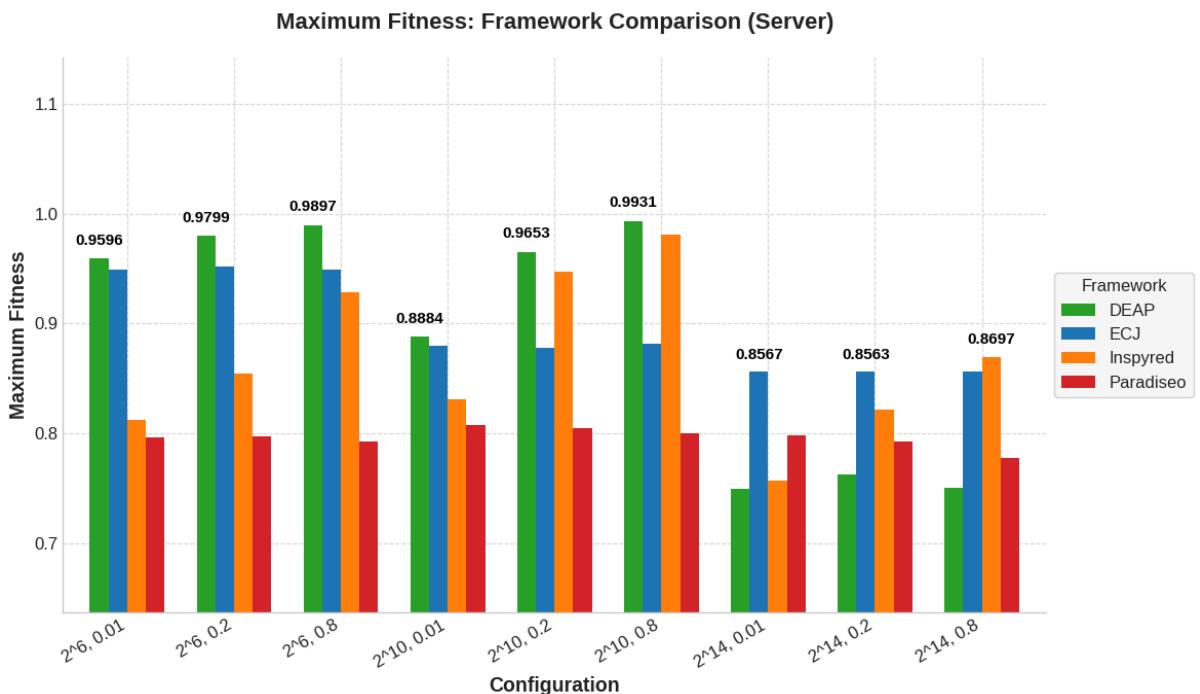
En segundo lugar, en la figura 42 se muestra que ECJ y DEAP parten con un *fitness* inicial promedio de 0,689 y 0,691 mientras que ParadisEO e Inspyred parten de un *fitness* inicial promedio un poco menor, rondando el 0,687.

Figura 43: variación promedio del *fitness* de Sphere



Fuente: elaboración propia

Figura 44: *fitness* máximo promedio alcanzado de Sphere



Fuente: elaboración propia

En tercer lugar, en la figura 43 se muestra la variación del *fitness*, donde es lógico pensar que, con un mayor porcentaje de probabilidad de cruce ( $p_c$ ), la variación tenderá a ser

mayor. Esto se puede observar en las tres configuraciones que tienen esa probabilidad, viendo que dentro del mismo tamaño de población, donde se alcanza mayor variación es cuando se tiene una  $p_c = 0,8$ . Así, se puede destacar que:

- DEAP sobresale cuando la población es de tamaño pequeño o medio, dado que su política de mutación explora rápido la diversidad inicial y alcanza la mayor subida absoluta, de casi 0,3 puntos. Sin embargo, cuando la población alcanza un  $N = 2^{14}$  la variación es prácticamente nula con un  $p_c = 0,01$ .
- ECJ mejora aunque se mantiene casi constante dentro de un mismo tamaño de población, independientemente de la probabilidad de cruce, reduciendo ligeramente la variación según aumenta el tamaño de la población.
- Se puede observar que ParadisEO mantiene una variación casi constante en cualquiera de las nueve configuraciones, teniendo un descenso, como era de esperar, cuando la población alcanza el  $N = 2^{14}$ .
- Inspyred sobresale cuando el tamaño de la población es mediano. Aunque no llega a ser el dominante, se aprecia claramente una tendencia de alto crecimiento dentro de un mismo tamaño de población, llegando a ser el mejor en  $N = 2^{14}, p_c = 0,8$ .

Por último, en la figura 44 se muestra la calidad de la solución alcanzada, que en general se maximiza con  $N = 2^{10}$  y  $p_c = 0,8$ :

- DEAP logra el *fitness* máximo más alto, 0,9931, muy próximo al óptimo teórico.
- Inspyred le sigue con 0,97 (en  $N = 2^6, p_c = 0,8$ ) y ECJ alcanza 0,86 en esa misma configuración.
- En  $N = 2^{14}$ , ECJ es el líder relativo con 0,8563, mientras que es seguido por Inspyred (e incluso superado en la última configuración) beneficiándose de la mayor variación residual en esa franja.

Tabla 5: Resumen de orden de *fitness* máximo alcanzado

<i>N</i>	<i>Orden</i>
$2^6$	DEAP > ECJ > Inspyred > ParadisEO
$2^{10}$	DEAP > Inspyred > ECJ > ParadisEO
$2^{14}$	ECJ > Inspyred > ParadisEO > DEAP

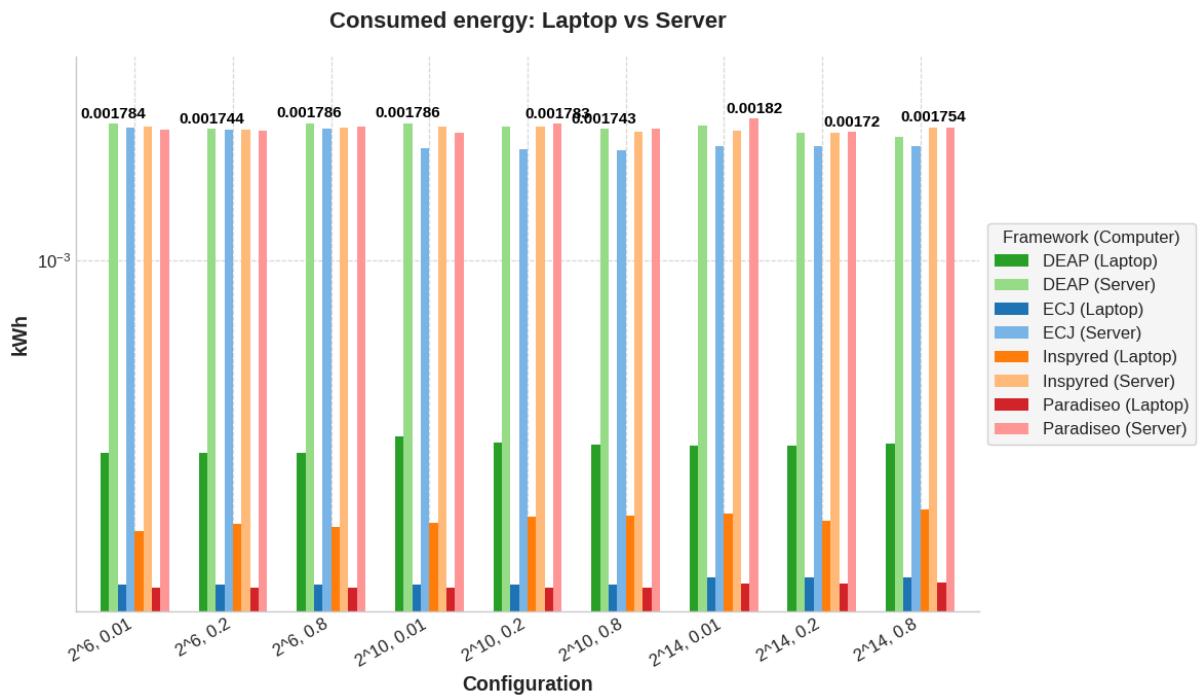
Fuente: elaboración propia

Si se hace una asignación de puntos para tratar de averiguar el mejor *framework* de manera que a la mejor posición obtenida en cada tamaño de población obtiene 1 punto y la peor obtiene 4, se puede observar que tanto DEAP como ECJ dominan al resto, seguidos de Inspyred y quedando como el peor ParadisEO.

#### 4.2.7. Consumo energético en el portátil frente al servidor

En la figura 45 se aprecia de entrada que, para todas las configuraciones y *frameworks*, el servidor consume sistemáticamente casi 4 veces más energía que el portátil. Por ejemplo, en DEAP, con  $N = 2^6$  y  $p_c = 0,01$  el portátil consume, aproximadamente, 0,0004453 kWh frente a los casi 0,00178 kWh del servidor, ratio que se mantiene muy similar para ECJ, Inspyred y ParadisEO. Esta diferencia del orden de la unidad de magnitud confirma el efecto de que, aunque los servidores ofrecen mayor potencia de cálculo, su coste energético absoluto es sensiblemente superior.

Figura 45: comparación de energía consumida de Sphere entre portátil y servidor



Fuente: elaboración propia

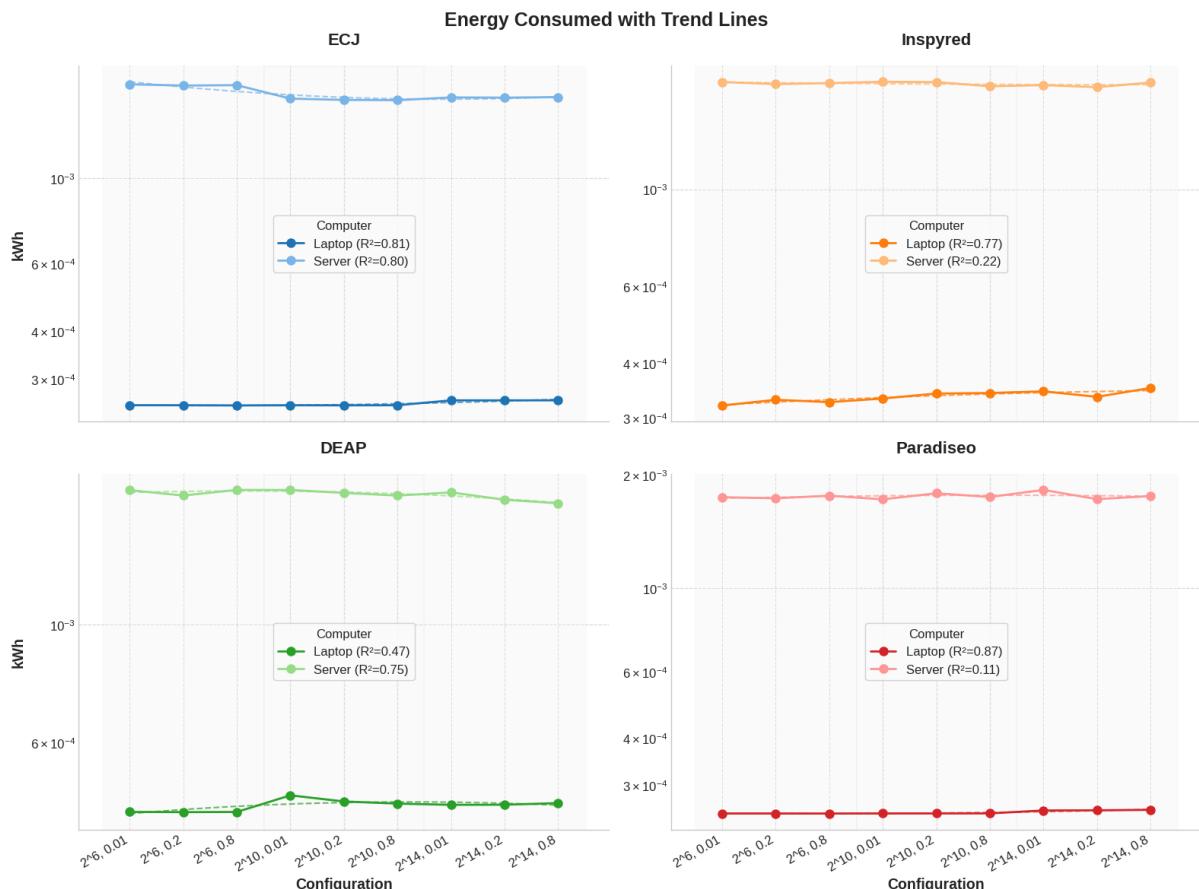
Al analizar la evolución del consumo en función del tamaño de población (de  $2^6$  a  $2^{14}$  individuos) se observan las siguientes tendencias:

- ECJ presenta en el portátil un ligero crecimiento del consumo medio por generación conforme crece N ( $R^2 = 0,81$ ), mientras que en el servidor muestra un comportamiento levemente decreciente ( $R^2 = 0,8$ ).
- Inspyred en el portátil reduce moderadamente su consumo al aumentar N ( $R^2 = 0,77$ ), pero en servidor la dependencia es casi nula ( $R^2 = 0,22$ ), indicando que su gasto energético se ajusta más al tiempo total de ejecución que al propio tamaño de la población.

- DEAP mantiene un patrón prácticamente constante en el portátil ( $R^2 = 0,47$ ), y exhibe un descenso moderado en el servidor ( $R^2 = 0,75$ ), probablemente por la sobrecarga de gestión de hilos y sincronización en entornos masivamente paralelos.
- ParadisEO muestra un consumo extraordinariamente estable en el portátil ( $R^2 = 0,87$ ) y sin apenas variación en el servidor ( $R^2 = 0,11$ ), lo cual puede ser valioso cuando se requiere predictibilidad energética.

Estos resultados corroboran la desigual eficiencia energética entre dispositivos de distintas prestaciones.

Figura 46: comparación desglosada de energía consumida de Sphere entre portátil y servidor



Fuente: elaboración propia

#### 4.2.8. Emisiones totales de CO<sub>2</sub> en el portátil frente al servidor

Fernández de Vega *et al.* (2016) señalan que el *Green Computing* surgió para abordar el consumo energético en la computación, particularmente en grandes instalaciones como centros de datos, principio que es aplicable al comparar diferentes arquitecturas en la ejecución de algoritmos evolutivos (p. 368).

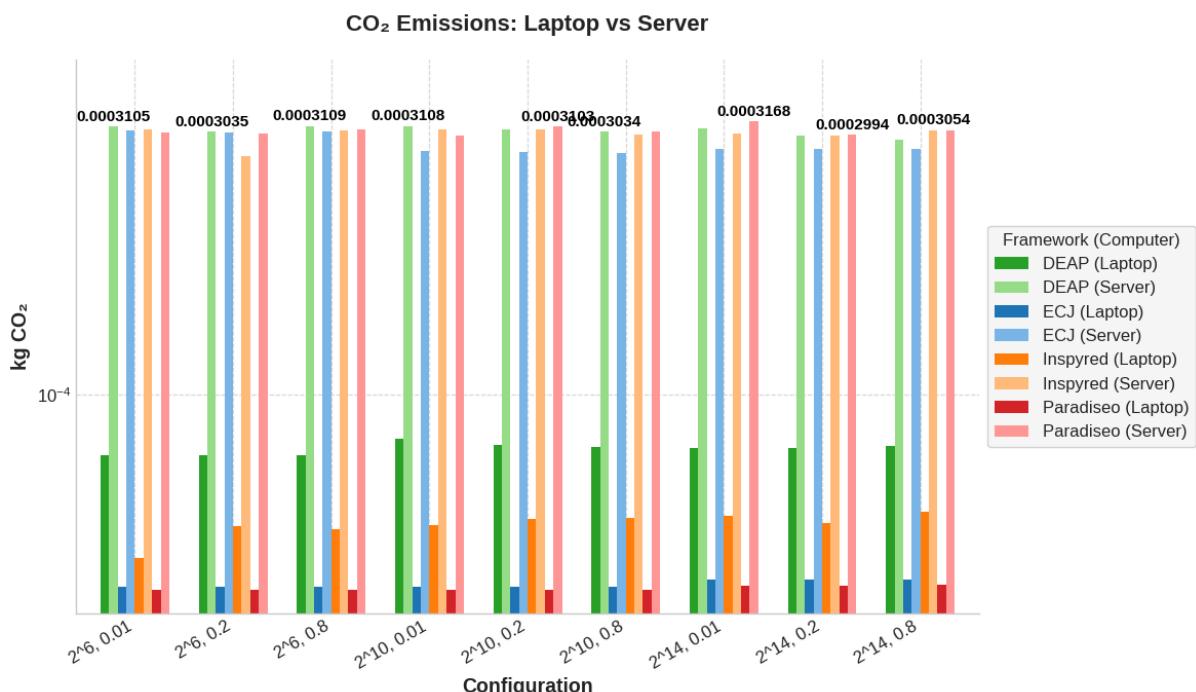
Los datos experimentales muestran que, para configuraciones idénticas de tamaño de población y probabilidad de cruce, el portátil emite entre 0,000041 kg CO<sub>2</sub> y 0,000058 kg CO<sub>2</sub>, mientras que el servidor emite entre 0,000303 kg CO<sub>2</sub> y 0,000317 kg CO<sub>2</sub>. En todos los

casos, la plataforma de servidor genera casi 10 veces más emisiones que el portátil para resolver Sphere bajo las mismas condiciones.

Al analizar la evolución de las emisiones con el tamaño de población (de  $2^6$  a  $2^{14}$  individuos) se aprecian estas tendencias:

- ECJ presenta en el portátil un ligero aumento de emisiones con el aumento del tamaño de la población ( $R^2 = 0,82$ ), mientras que en servidor sigue una pendiente similar tras un leve descenso al principio ( $R^2 = 0,8$ ).
- Inspyred aumenta moderadamente sus emisiones en el portátil al crecer el tamaño de la población ( $R^2 = 0,78$ ), pero en servidor la variación es casi nula tas una mínima fluctuación ( $R^2 = 0,07$ ), indicando que su huella de CO<sub>2</sub> se ajusta más al tiempo total de ejecución que al tamaño de la población.
- DEAP mantiene emisiones casi constantes en el portátil ( $R^2 = 0,47$ ) y muestra un descenso moderado en servidor ( $R^2 = 0,75$ ), posiblemente por la sobrecarga de gestión de hilos en entornos masivamente paralelos.
- ParadisEO es el más predecible: emisiones muy estables en portátil ( $R^2 = 0,88$ ) y sin apenas tendencia en servidor ( $R^2 = 0,11$ ), factor útil cuando se requiere control estricto de la huella de carbono.

Figura 47: comparación de emisiones de CO<sub>2</sub> de Sphere entre portátil y servidor

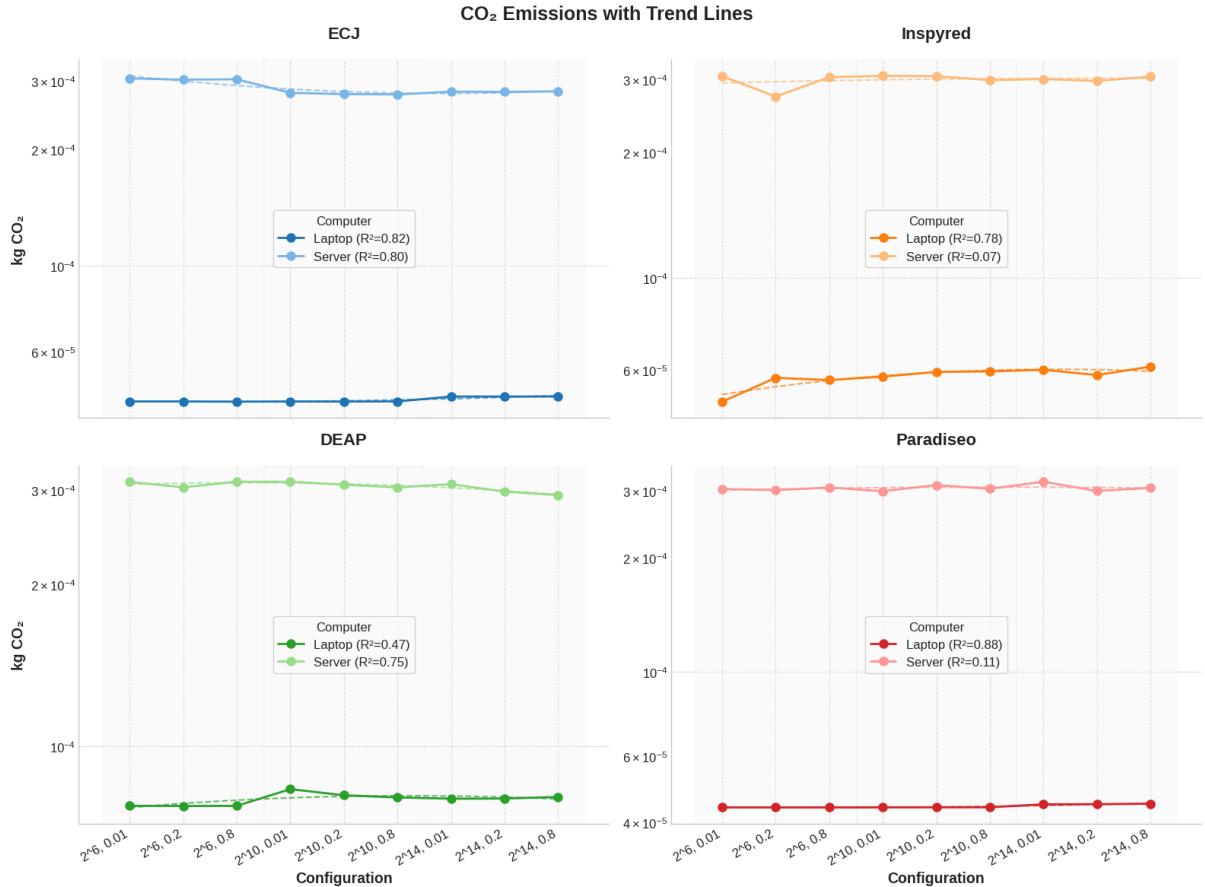


Fuente: elaboración propia

Estos resultados confirman la disparidad de emisiones entre dispositivos portátiles y servidores: a pesar de su velocidad de cómputo, los servidores generan un coste de CO<sub>2</sub> sustancialmente mayor que los sistemas portátiles bajo cargas comparables. En conclusión, el

análisis de las gráficas de emisiones totales de CO<sub>2</sub> demuestra que el portátil ofrece una ventaja clara en eficiencia y huella de carbono frente al servidor.

Figura 48: comparación desglosada de emisiones de CO<sub>2</sub> de Sphere entre portátil y servidor



Fuente: elaboración propia

#### 4.2.9. Análisis la evolución del *fitness* en el portátil frente al servidor

Ambas máquinas comparten el mismo proceso de inicialización aleatoria, por lo que los valores promedio de *fitness* al arranque se sitúan cerca de 0,69 para DEAP y ECJ en todas las configuraciones, sin diferencias significativas entre portátil y servidor. Sin embargo, en Inspyred y ParadisEO el portátil muestra un inicio ligeramente superior al servidor en poblaciones grandes (2<sup>14</sup>).

La variación de *fitness*, calculada como la diferencia entre *fitness* máximo e inicial, muestra patrones parecidos en ambas plataformas:

- Inspyred obtiene la mayor ganancia en poblaciones intermedias ( $N = 2^{10}$ ,  $p_c = 0,8$ ): en torno a 0,28 y 0,30 puntos en el portátil y aproximadamente 0,29 en el servidor.
- DEAP se sitúa en torno a 0,26 y 0,28 en el portátil para poblaciones de tamaño  $N = 2^6$  y  $N = 2^{10}$ , ascendiendo hasta casi 0,30 en el servidor con  $N = 2^{10}$ , y cayendo a valores residuales en  $N = 2^{14}$ .

- ParadisEO mantiene variaciones moderadas, situándose alrededor de 0,11 puntos tanto en el portátil como en el servidor, con un ligero pico en  $N = 2^{10}$ .
- ECJ registra la ganancia más estable en ambas máquinas, alcanzando los 0,18 puntos en  $N = 2^{10}$  y casi 0,15 puntos en  $N = 2^{14}$ .

En términos generales, la curva de ganancia se superpone casi idéntica entre portátil y servidor, indicando que el paralelismo del servidor no altera la dinámica de búsqueda, sino únicamente la velocidad de ejecución.

El *fitness* perfecto ( $fitness = 1$ ) no se llega a lograr, y los máximos reales dependen fuertemente de la población y del *framework*:

- Inspyred lidera en  $N = 2^{10}$ ,  $p_c = 0,8$ , con casi 0,974 puntos en el portátil y 0,980 en el servidor.
- DEAP exhibe valores desde 0,87 (portátil,  $N = 2^{10}$ ) hasta 0,993 (servidor,  $N = 2^{10}$ ), mientras que en poblaciones grandes cae alrededor de los 0,72 y 0,75 puntos en ambas máquinas.
- ParadisEO alcanza los 0,79 puntos (portátil,  $N = 2^{10}$ ) y los 0,81 puntos (servidor,  $N = 2^{10}$ ).
- ECJ se mantiene estable sobre los 0,88 y 0,96 puntos en el portátil y sobre los 0,85 y 0,95 puntos en el servidor.

Aunque las diferencias absolutas son pequeñas entre plataformas, la portátil tiende a obtener ligeramente mejores óptimos en configuraciones medias ( $2^{10}$ ), quizás por una convergencia más estable ante recursos limitados.

Por último, el recuento de generaciones completadas antes de detenerse revela una clara ventaja de cálculo bruto del servidor, visible en la tabla 6:

- Para poblaciones pequeñas ( $2^6$ ), el servidor completa, aproximadamente, 330.000 generaciones, frente a las 126.000 en el portátil, lo que supone casi 2,62 veces más.
- Con poblaciones intermedias ( $2^{10}$ ), esa relación casi se mantiene (6.000 vs. 2.900), suponiendo el doble de generaciones alcanzadas en el servidor frente al portátil.
- En poblaciones grandes ( $2^{14}$ ), ambas máquinas requieren muchas menos generaciones (unas 170 en el portátil, y casi 300 en el servidor), reduciéndose nuevamente la proporción, siendo en esta ocasión de 1,76 veces más generaciones ejecutadas.

Este patrón uniforme indica que el servidor acelera el ritmo de iteración en un factor cercano a cuatro, sin alterar el número de generaciones necesario para que el algoritmo alcance al criterio de parada.

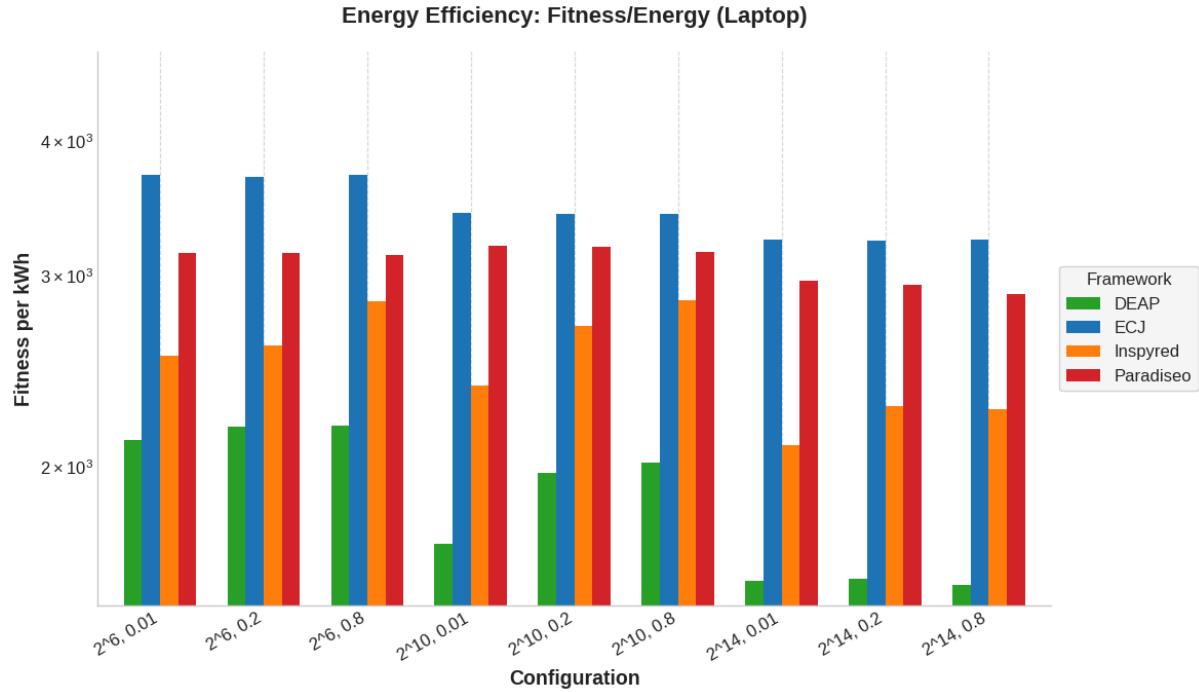
Tabla 6: Resumen aproximado de las generaciones alcanzadas de Sphere

$N y p_c$	<i>portátil</i>	<i>servidor</i>	Razón <i>servidor/portátil</i>
$2^6, 0,01$	126.100	341.000	2,7
$2^6, 0,2$	128.400	331.300	2,58
$2^6, 0,8$	126.300	333.200	2,64
$2^{10}, 0,01$	2.899	6.189	2,13
$2^{10}, 0,2$	2.898	5.179	1,79
$2^{10}, 0,8$	2.883	4.677	1,62
$2^{14}, 0,01$	171	365	2,13
$2^{14}, 0,2$	172	313	1,81
$2^{14}, 0,8$	171	291	1,7

Fuente: elaboración propia

#### 4.2.10. Análisis de la eficiencia energética en el portátil

Figura 49: comparación entre *frameworks* del cálculo de  $\eta$  de Sphere en el portátil



Fuente: elaboración propia

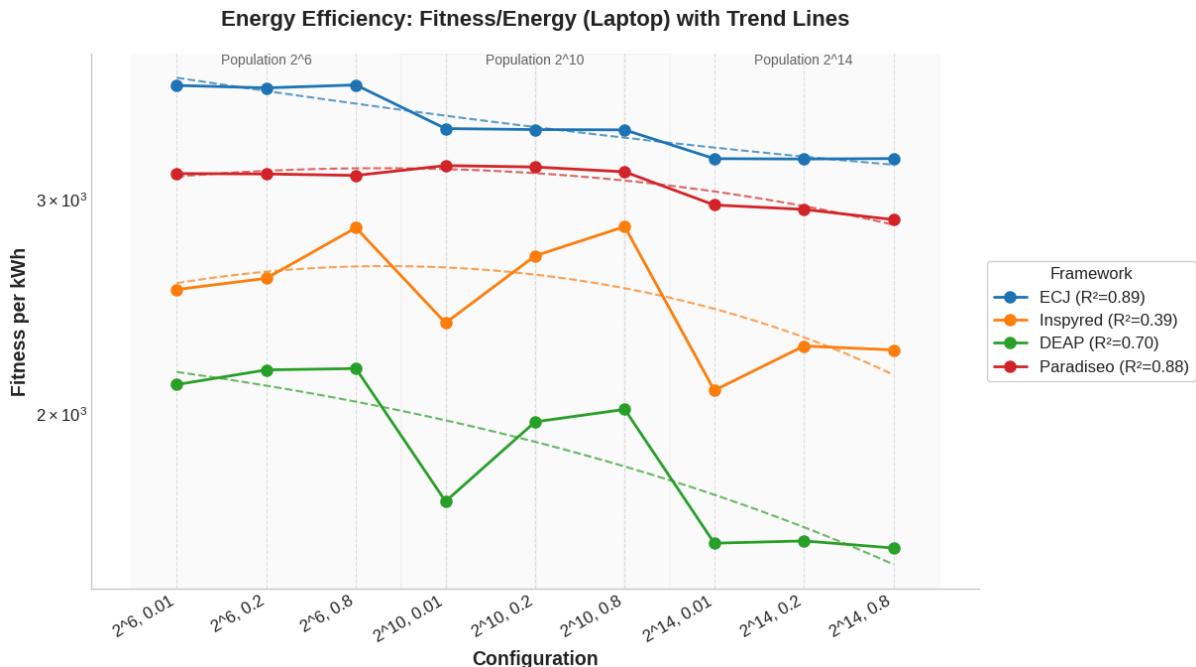
Para evaluar la eficiencia energética de cada *framework* en el portátil, se define la métrica

$$\eta = \frac{\text{Fitness máximo alcanzado}}{\text{Consumo (kWh)}}$$

que indica cuántas unidades de *fitness* se obtienen por cada kWh consumido. Así, gracias a la figura 49, se puede observar que:

- ECJ es el más eficiente en todas las configuraciones, con valores de  $\eta$  entre 3.100 y 3.700 *fitness/kWh*. Esto revela que, pese a su estrategia conservadora de variación de *fitness*, ECJ convierte cada vatio consumido en alta ganancia de solución.
- ParadisEO ocupa la segunda posición, con  $\eta$  oscilando entre 2.950 y 3.230 *fitness/kWh*. Su implementación, ligera y predecible, aprovecha muy bien la energía disponible.
- Inspyred muestra una eficiencia intermedia, con  $\eta$  comprendido entre 2.300 y 2.850 *fitness/kWh*, destacando en  $N = 2^6$ ,  $p_c = 0,8$  con casi 2.850 *fitness/kWh*, pero perdiendo rentabilidad en poblaciones más grandes ( $N = 2^{14}$ ).
- DEAP es el menos eficiente energéticamente, con  $\eta$  en torno a 1.300 y 2.200 *fitness/kWh*, alcanzando su pico en  $N = 2^6$  con casi 2.200 *fitness/kWh* y cayendo drásticamente en  $N = 2^{14}$  por el mayor coste por generación.

Figura 50: comparación entre tendencias de *frameworks* del cálculo de  $\eta$  de Sphere en el portátil



Fuente: elaboración propia

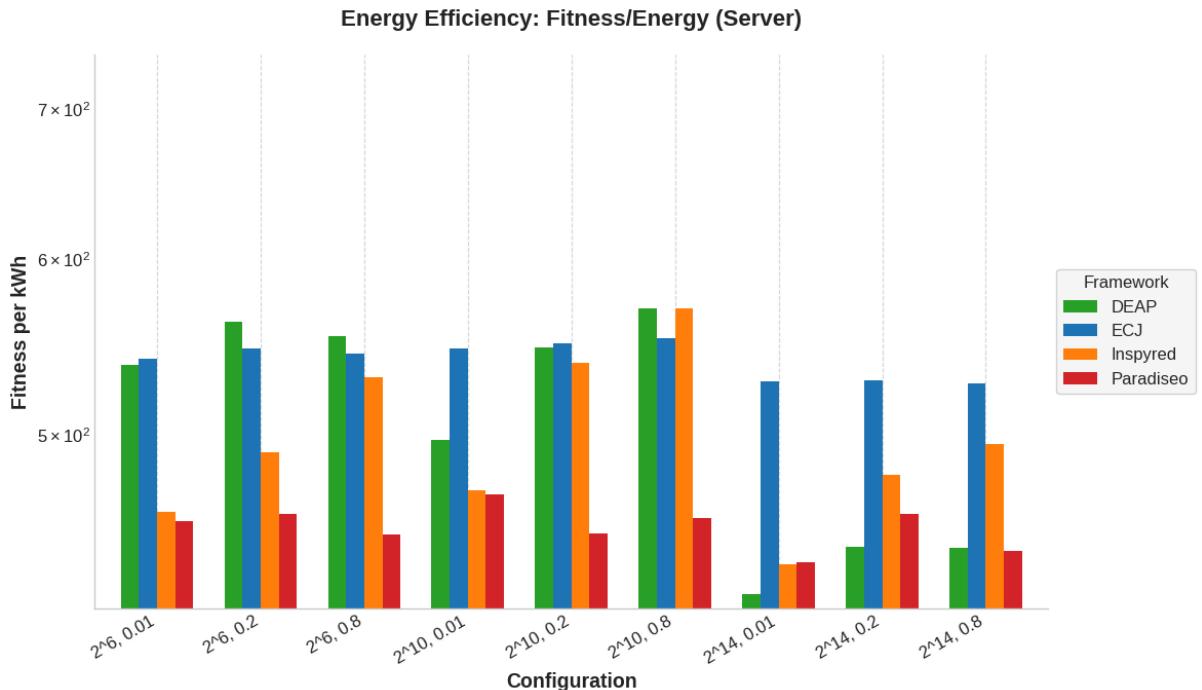
Las líneas de tendencia polinomiales revelan comportamientos característicos para cada *framework*:

- ECJ exhibe una tendencia ligeramente decreciente ( $R^2 = 0,89$ ), indicando que su eficiencia cae de modo casi lineal al crecer la población.
- ParadisEO denota una curva casi plana ( $R^2 = 0,88$ ), lo que sugiere que su eficiencia energética se mantiene estable independientemente del tamaño de población.

- Inspyred presenta un pico en  $N = 2^{10}$ ,  $p_c = 0,8$  seguido de un descenso pronunciado en  $N = 2^{14}$  ( $R^2 = 0,39$ ), reflejo de su alta sensibilidad al balance exploración-explotación.
- DEAP muestra una concavidad suave ( $R^2 = 0,7$ ), con eficiencia creciente hasta  $N = 2^{10}$  y posterior caída al llegar a  $N = 2^{14}$ , confirmando un óptimo intermedio de rentabilidad energética.

#### 4.2.11. Análisis de la eficiencia energética en el servidor

Figura 51: comparación entre *frameworks* del cálculo de  $\eta$  de Sphere en el servidor



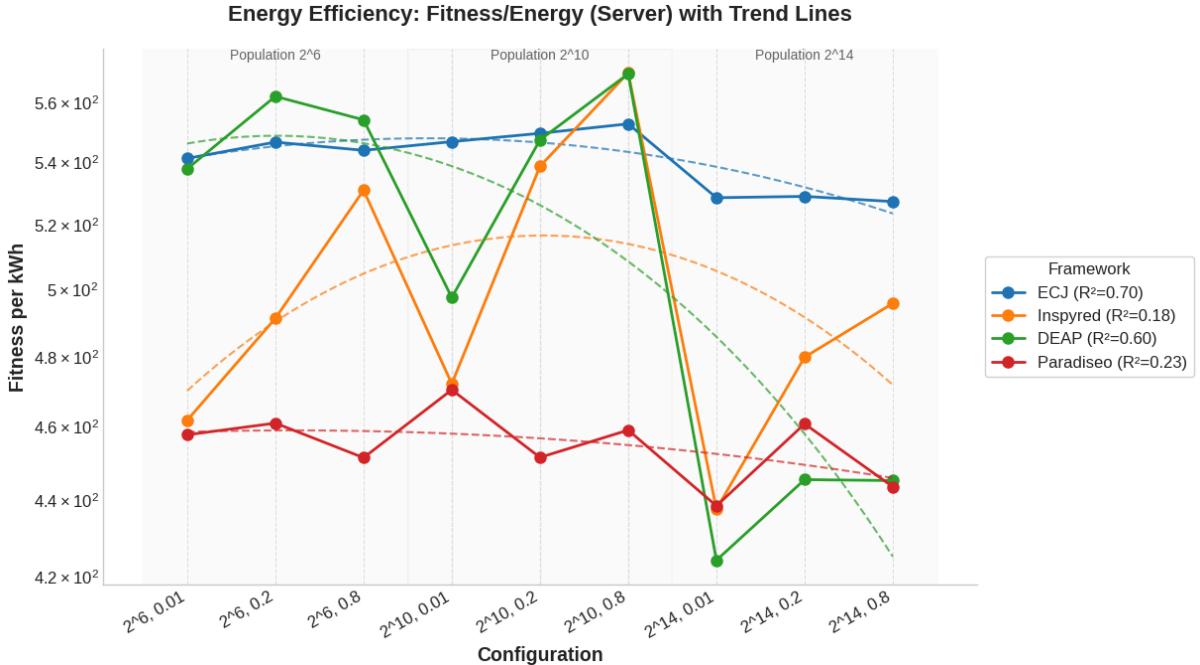
Fuente: elaboración propia

Usando la misma métrica que en el apartado anterior, se puede observar en las figuras 51 y 52 que:

- Inspyred es el *framework* más eficiente, alcanzando un pico de casi 570 *fitness/kWh* en la configuración de población intermedia ( $N = 2^{10}$ ,  $p_c = 0,8$ ).
- DEAP le sigue muy de cerca, con un máximo de casi 560 *fitness/kWh* en  $N = 2^{10}$ ,  $p_c = 0,2$  y en torno a 570 en  $N = 2^{10}$ ,  $p_c = 0,8$ .
- ECJ ocupa la tercera posición, con  $\eta$  entre 530 y 550 *fitness/kWh*, y una clara tendencia decreciente al crecer N ( $R^2 = 0,7$ ), lo que indica que su consumo escala ligeramente más rápido que su ganancia de *fitness*.
- ParadisEO es el menos eficiente, con valores de  $\eta$  constantes que oscilan entre 440 y 470 *fitness/kWh* ( $R^2 = 0,23$ ), reflejando un perfil estable pero de menor rendimiento por unidad de energía.

La tendencia polinómica para Inspyred muestra un abrupto incremento de eficiencia al pasar de  $N = 2^6$  a  $N = 2^{10}$  y una suave caída en  $N = 2^{14}$ , evidenciando un óptimo poblacional claro. En contraste, ECJ presenta un decrecimiento casi lineal de  $\eta$  con el tamaño de la población.

Figura 52: comparación entre tendencias de *frameworks* del cálculo de  $\eta$  de Sphere en el servidor



Fuente: elaboración propia

#### 4.2.12. Análisis de la eficiencia energética del portátil frente al servidor

A continuación, se comparan los resultados obtenidos en el portátil y en el servidor usando la misma métrica que la expuesta en el punto 4.2.10. Como se puede observar en las figuras 53 y 54, en todas las configuraciones y *frameworks*, el portátil multiplica por cuatro o, incluso, casi por siete la eficiencia de la misma combinación en el servidor. Para población  $N = 2^{10}$  y  $p_c = 0,8$ :

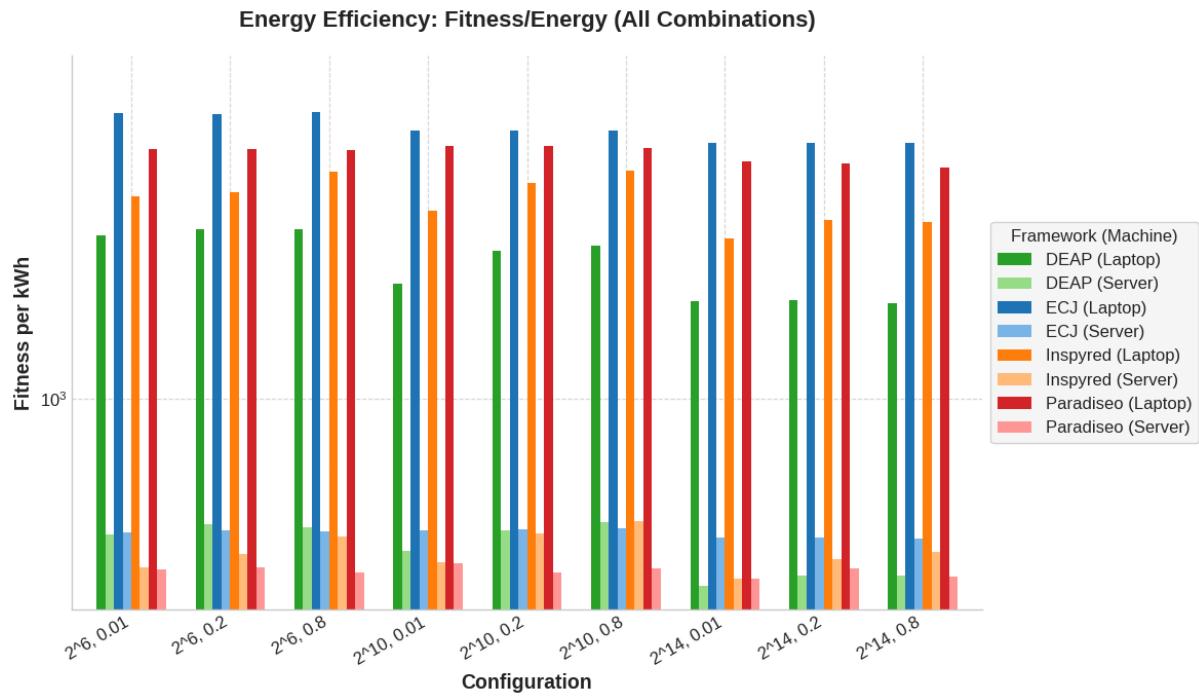
Tabla 7: Ejemplo de comparación de  $\eta$  de Sphere para  $N = 2^{10}$  y  $p_c = 0,8$  en portátil y servidor

Framework	$\eta$ (portátil)	$\eta$ (servidor)	Factor multiplicativo
DEAP	2.200	570	3,85
ParadisEO	3.230	470	6,87
Inspyred	2.850	570	5
ECJ	3.700	550	6,73

Fuente: elaboración propia

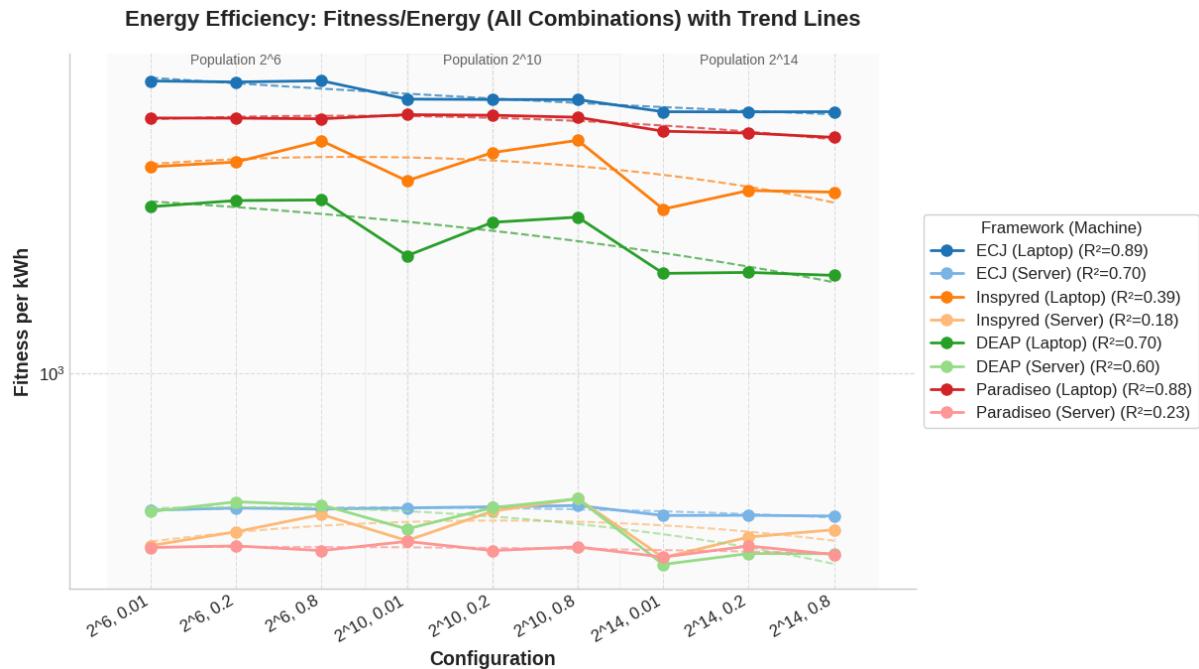
Este salto refleja cómo la sobrecarga de gestión de paralelismo masivo en el servidor penaliza drásticamente la rentabilidad energética, pese a su mayor número de iteraciones por unidad de tiempo.

Figura 53: comparación entre *frameworks* del cálculo de  $\eta$  de Sphere en portátil y servidor



Fuente: elaboración propia

Figura 54: comparación entre tendencias de *frameworks* del cálculo de  $\eta$  de Sphere en portátil y servidor



Fuente: elaboración propia

Las líneas de tendencia polinomiales indican que todas las curvas sitúan su máximo  $\eta$  en poblaciones  $N = 2^6$ .

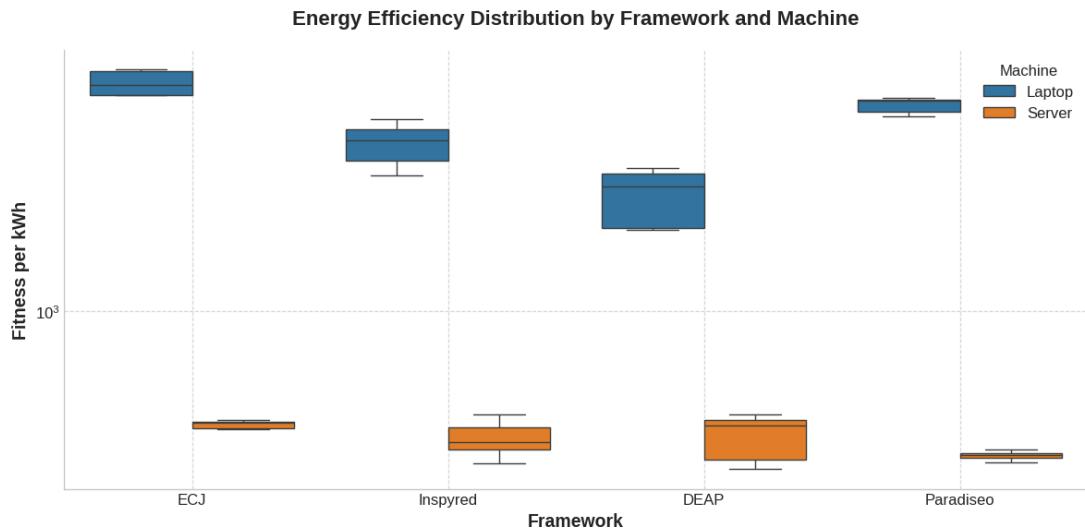
En conclusión, aunque el servidor acelera notablemente el cómputo, ese aumento de velocidad no compensa el sobrecoste energético por generación. El portátil, combinado con frameworks bien optimizados (especialmente ECJ y ParadisEO), ofrece la mejor eficiencia *fitness/kWh*, subrayando la necesidad de incluir métricas energéticas como criterio decisivo al seleccionar arquitecturas y algoritmos evolutivos para problemas de optimización.

#### 4.2.13. Distribución de eficiencia energética y de variación de *fitness*

Para completar el análisis de eficiencia, se examina la distribución de las métricas de *fitness* por kWh (figura 55) y de variación de *fitness* por kWh (figura 56). Cabe así destacar, por framework:

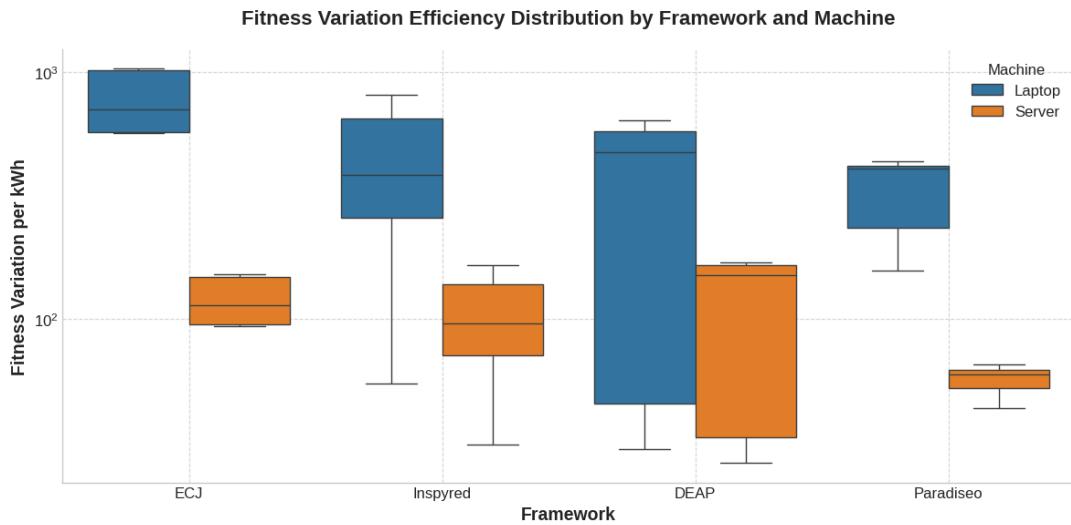
- ECJ: en el portátil, presenta una caja muy estrecha comprendida entre 3.550 y 3.650 *fitness/kWh* con *whiskers* de apenas  $\pm 50$ , y su variación de *fitness* por kWh se concentra entre 800 y 1.000 *fitness/kWh*. En el servidor, la mediana de eficiencia baja a casi 540 *fitness/kWh*, con *whiskers* de  $\pm 20$ , mientras que la variación/kWh ronda los 90 y 120 *fitness/kWh*, también con dispersión mínima. Este bajo rango en ambas métricas refleja una implementación extremadamente estable y poco susceptible a los cambios de tamaño de población o de probabilidad de cruce.
- Inspyred: En el portátil, muestra la caja más ancha en eficiencia energética, entre 2.550 y 2.850 *fitness/kWh*, y su variación/kWh oscila ampliamente de 250 a 650 variación/kWh. En el servidor, su eficiencia varía entre 450 y 530 *fitness/kWh* (*whiskers* 350/580) y la variación/kWh entre 80 y 140, confirmando que alcanza picos en poblaciones intermedias pero sufre caídas pronunciadas en extremos.
- DEAP: en el portátil, la eficiencia se distribuye de forma moderada entre 2.100 y 2.300 *fitness/kWh* (*whiskers* 1.500/2.350), con variación/kWh situada en torno a 150 y 500. En el servidor, la eficiencia abarca los 450 y 550 *fitness/kWh* (*whiskers* 320/580) y la variación/kWh se sitúa alrededor de 50 y 200, mostrando un buen equilibrio entre consistencia y adaptabilidad al paralelismo.
- Paradiseo: en el portátil es también muy predecible, con caja estrecha de 3.000 a 3.200 *fitness/kWh* y *whiskers* de  $\pm 50$ , y variación/kWh concentrada entorno a 250 y 350. En el servidor, su eficiencia varía poco, entre 440 y 460 *fitness/kWh*, y la variación/kWh entre 60 y 90, siendo la segunda implementación más homogénea.

Figura 55: distribución de la eficiencia energética de Sphere en portátil y servidor



Fuente: elaboración propia

Figura 56: distribución de la variación del *fitness* de Sphere en portátil y servidor



Fuente: elaboración propia

### 4.3. Rosenbrock

En esta subsección se repetirá la misma estructura de análisis del experimento Sphere, pero aplicándolo al *benchmark* Rosenbrock.

#### 4.3.1. Consumo energético en el portátil

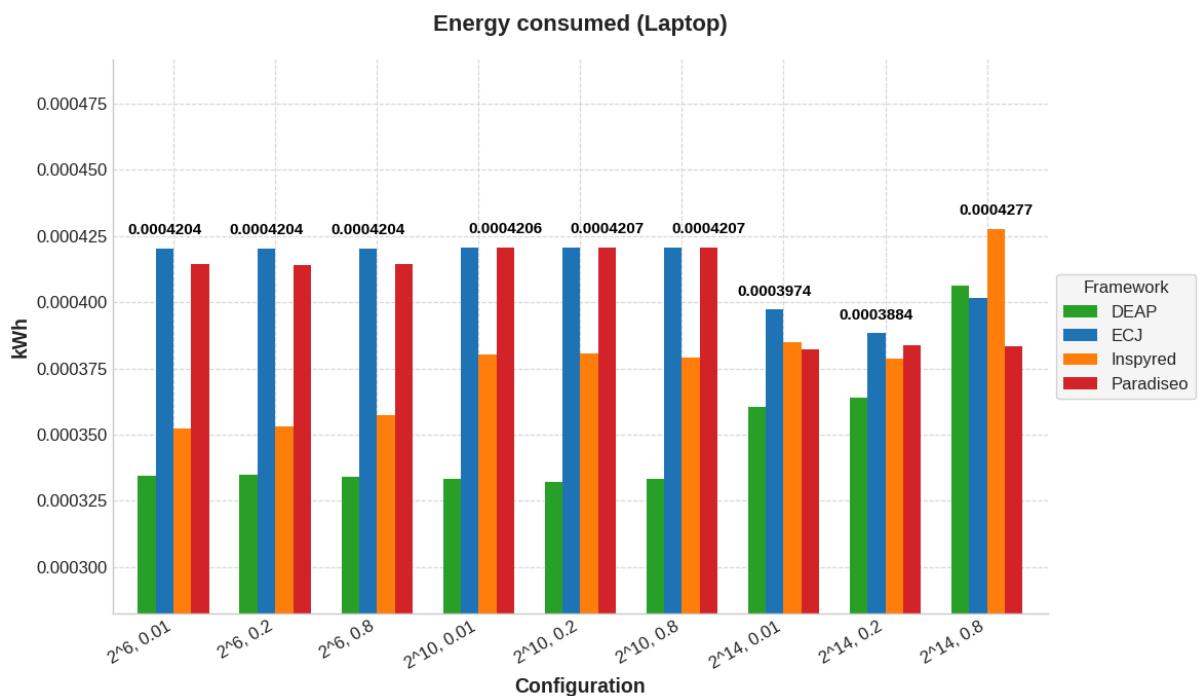
Como se puede observar en la figura 57, la energía consumida entre los *frameworks* difiere de forma significativa en el portátil al optimizar Rosenbrock:

- DEAP oscila, aproximadamente, entre  $0,000333 \text{ kWh}$  ( $N = 2^6$ ,  $p_c = 0,8$ ) y  $0,000407 \text{ kWh}$  ( $N = 2^{14}$ ,  $p_c = 0,8$ ), situándose como el más eficiente en términos de consumo absoluto.

- Inspyred se mueve de 0,000353 kWh ( $N = 2^6$ ,  $p_c = 0,01$ ) a 0,000427 kWh ( $N = 2^{14}$ ,  $p_c = 0,8$ ), ocupando el segundo puesto.
- ParadisEO registra valores bastante estables hasta  $N = 2^{10}$  entre 0,000413 y 0,000421 kWh, con un descenso en poblaciones grandes hasta casi los 0,00038 kWh.
- ECJ presenta el consumo más alto y uniforme, rondando los 0,00042 kWh en casi todas las configuraciones, reduciéndose al llegar a  $N = 2^{14}$ .

Esto revela que DEAP es el *framework* más “ligero” energéticamente, mientras ECJ presenta casi un 23 % más de gasto por ejecución que DEAP.

Figura 57: energía consumida por el portátil en las ejecuciones de Rosenbrock

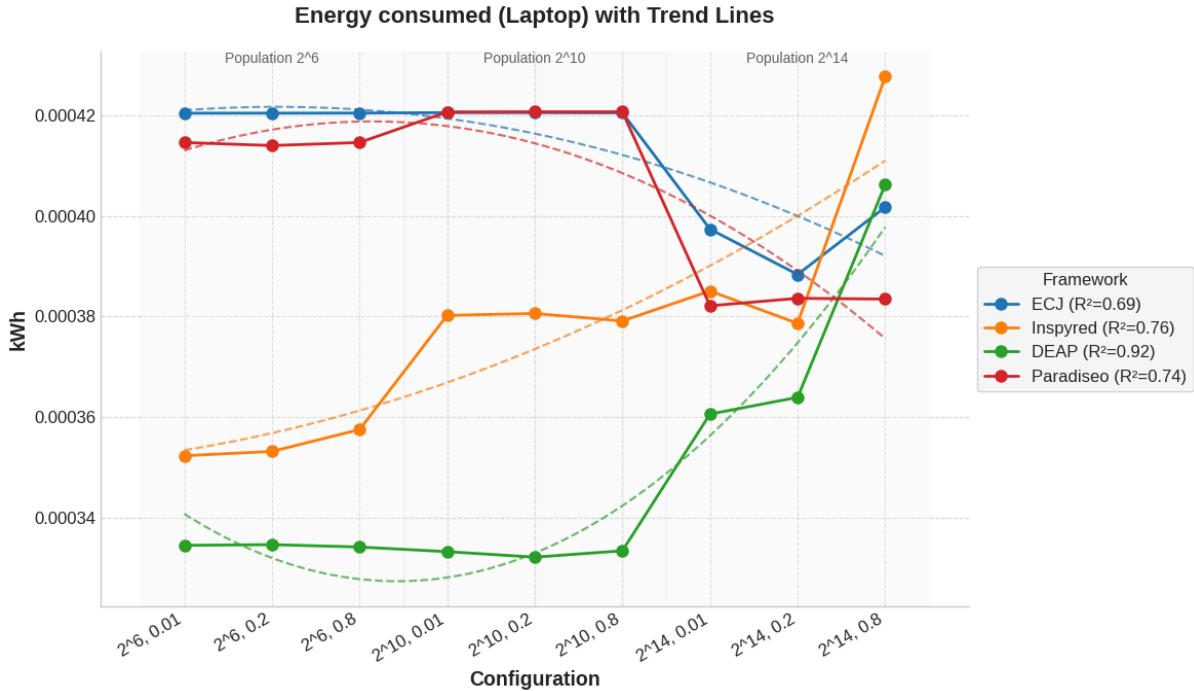


Fuente: elaboración propia

Al atender a las líneas de tendencia polinómicas visibles en la figura 58:

- ECJ y ParadisEO registran un escalado casi lineal y suave ( $R^2 = 0,69$  y  $R^2 = 0,74$ , respectivamente), lo que sugiere variaciones en el consumo energético proporcionales al tamaño de población, atribuibles a la utilización de la jerarquía de memoria (Fernández de Vega *et al.*, 2016, p. 374).
- Inspyred muestra una pendiente positiva notable ( $R^2 = 0,76$ ), con consumo creciente conforme la población crece, lo que indica mayor coste de gestión al manejar poblaciones más grandes.
- DEAP exhibe una curvatura convexa ( $R^2 = 0,92$ ), con un mínimo de consumo en  $N = 2^{10}$  y un repunte en  $N = 2^{14}$ , posiblemente debido a la amortización de costes de arranque frente al tamaño poblacional.

Figura 58: tendencia de la energía consumida por el portátil en las ejecuciones de Rosenbrock



Fuente: elaboración propia

Para poner cifras en contexto:

- Poblaciones pequeñas ( $N = 2^6$ ): consumo aproximado de 0,000334 kWh (DEAP), 0,000353 kWh (Inspyred), 0,000420 kWh (ECJ) y 0,000415 kWh (ParadisEO).
- Poblaciones intermedias ( $N = 2^{10}$ ): DEAP reduce su consumo un 0,3 %, Inspyred aumenta un 7,6 %, ECJ apenas varía (+0,1 %) y ParadisEO sube un 1,9 %.
- Poblaciones grandes ( $N = 2^{14}$ ): DEAP incrementa su consumo un 8,1 % respecto a  $N = 2^{10}$ , Inspyred lo hace un 1,1 %, mientras ECJ y ParadisEO reducen su gasto un 5,7 % y 8,9 %, respectivamente.

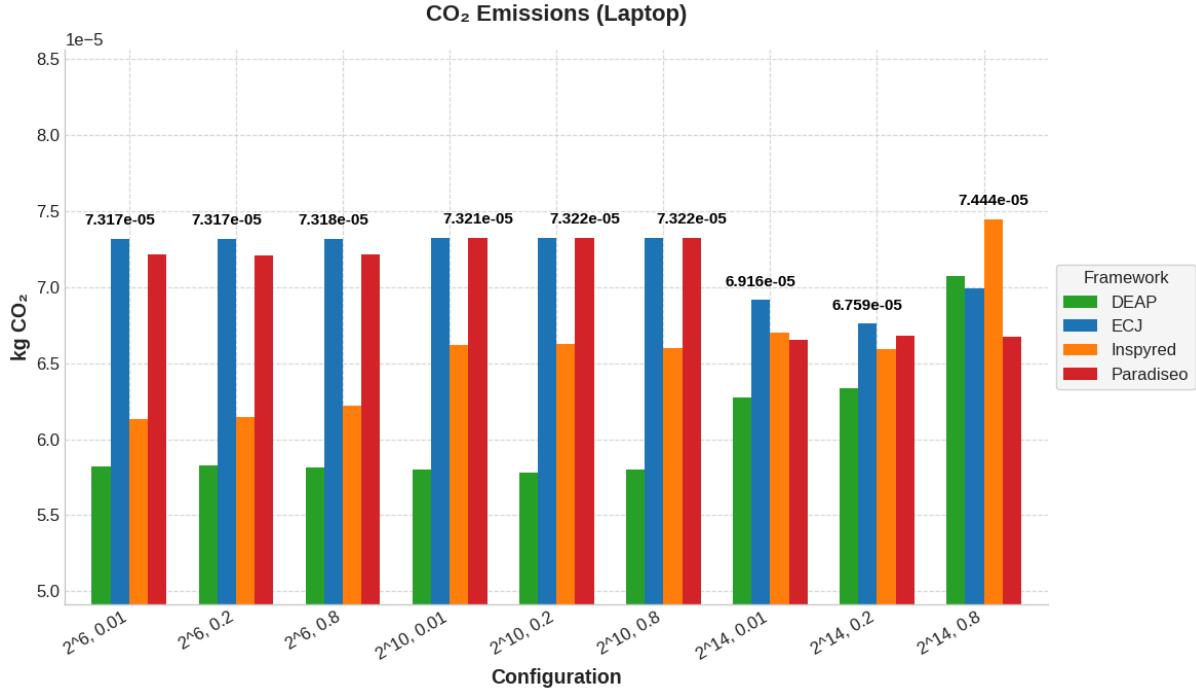
En conjunto, el portátil muestra un comportamiento muy cercano al paradigma de la computación energéticamente proporcional para ECJ y ParadisEO, mientras que Inspyred y DEAP muestran mayores sobrecostes al aumentar la población, con DEAP anotándose el menor consumo absoluto en configuraciones medias y altas.

#### 4.3.2. Emisiones totales de CO<sub>2</sub> en el portátil

Según se puede observar en la figura 59, las emisiones de CO<sub>2</sub> reproducen fielmente el patrón de consumo energético visto en la sección anterior.

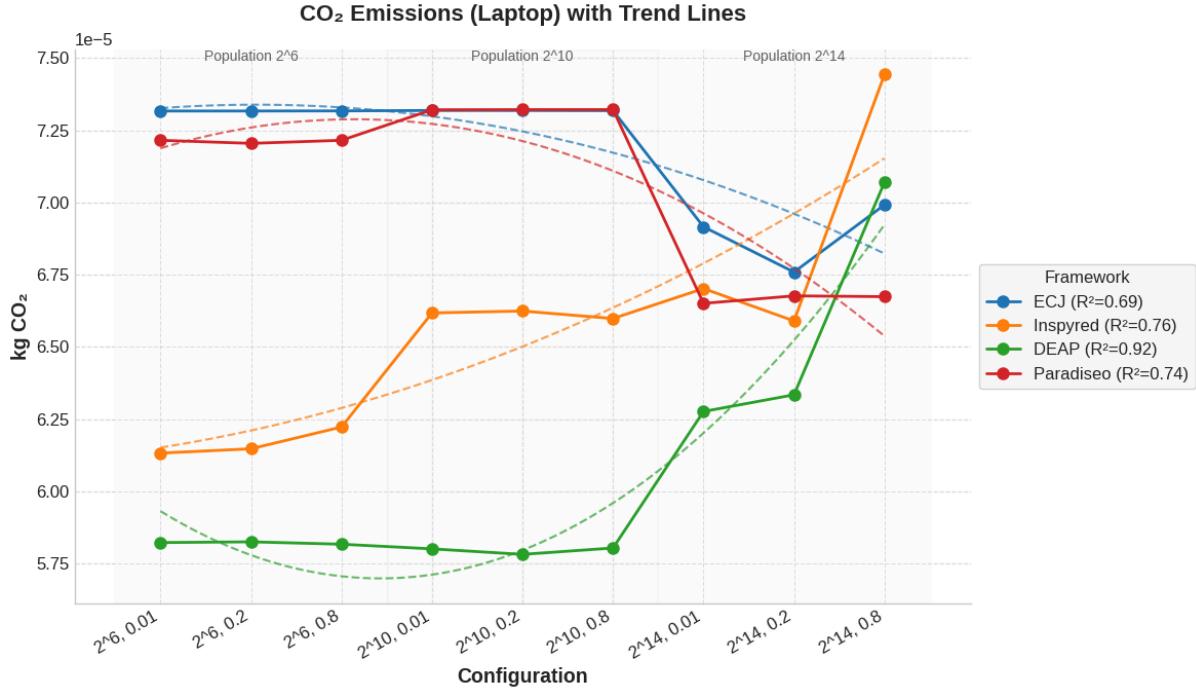
En primer lugar, DEAP presenta las emisiones más bajas de todos los *frameworks*, oscilando de 0,0000582 kg CO<sub>2</sub> ( $N = 2^6$ ,  $p_c = 0,2$ ) a 0,0000708 kg CO<sub>2</sub> ( $N = 2^{14}$ ,  $p_c = 0,8$ ). A medida que la población crece, DEAP mantiene un consumo de emisiones relativamente plano hasta  $N = 2^{10}$  y experimenta un repunte al pasar a  $N = 2^{14}$ .

Figura 59: emisiones totales de CO<sub>2</sub> por el portátil en las ejecuciones de Rosenbrock



Fuente: elaboración propia

Figura 60: tendencia de las emisiones totales de CO<sub>2</sub> por el portátil en las ejecuciones de Rosenbrock



Fuente: elaboración propia

En segundo lugar, Inspyred ocupa la segunda posición en el *framework* con la huella de carbono más baja, con emisiones que van de 0,0000613 kg CO<sub>2</sub> ( $N = 2^6$ ,  $p_c = 0,01$ ) a 0,0000744 kg CO<sub>2</sub> ( $N = 2^{14}$ ,  $p_c = 0,8$ ). Su tendencia positiva ( $R^2 = 0,76$ ) muestra un incremento sostenido de emisiones conforme la población crece, con un ligero pico en la

primera configuración de  $N = 2^{14}$ ,  $p_c = 0,01$ , el cual sugiere una combinación no lineal de accesos a memoria y gestión de intérprete Python.

En tercer lugar, ECJ, que, a pesar de su elevado consumo energético, mantiene emisiones relativamente constantes en torno a 0,0000732 kg CO<sub>2</sub> para  $N = 2^6$  y  $N = 2^{10}$ , descendiendo hasta 0,0000676 kg CO<sub>2</sub> en  $N = 2^{14}$ . Su ajuste polinómico ( $R^2 = 0,69$ ) revela un suave decrecimiento de emisiones al aumentar N, lo que indica un comportamiento de incremento energético con el aumento de población, posiblemente relacionado con la gestión de memoria del sistema (Fernández de Vega *et al.*, 2016, pp. 554-555).

Por último, ParadisEO registra las emisiones más elevadas en poblaciones intermedias, con aproximadamente 0,0000733 kg CO<sub>2</sub> en  $N = 2^{10}$  y desciende ligeramente hasta 0,0000668 kg CO<sub>2</sub> en  $N = 2^{14}$ . Su curva de tendencia ( $R^2 = 0,74$ ) muestra baja variabilidad entre configuraciones y confirmando su predictibilidad en huella de carbono.

En conjunto, estos resultados subrayan que DEAP e Inspyred son las opciones más “verdes” para Rosenbrock en el portátil, mientras que ECJ y ParadisEO, aunque muy estables, presentan emisiones sensiblemente más altas.

#### **4.3.3. Evolución del *fitness* en el portátil**

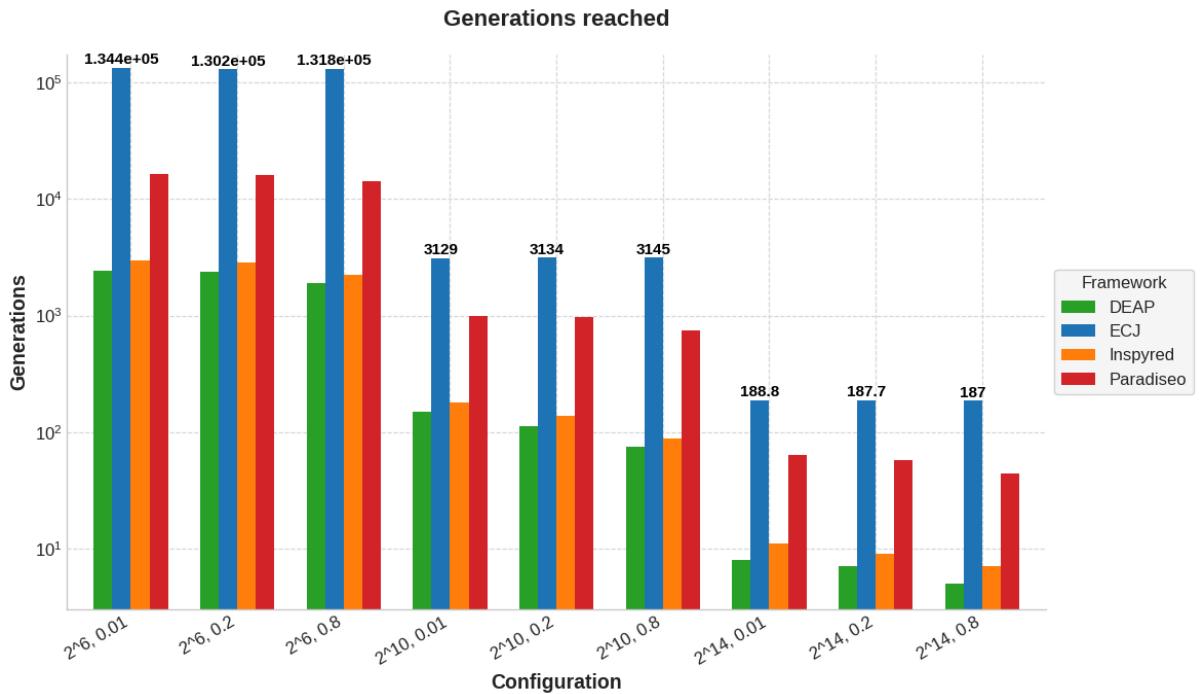
Para representar el comportamiento evolutivo y la calidad de las soluciones se han propuesto cuatro métricas a evaluar: *fitness* inicial, variación del *fitness*, *fitness* máximo alcanzado y número de generaciones alcanzadas.

En primer lugar, en la figura 61 se aprecian tres órdenes de magnitud de diferencia en el número de generaciones alcanzadas:

- Con  $N = 2^6$  los *frameworks* completan casi 134.400 generaciones en el caso de ECJ, alrededor de 3.400 y 3.800 generaciones para DEAP e Inspyred, y 12.500 generaciones para ParadisEO, todo ello antes de los dos minutos tope de ejecución.
- Al crecer a  $N = 2^{10}$ , ECJ reduce el recuento a 3.130 generaciones, mientras DEAP e Inspyred sólo llegan a 120 y 200 generaciones, respectivamente, y ParadisEO a 1.000.
- Para  $N = 2^{14}$ , ECJ completa 188 generaciones, DEAP e Inspyred caen por debajo de 10, y ParadisEO se sitúa cerca de 75 generaciones.

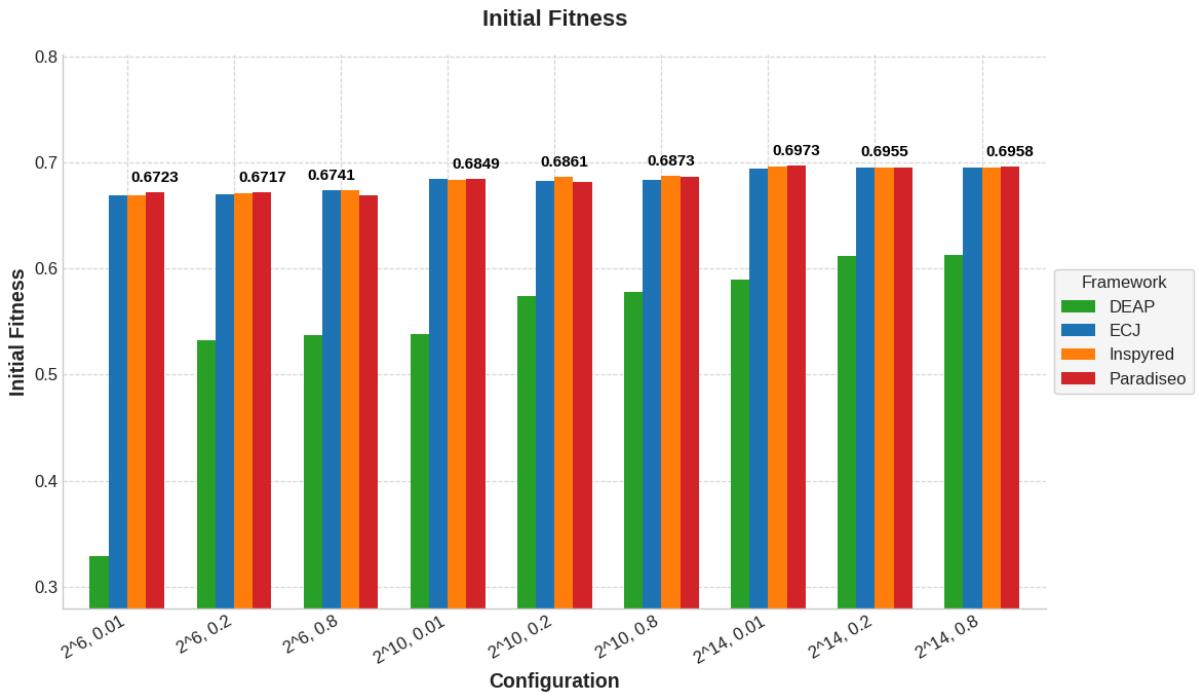
Esta reducción tan drástica confirma una “ley inversa” existente entre generaciones alcanzadas y tamaño de la población en la cual explican que para una condición de parada similar, los mecanismos de variación operan menos veces cuanto mayor sea el tamaño de la población.

Figura 61: número máximo de generaciones alcanzadas de Rosenbrock



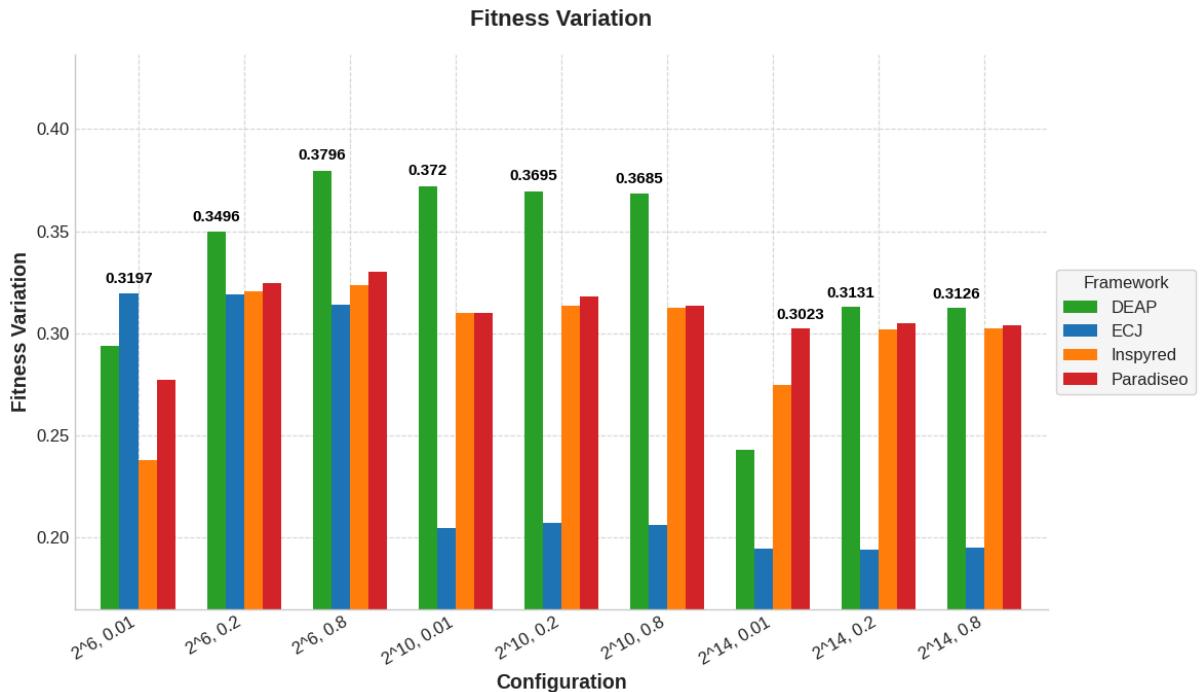
Fuente: elaboración propia

Figura 62: *fitness* inicial promedio de Rosenbrock



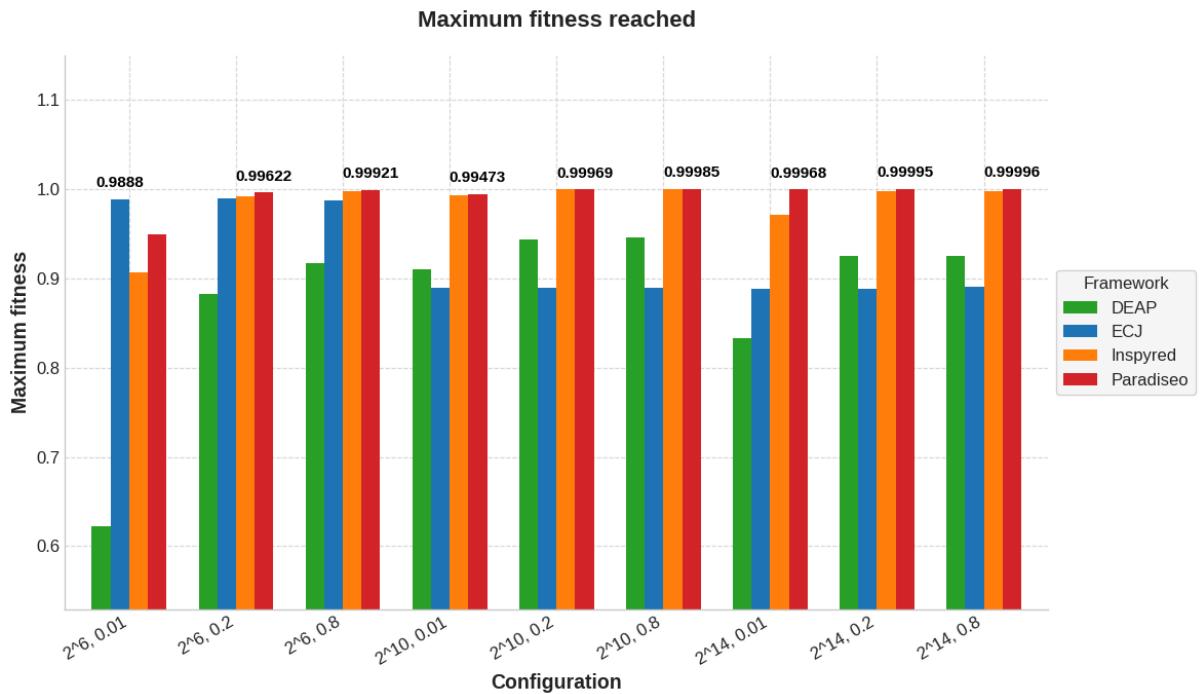
Fuente: elaboración propia

Figura 63: variación promedio del *fitness* de Rosenbrock



Fuente: elaboración propia

Figura 64: *fitness* máximo promedio alcanzado de Rosenbrock



Fuente: elaboración propia

En segundo lugar, la figura 62 muestra el *fitness* inicial: ECJ e Inspyred parten de un valor medio de 0,685, ParadisEO de 0,674, mientras que DEAP arranca en 0,345, reflejo de su mayor dispersión en la población inicial.

En tercer lugar, la figura 63 expone la variación del *fitness*, calculado como la diferencia entre el *fitness* máximo alcanzado y el inicial. Dentro de cada tamaño de población, la mayor ganancia se produce con  $p_c = 0,8$ :

- DEAP alcanza un pico de 0,380 puntos en  $N = 2^6$ ,  $p_c = 0,8$  y desciende hasta 0,245 en  $N = 2^{14}$ ,  $p_c = 0,01$ .
- Inspyred sobresale en  $N = 2^6$ ,  $p_c = 0,8$  con 0,326, cayendo a 0,275 en  $N = 2^{14}$ ,  $p_c = 0,01$ .
- ECJ se mantiene casi plano entre 0,205 y 0,320, siendo el más conservador.
- ParadisEO presenta variaciones moderadas de 0,278 a 0,331, con ligera tendencia a la baja al crecer N.

Por último, en la figura 64 se muestra la calidad de la solución alcanzada, que en general se maximiza en poblaciones medianas y altas con probabilidades altas de cruce:

- ParadisEO e Inspyred rozan el óptimo teórico, superando 0,999 en  $N = 2^6$ ,  $p_c = 0,8$  y alcanzando casi el óptimo perfecto en configuraciones de  $N = 2^{14}$ .
- ECJ obtiene 0,9888 en  $N = 2^6$ ,  $p_c = 0,01$ , pero cae a 0,892 en  $N = 2^{14}$ ,  $p_c = 0,8$ .
- DEAP, aunque mejora con N, sólo llega a 0,949 en  $N = 2^{10}$ ,  $p_c = 0,8$ .

Este conjunto de resultados confirma que, para Rosenbrock, los frameworks basados en estrategias de exploración más agresivas (Inspyred y ParadisEO) consiguen alcanzar óptimos cercanos a 1 con menos generaciones, mientras que ECJ y DEAP ofrecen un comportamiento más estable pero con óptimos y recuentos de generaciones menores.

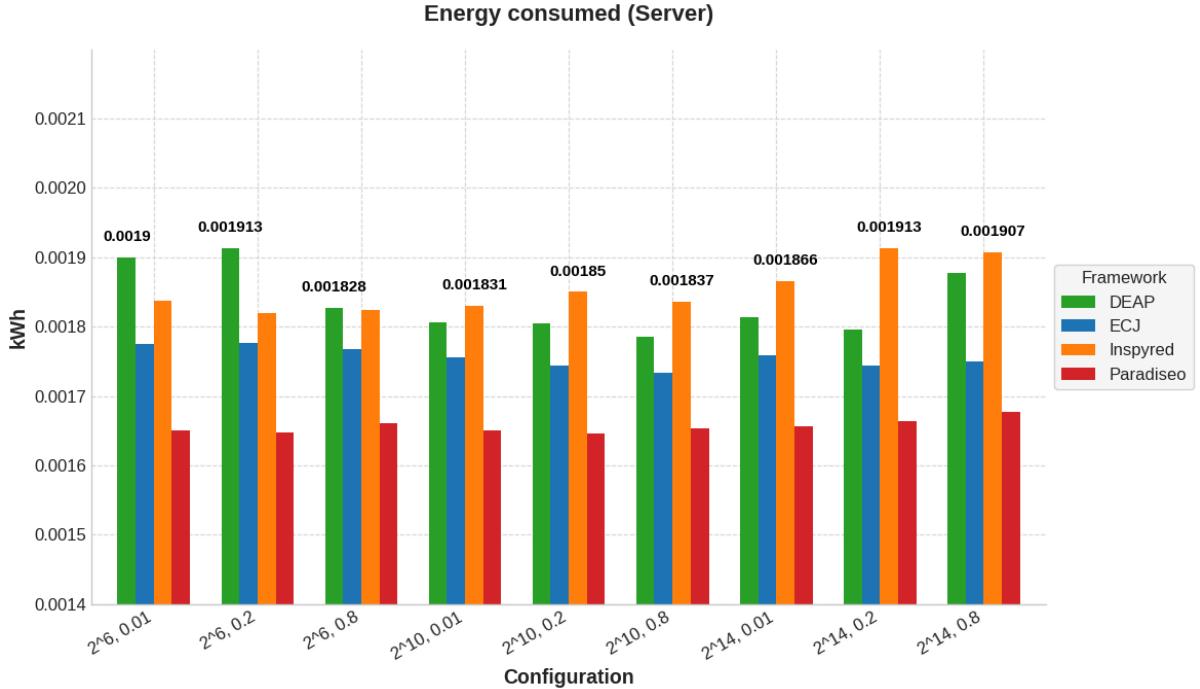
#### 4.3.4. Consumo energético en servidor

Como se aprecia en las figuras 65 y 66, el orden jerárquico de consumo energético en el servidor se mantiene casi estable a lo largo de las distintas configuraciones evaluadas, con un solo cruce entre frameworks, lo que facilita su comparación directa.

Inspyred es, de forma consistente, el framework que más energía consume, con valores que van desde los 0,001828 kWh ( $N = 2^6$ ,  $p_c = 0,8$ ) hasta un pico de 0,001913 kWh ( $N = 2^{14}$ ,  $p_c = 0,2$ ), siendo este el valor más alto registrado en todo el conjunto de pruebas.

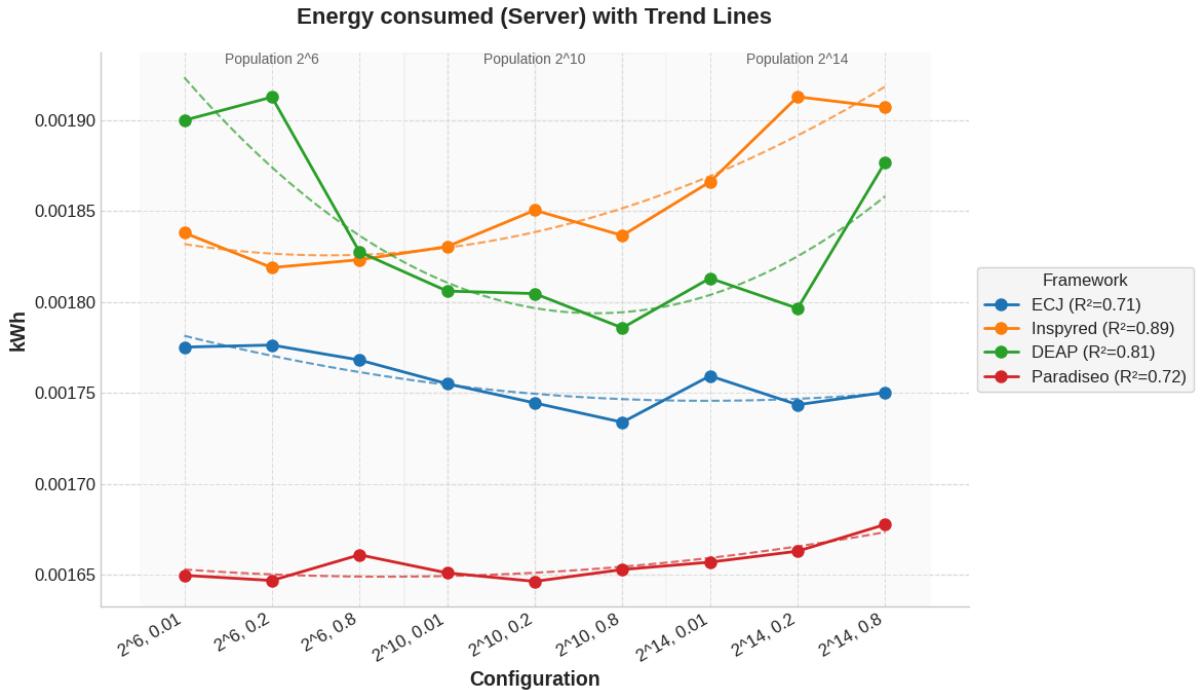
DEAP ocupa la segunda posición en consumo, iniciando en 0,001900 kWh ( $N = 2^6$ ,  $p_c = 0,01$ ) y mostrando variaciones significativas hasta alcanzar los 0,001880 kWh ( $N = 2^{14}$ ,  $p_c = 0,8$ ). Su comportamiento es más irregular que el de otros frameworks, con una notable caída en poblaciones intermedias.

Figura 65: energía consumida por el servidor en las ejecuciones de Rosenbrock



Fuente: elaboración propia

Figura 66: tendencia de la energía consumida por el servidor en las ejecuciones de Rosenbrock



Fuente: elaboración propia

ECJ, por su parte, se posiciona como el segundo framework más eficiente, con registros que oscilan entre los 0,001778 kWh ( $N = 2^6$ ,  $p_c = 0,01$ ) y los 0,001734 kWh ( $N = 2^{10}$ ,  $p_c = 0,8$ ), manteniéndose consistentemente por debajo de Inspyred y DEAP. Su comportamiento moderadamente decreciente apunta a una buena escalabilidad energética.

Paradiseo destaca como el más eficiente, con valores consistentemente bajos y estables, que van de los 0,001648 kWh ( $N = 2^6$ ,  $p_c = 0,01$ ) hasta los 0,001678 kWh ( $N = 2^{14}$ ,  $p_c = 0,8$ ), siendo el único *framework* cuya curva de consumo nunca supera los 0,0017 kWh.

La diferencia absoluta entre el *framework* más eficiente (Paradiseo en  $N = 2^6$ ,  $p_c = 0,01$ ) y el más costoso (Inspyred en  $N = 2^{14}$ ,  $p_c = 0,2$ ) es de 0,000265 kWh, lo que representa un 16,1 % de margen relativo, una cifra moderada si se compara con entornos más limitados como portátiles o sistemas embebidos. Este estrechamiento en las diferencias energéticas sugiere que la capacidad computacional del servidor amortigua las ineficiencias estructurales propias de cada *framework*.

Además, la figura de líneas con tendencia polinomial muestra con claridad las dinámicas de escalado energético:

- DEAP ( $R^2 = 0,81$ ) presenta una curva en forma de V pronunciada con una caída notable entre  $N = 2^6$  y  $N = 2^{10}$ , seguida de una recuperación ascendente significativa. Esto indica que su eficiencia mejora considerablemente con poblaciones intermedias, pero se deteriora notablemente con poblaciones muy grandes.
- Inspyred ( $R^2 = 0,89$ ) describe una curva convexa con tendencia general creciente, lo que denota un empeoramiento progresivo de su eficiencia a medida que se incrementa el tamaño poblacional, posiblemente debido a un mayor *overhead* interpretativo de Python.
- ECJ ( $R^2 = 0,71$ ) muestra una tendencia descendente bastante regular con ligeras oscilaciones, lo que evidencia un aprovechamiento progresivo de los recursos del servidor gracias a su implementación en Java compilado y optimizaciones de la JVM.
- Paradiseo ( $R^2 = 0,72$ ) mantiene una línea relativamente plana con una ligera tendencia ascendente, reflejo de su perfil energético excepcionalmente estable y bajo, propio de una arquitectura con bajo overhead desde el inicio, probablemente por su implementación nativa en C++.

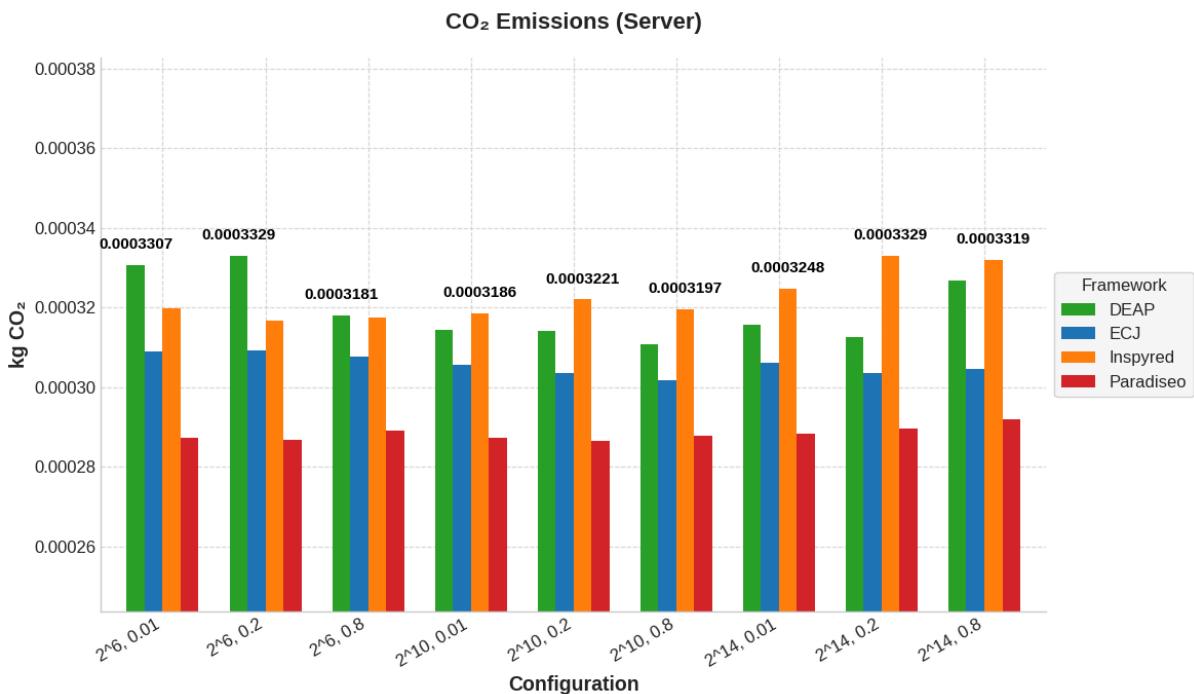
En conjunto, estos resultados corroboran que, en entornos de alta capacidad como un servidor, las diferencias de eficiencia energética entre *frameworks* tienden a suavizarse, aunque se mantienen suficientemente marcadas como para justificar la elección de herramientas más eficientes como Paradiseo o ECJ frente a opciones más costosas como Inspyred o DEAP.

#### 4.3.5. Emisiones totales de CO<sub>2</sub> en servidor

En las figuras 67 y 68 se expone el orden jerárquico de emisiones de CO<sub>2</sub> en el servidor. En las nueve configuraciones ejecutadas ( $N \in \{2^6, 2^{10}, 2^{14}\}$  y probabilidad de cruce  $p_c \in \{0,01, 0,2, 0,8\}$ ), se observa que:

- Inspyred ocupa el primer puesto como el *framework* más emisivo en ocho de las nueve configuraciones, con emisiones que oscilan desde 0,0003181 kg CO<sub>2</sub> ( $N = 2^6$ ,  $p_c = 0,8$ ) hasta un pico de 0,0003329 kg CO<sub>2</sub> ( $N = 2^{14}$ ,  $p_c = 0,2$ ). La tendencia polinomial ( $R^2 = 0,89$ ) confirma una curva convexa ascendente que refleja el creciente *overhead* de Python conforme aumenta la complejidad poblacional, penalizando progresivamente en consumo energético y, por tanto, en emisiones.
- DEAP queda en segundo lugar en términos de emisiones totales, mostrando un comportamiento variable que parte de 0,0003307 kg CO<sub>2</sub> ( $N = 2^6$ ,  $p_c = 0,01$ ), desciende notablemente hasta un mínimo de 0,0003106 kg CO<sub>2</sub> ( $N = 2^{10}$ ,  $p_c = 0,8$ ) y repunta hasta 0,0003271 kg CO<sub>2</sub> ( $N = 2^{14}$ ,  $p_c = 0,8$ ). Su línea de tendencia en forma de V ( $R^2 = 0,81$ ) sugiere que el *framework* optimiza su rendimiento con poblaciones intermedias, pero se ve penalizado tanto con poblaciones pequeñas como grandes.
- ECJ exhibe un perfil de emisiones moderadamente bajo y estable, oscilando entre 0,0003021 kg CO<sub>2</sub> ( $N = 2^{10}$ ,  $p_c = 0,8$ ) y 0,0003096 kg CO<sub>2</sub> ( $N = 2^6$ ,  $p_c = 0,01$ ), con una tendencia descendente suave ( $R^2 = 0,71$ ) que refleja la amortización progresiva del *warm-up* de la JVM frente a la elevada capacidad de procesamiento paralelo del servidor.
- Paradiseo destaca como el *framework* menos emisivo, con la banda más estrecha de emisiones entre 0,0002855 kg CO<sub>2</sub> ( $N = 2^{10}$ ,  $p_c = 0,2$ ) y 0,0002918 kg CO<sub>2</sub> ( $N = 2^{14}$ ,  $p_c = 0,8$ ). Su línea de tendencia prácticamente plana ( $R^2 = 0,72$ ) denota un escalado casi lineal y un perfil de ejecución de bajo *overhead*, característico de su implementación nativa en C++.

Figura 67: emisiones totales de CO<sub>2</sub> por el servidor en las ejecuciones de Rosenbrock

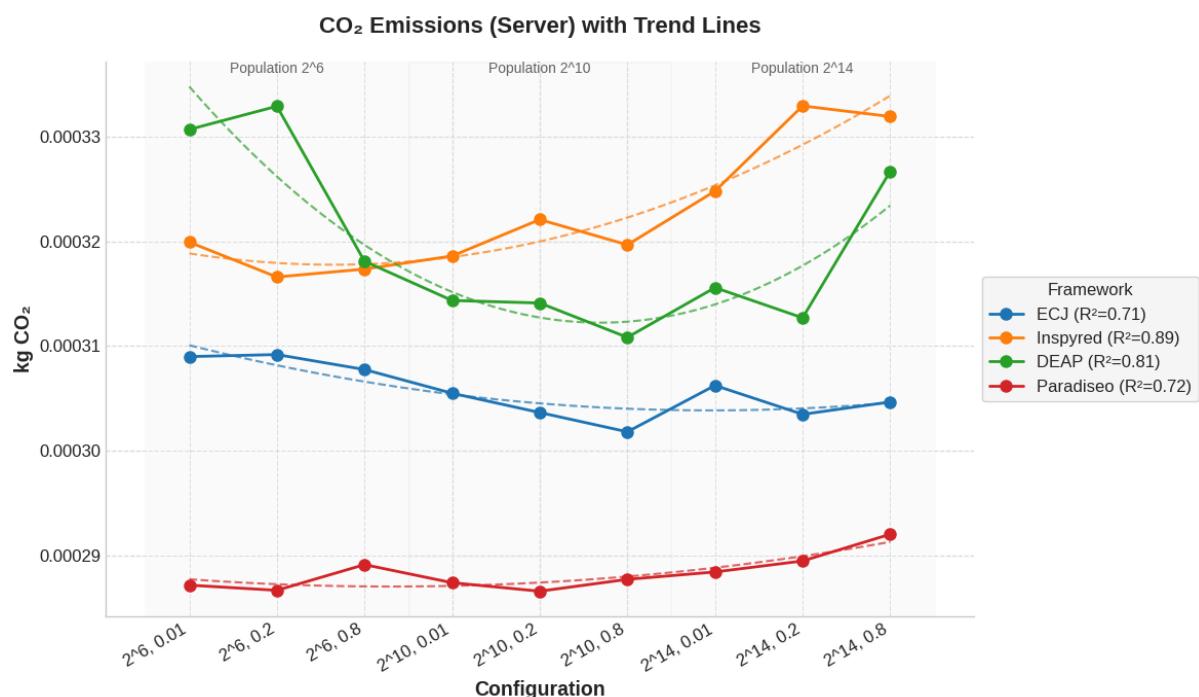


Fuente: elaboración propia

La brecha absoluta entre el más contaminante (Inspyred en  $N = 2^{14}$ ,  $p_c = 0,8$ ) y el menos (Paradiseo en  $N = 2^{10}$ ,  $p_c = 0,2$ ) es de 0,0000474 kg CO<sub>2</sub>, aproximadamente un 14,2 % del valor máximo. Esta diferencia, aunque significativa, es considerablemente menor en términos relativos comparada con entornos más limitados, confirmando que a mayor paralelismo y capacidad computacional, el coste fijo de arranque y sincronización se distribuye mejor, reduciendo la variabilidad en emisiones entre *frameworks*.

El comportamiento general de las emisiones sigue fielmente el patrón del consumo energético, lo que es esperado dado que las emisiones de CO<sub>2</sub> son directamente proporcionales a la energía consumida.

Figura 68: tendencia de las emisiones totales de CO<sub>2</sub> por el servidor en las ejecuciones de Rosenbrock



Fuente: elaboración propia

#### 4.3.6. Evolución del *fitness* en servidor

Para evaluar el comportamiento evolutivo y la calidad de las soluciones de los *frameworks* DEAP, ECJ, Inspyred y ParadisEO, se han analizado cuatro métricas fundamentales: número de generaciones completadas, *fitness* inicial, variación del *fitness* y *fitness* máximo alcanzado.

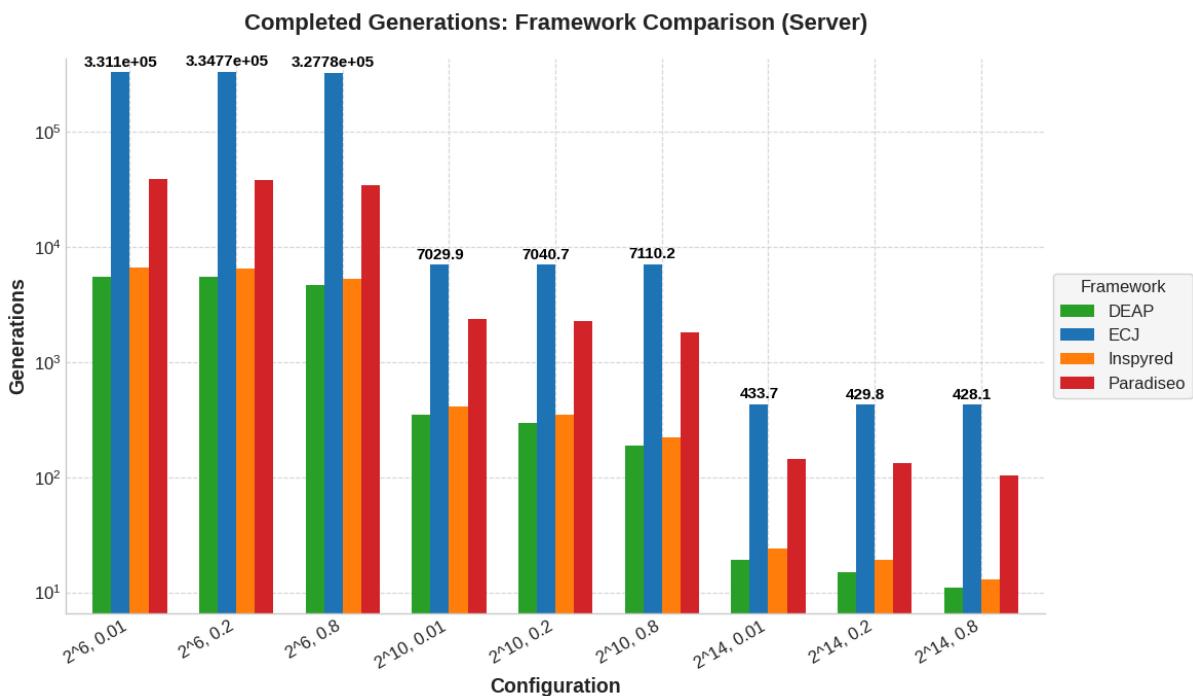
En la figura 69 se observa una clara relación inversa entre el tamaño de población y el número de generaciones alcanzadas, confirmando los principios teóricos establecidos con anterioridad:

- Con  $N = 2^6$  (64 individuos): ECJ domina significativamente con aproximadamente 330.000 generaciones completadas en las tres configuraciones, manteniendo una consistencia notable independientemente de la probabilidad de cruce. ParadisEO

alcanza valores intermedios cercanos a las 30.000 generaciones, mientras que DEAP e Inspyred se sitúan en rangos inferiores entre 5.000 y 7.000 generaciones.

- Con  $N = 2^{10}$  (1.024 individuos): Se produce una reducción drástica donde ECJ mantiene su liderazgo con aproximadamente 7.000 generaciones, seguido muy de cerca por ParadisEO. Esta configuración marca un punto de equilibrio donde los *frameworks* muestran comportamientos más homogéneos, con DEAP e Inspyred alcanzando valores cercanos a 300 y 400 generaciones.
- Con  $N = 2^{14}$  (16.384 individuos): La reducción se intensifica hasta rangos de 10 a 400 generaciones. ECJ sigue liderando con aproximadamente 430 generaciones, mientras que los demás *frameworks* alcanzan valores muy bajos, especialmente DEAP e Inspyred, que apenas superan las 20 generaciones.

Figura 69: número máximo de generaciones alcanzadas de Rosenbrock

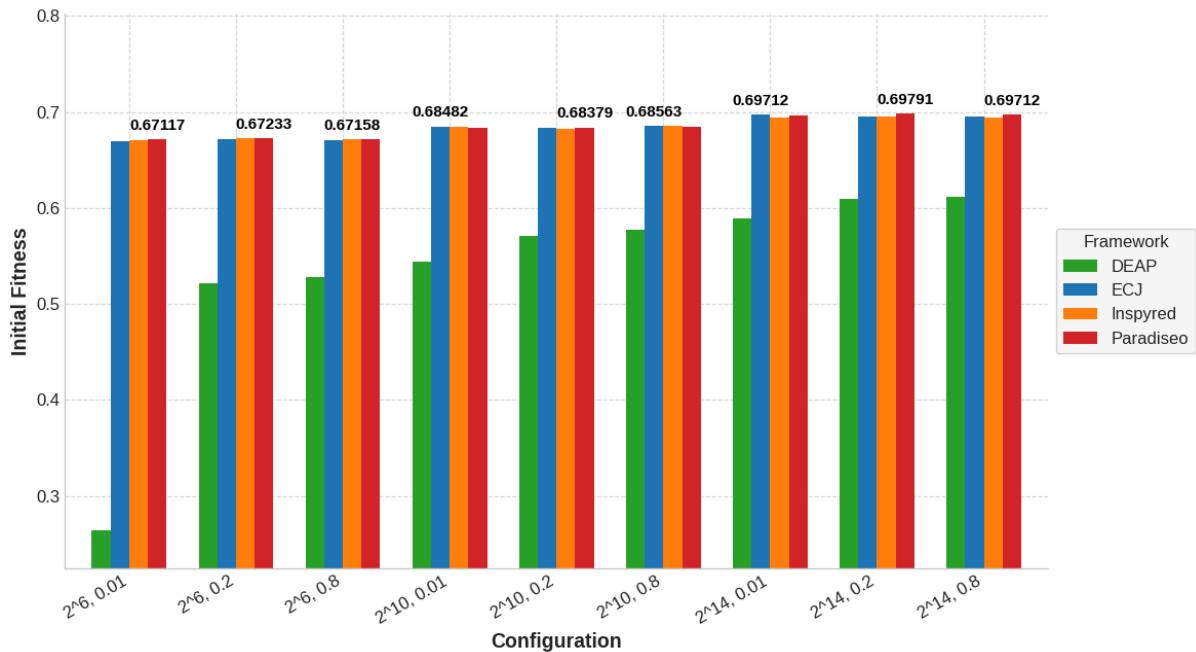


Fuente: elaboración propia

El análisis del *fitness* inicial mostrado en la figura 70 revela patrones de inicialización diferenciados entre *frameworks*. ECJ, Inspyred y ParadisEO mantienen valores consistentes entre 0,67 y 0,70 en todas las configuraciones, mostrando estrategias de inicialización estables. En contraste, DEAP presenta un comportamiento peculiar con valores significativamente menores (0,26 a 0,61) en configuraciones de población pequeña, pero convergiendo hacia valores similares al resto de *frameworks* conforme aumenta el tamaño poblacional.

Figura 70: *fitness* inicial promedio de Rosenbrock

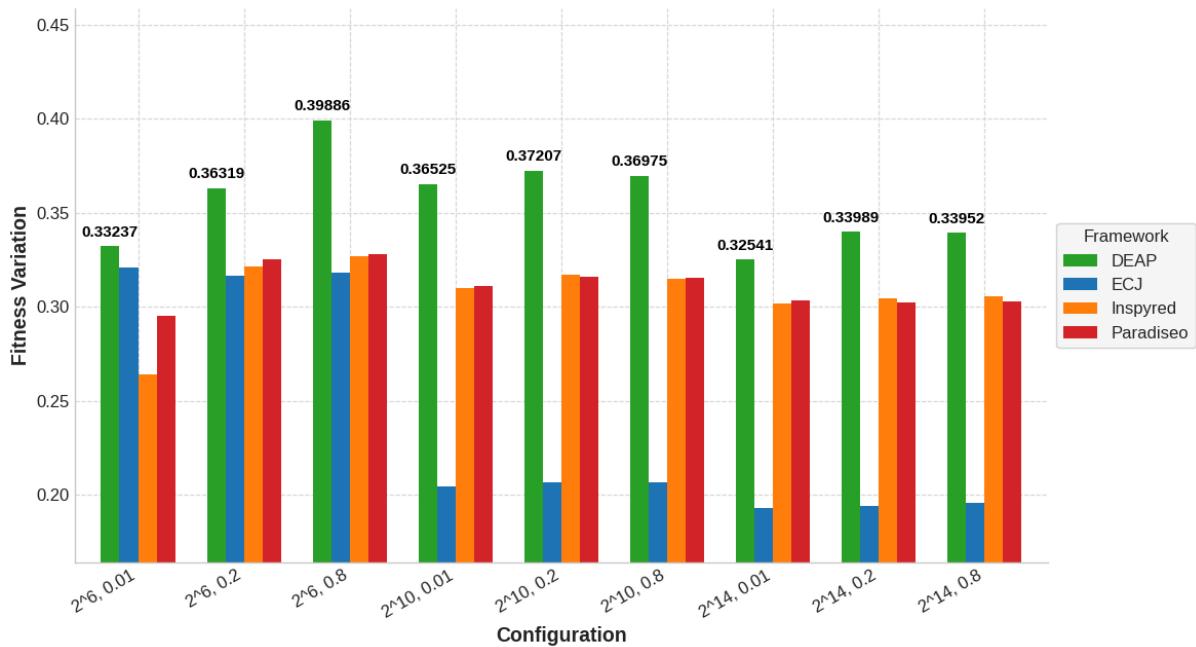
**Initial Fitness: Framework Comparison (Server)**



Fuente: elaboración propia

Figura 71: variación promedio del *fitness* de Rosenbrock

**Fitness Variation: Framework Comparison (Server)**



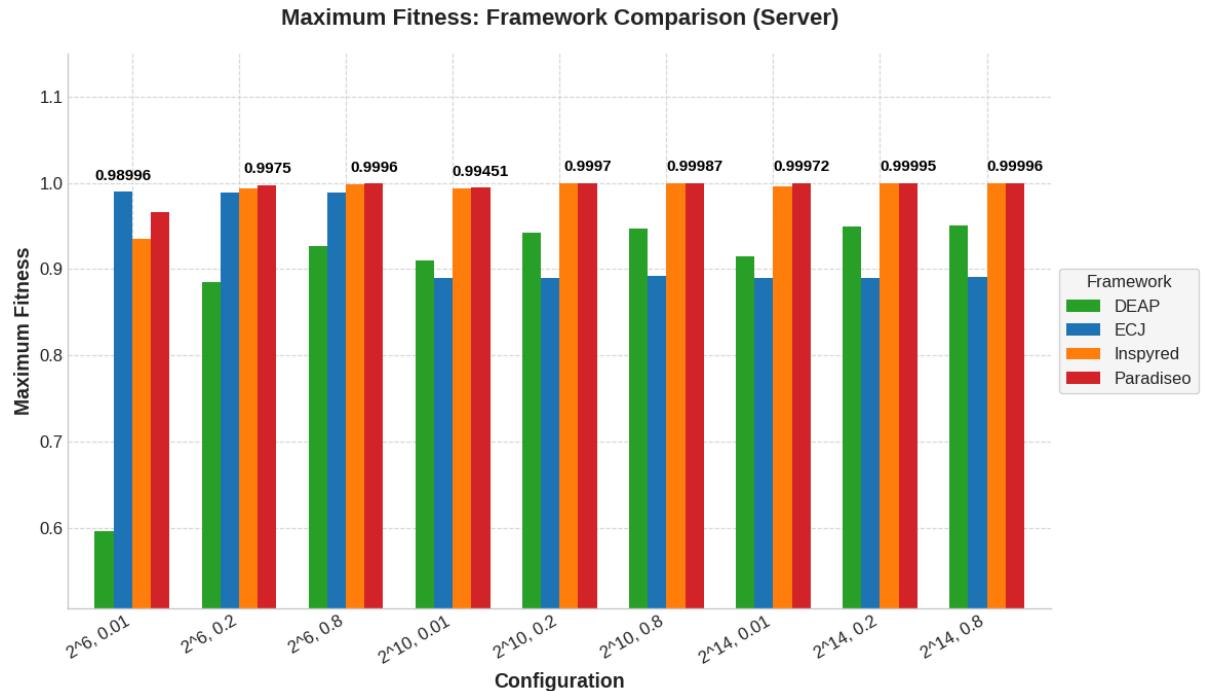
Fuente: elaboración propia

La variación del *fitness* visible en la figura 71 confirma la influencia directa de la probabilidad de cruce en la exploración del espacio de soluciones. Se observa

consistentemente que configuraciones con  $p_c = 0,8$  generan las mayores variaciones en todos los *frameworks*:

- DEAP destaca notablemente en poblaciones pequeñas y medianas, alcanzando la variación máxima absoluta de 0,4 puntos en  $N = 2^6$ ,  $p_c = 0,8$ . Sin embargo, su rendimiento se degrada significativamente en poblaciones grandes donde la variación se reduce a 0,33.
- ECJ mantiene un comportamiento estable y predecible, con variaciones que oscilan entre 0,19 y 0,32 puntos, mostrando menor sensibilidad a los cambios en la probabilidad de cruce pero mayor consistencia entre configuraciones.
- Inspyred presenta un perfil de rendimiento equilibrado, especialmente efectivo en poblaciones medianas donde alcanza variaciones competitivas cercanas a 0,31 puntos.
- ParadisEO muestra la mayor estabilidad entre configuraciones, manteniendo variaciones consistentes alrededor de 0,3 y 0,33 puntos independientemente del tamaño poblacional.

Figura 72: *fitness* máximo promedio alcanzado de Rosenbrock



Fuente: elaboración propia

El análisis del *fitness* máximo revela patrones de convergencia hacia el óptimo teórico que varían significativamente según la configuración:

- En  $N = 2^6$  se observa un comportamiento heterogéneo donde los *frameworks* muestran sus mayores diferencias. DEAP alcanza apenas 0,6 en  $p_c = 0,01$  pero mejora progresivamente hasta 0,93 en configuraciones con mayor cruce, mientras ECJ mantiene consistencia cerca de 0,99 en todas las probabilidades.

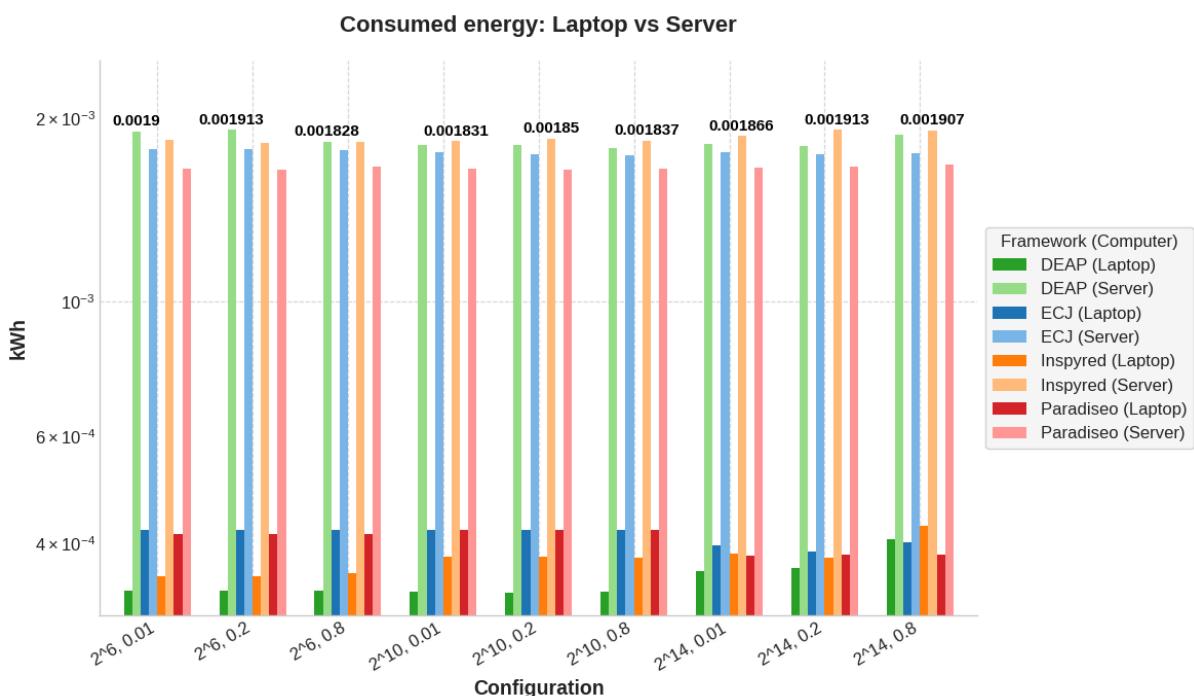
- En  $N = 2^{10}$  todos los *frameworks* convergen hacia el óptimo teórico con valores superiores a 0,94, siendo esta la configuración más efectiva globalmente para la calidad de solución. Los *frameworks* muestran rendimientos muy similares, sugiriendo que esta configuración poblacional representa un punto óptimo de equilibrio.
- En  $N = 2^{14}$  se mantiene la convergencia hacia valores óptimos (0,91 y 0,99996), aunque con ligeras variaciones entre *frameworks*. La alta densidad poblacional compensa las menores generaciones completadas, permitiendo explorar eficientemente el espacio de soluciones.

En resumen, todo lo visto revela que ECJ destaca por su capacidad de procesamiento, completando significativamente más generaciones que sus competidores, especialmente en poblaciones pequeñas. DEAP muestra un comportamiento variable pero con alta capacidad de mejora cuando las condiciones son favorables. ParadisEO sobresale por su estabilidad y consistencia entre configuraciones, mientras que Inspyred presenta un rendimiento equilibrado sin destacar en aspectos específicos.

La configuración  $N = 2^{10}$  emerge como el punto óptimo donde todos los *frameworks* logran convergencia hacia soluciones de alta calidad, sugiriendo que tamaños poblacionales intermedios ofrecen el mejor balance entre eficiencia computacional y calidad de solución.

#### 4.3.7. Consumo energético en el portátil frente al servidor

Figura 73: comparación de energía consumida de Rosenbrock entre portátil y servidor

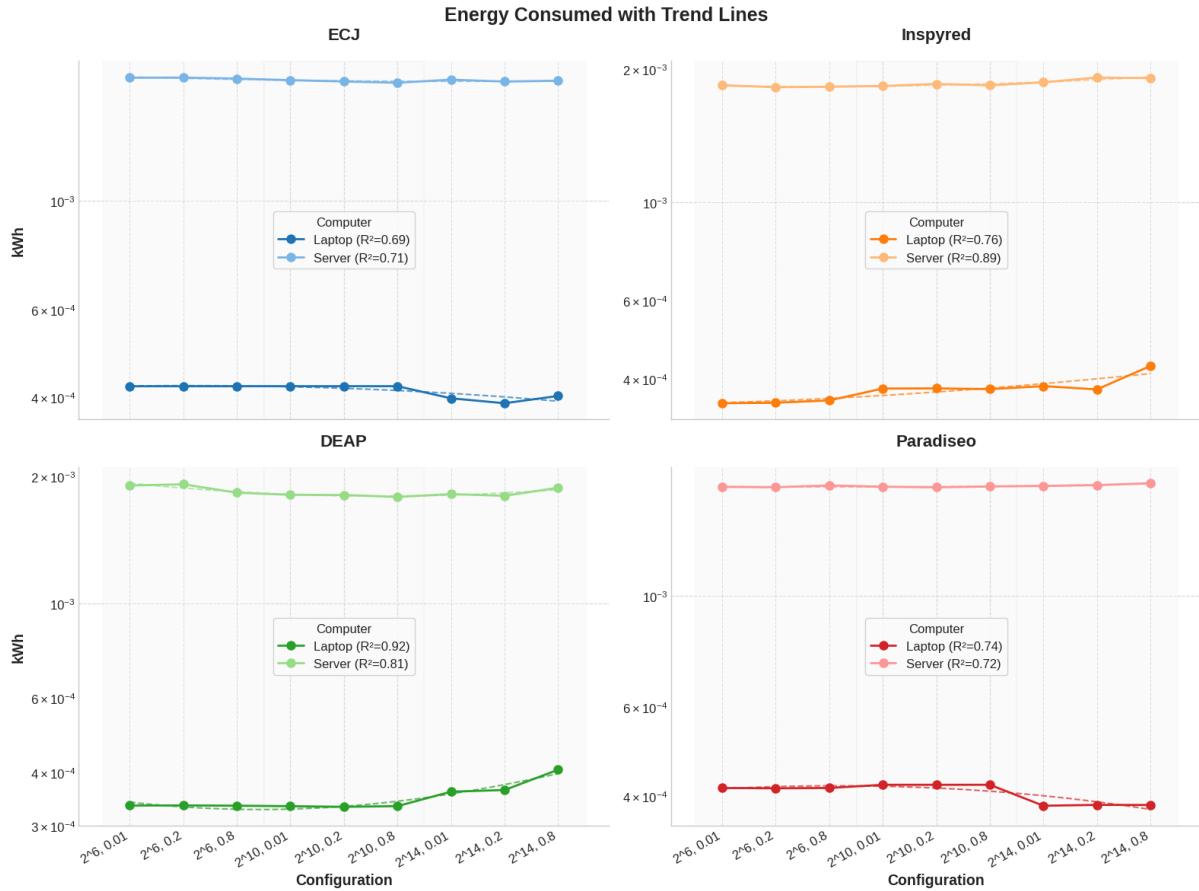


Fuente: elaboración propia

En las figuras 73 y 74 se aprecia que, para todas las configuraciones y *frameworks*, el servidor consume sistemáticamente casi 5 veces más energía que el portátil. Por ejemplo, en

DEAP, con  $N = 2^6$  y  $p_c = 0,01$  el portátil consume, aproximadamente, 0,00034 kWh frente a los casi 0,0019 kWh del servidor, ratio que se mantiene muy similar para ECJ (0,00042 frente 0,0019 kWh), Inspyred (0,00036 frente 0,00191 kWh) y ParadisEO (0,00041 frente 0,00175 kWh). Esta diferencia del orden de la unidad de magnitud confirma el efecto de que, aunque los servidores ofrecen mayor potencia de cálculo, su coste energético absoluto es sensiblemente superior.

Figura 74: comparación desglosada de energía consumida de Rosenbrock entre portátil y servidor



Fuente: elaboración propia

Al analizar la evolución del consumo en función del tamaño de población (de  $2^6$  a  $2^{14}$  individuos) se observan las siguientes tendencias:

- ECJ presenta en el portátil un comportamiento prácticamente estable del consumo medio por configuración ( $R^2 = 0,69$ ), manteniéndose alrededor de 0,00042 kWh, mientras que en el servidor muestra igualmente un patrón estable ( $R^2 = 0,71$ ) en torno a 0,0019 kWh. Esta estabilidad indica que ECJ mantiene un consumo energético predecible independientemente del tamaño poblacional.
- Inspyred en el portátil muestra una tendencia ligeramente creciente conforme aumenta N ( $R^2 = 0,76$ ), partiendo de 0,00036 kWh en configuraciones pequeñas hasta alcanzar aproximadamente 0,00043 kWh en poblaciones grandes. En el servidor se muestra el coeficiente de correlación más alto ( $R^2 = 0,89$ ) con una tendencia claramente

creciente desde 0,00191 hasta 0,0019 kWh, indicando que su gasto energético se ajusta directamente al tamaño de la población.

- DEAP mantiene el patrón más estable en el portátil ( $R^2 = 0,92$ ), con un consumo prácticamente constante alrededor de 0,0018 y 0,0019 kWh, y exhibe una ligera tendencia decreciente en el servidor ( $R^2 = 0,81$ ), comenzando en 0,0019 kWh y finalizando cerca de 0,00178 kWh, probablemente debido a optimizaciones en la gestión de recursos cuando la población se incrementa significativamente.
- ParadisEO muestra un consumo estable en portátil ( $R^2 = 0,74$ ) oscilando entre 0,00041 y 0,00039 kWh, y presenta una correlación moderada en servidor ( $R^2 = 0,72$ ) con una tendencia ligeramente decreciente desde 0,00175 hasta 0,00173 kWh, lo cual puede ser valioso cuando se requiere predictibilidad energética.

Así, se corrobora la desigual eficiencia energética entre dispositivos de distintas prestaciones: los servidores consumen significativamente más energía absoluta que los sistemas portátiles, a pesar de sus mejoras en capacidad de procesamiento. El factor multiplicativo, de aproximadamente un valor de 5, debe considerarse en el contexto del rendimiento computacional obtenido, especialmente cuando se evalúa el coste-beneficio energético en aplicaciones de computación evolutiva a gran escala.

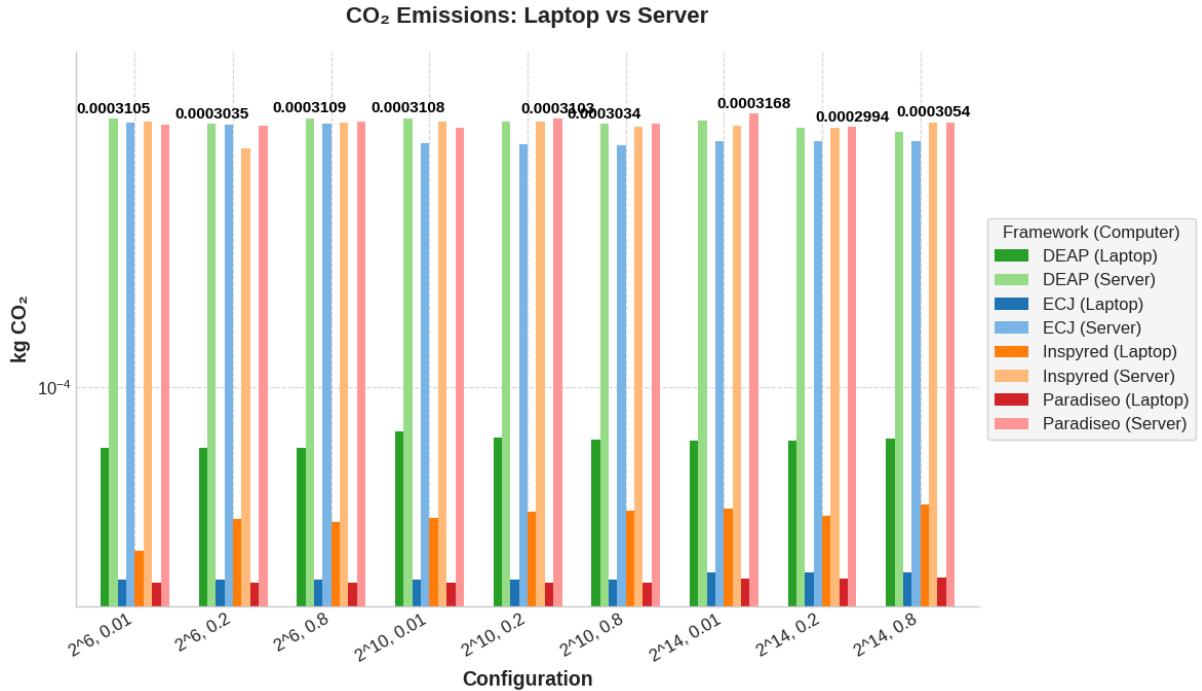
La estabilidad energética mostrada por algunos *frameworks* como DEAP y ParadisEO, junto con la predictibilidad de ECJ, contrasta con el comportamiento más variable de Inspyred, proporcionando criterios adicionales para la selección de *frameworks* en entornos donde la eficiencia energética constituye un factor crítico de decisión.

#### **4.3.8. Emisiones totales de CO<sub>2</sub> en el portátil frente al servidor**

Analizando la figura 75, se observa una clara diferencia en el comportamiento energético entre las dos arquitecturas evaluadas. El paradigma del *Green Computing*, orientado a abordar el consumo energético en entornos computacionales (Fernández de Vega *et al.*, 2016, p. 368), se manifiesta de manera diferenciada al comparar portátiles y sistemas más potentes en la ejecución de algoritmos evolutivos.

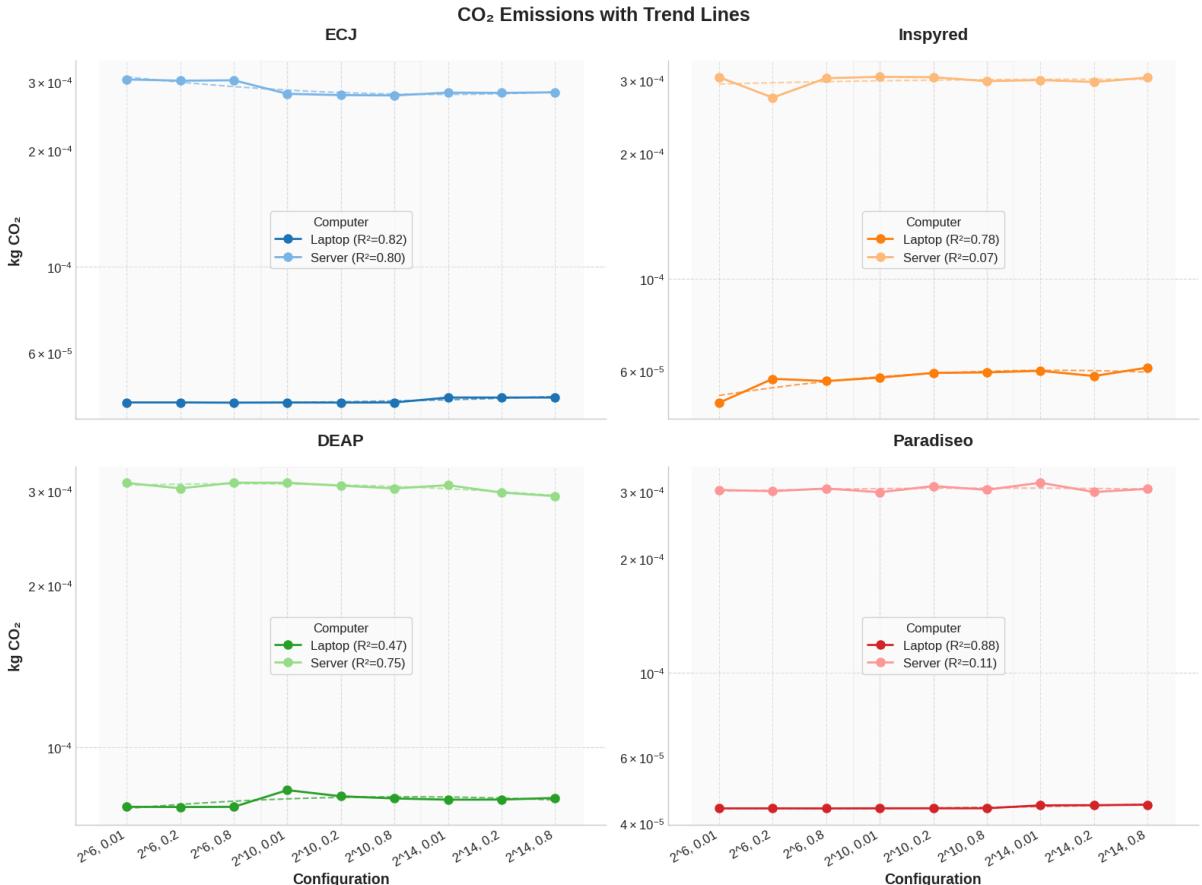
Los datos experimentales revelan una disparidad significativa en las emisiones base entre plataformas. El portátil genera emisiones que oscilan entre 0,0000589 kg CO<sub>2</sub> y 0,0000329 kg CO<sub>2</sub>, mientras que el servidor produce entre 0,000286 kg CO<sub>2</sub> y 0,000329 kg CO<sub>2</sub>. Esta diferencia de aproximadamente 5 a 10 veces más emisiones en el servidor refleja no solo la mayor potencia de procesamiento, sino también los costos energéticos asociados a la infraestructura de servidores.

Figura 75: comparación de emisiones de CO<sub>2</sub> de Rosenbrock entre portátil y servidor



Fuente: elaboración propia

Figura 76: comparación desglosada de emisiones de CO<sub>2</sub> de Rosenbrock entre portátil y servidor



Fuente: elaboración propia

Al examinar la evolución de las emisiones con el incremento del tamaño de población (desde  $2^6$  hasta  $2^{14}$  individuos), emergen patrones distintivos por *framework*:

- ECJ demuestra un comportamiento relativamente estable en ambas plataformas, con el portátil mostrando una correlación moderada ( $R^2 = 0,69$ ) y el servidor una correlación ligeramente superior ( $R^2 = 0,71$ ). Las emisiones permanecen prácticamente constantes alrededor de 0,003 kg CO<sub>2</sub>, sugiriendo una gestión eficiente de recursos independientemente del tamaño poblacional.
- Inspyred presenta el comportamiento más interesante: mientras en portátil mantiene una tendencia ascendente moderada ( $R^2 = 0,76$ ), en servidor la correlación es significativamente mayor ( $R^2 = 0,89$ ), indicando una mejor escalabilidad energética en arquitecturas de alto rendimiento. Las emisiones del servidor se mantienen estables hasta la población de tamaño  $N = 2^{10}$ , para luego mostrar un incremento controlado.
- DEAP exhibe el comportamiento más estable ( $R^2 = 0,92$  portátil,  $R^2 = 0,81$  servidor). En portátil, las emisiones se mantienen relativamente estables con una ligera tendencia descendente, mientras que en servidor muestran mayor variabilidad, posiblemente debido a la sobrecarga de gestión de hilos en configuraciones de alta concurrencia.
- ParadisEO demuestra la mayor estabilidad en portátil ( $R^2 = 0,74$ ) con emisiones prácticamente constantes, pero en servidor presenta una correlación menor ( $R^2 = 0,72$ ) con una tendencia ligeramente descendente hacia configuraciones poblacionales mayores. Este comportamiento sugiere una optimización interna que mejora la eficiencia energética con cargas de trabajo más intensas.

La figura 76 confirma que todos los *frameworks* mantienen patrones de emisión relativamente predecibles, siendo ParadisEO y ECJ los más consistentes en términos de huella de carbono, factor crucial cuando se requiere control estricto del impacto ambiental en aplicaciones de computación evolutiva a gran escala.

#### 4.3.9. Análisis la evolución del *fitness* en el portátil frente al servidor

Teniendo en consideración las figuras 61, 62, 63, 64, 69, 70, 71 y 72, se observa que tanto portátil como servidor comparten el mismo proceso de inicialización aleatoria, resultando en valores promedio de *fitness* inicial muy similares entre plataformas. Los valores se concentran alrededor de 0,67 y 0,69 para la mayoría de *frameworks* en todas las configuraciones, sin diferencias significativas entre máquinas.

Sin embargo, se observa una excepción notable en DEAP, que exhibe valores considerablemente inferiores (0,33 a 0,61) en configuraciones pequeñas ( $N = 2^6$ ), manteniéndose este patrón tanto en portátil como en servidor, lo que sugiere una diferencia inherente en la implementación del *framework* más que un efecto de la plataforma.

Por otro lado, La variación de *fitness*, calculada como la diferencia entre *fitness* máximo e inicial, muestra patrones muy similares en ambas plataformas:

- DEAP obtiene la mayor ganancia en poblaciones pequeñas e intermedias: alcanza picos de 0,35 a 0,38 en  $N = 2^6$ ,  $p_c = 0,8$  tanto en portátil como servidor, y se mantiene en torno a 0,37 y 0,40 en  $N = 2^{10}$ . Sin embargo, colapsa bruscamente a valores residuales (0,24 y 0,33) en poblaciones grandes ( $N = 2^{14}$ ) en ambas máquinas.
- ECJ registra la ganancia más estable entre plataformas, manteniéndose consistentemente en el rango 0,19 y 0,32 puntos a través de todas las configuraciones, con diferencias mínimas entre portátil y servidor.
- Inspyred muestra variaciones moderadas y consistentes, situándose entre 0,27 y 0,31 puntos en ambas máquinas, con los mejores resultados en configuraciones intermedias ( $N = 2^{10}$ ).
- ParadisEO mantiene variaciones estables alrededor de 0,27 y 0,32 puntos tanto en portátil como servidor, con un comportamiento muy similar al de Inspyred.

En términos generales, las curvas de ganancia se superponen casi idénticamente entre portátil y servidor, indicando que la mayor capacidad computacional del servidor no altera la dinámica de búsqueda, sino únicamente la velocidad de ejecución.

En adición, el *fitness* perfecto (*fitness* = 1) no se logra en ninguna configuración, y los máximos reales muestran patrones consistentes entre plataformas:

- Inspyred y ParadisEO lideran consistentemente, alcanzando valores muy cercanos al óptimo teórico en la mayoría de configuraciones, sin diferencias apreciables entre portátil y servidor.
- ECJ se mantiene estable en el rango 0,89 y 0,99 puntos en ambas máquinas, mostrando robustez pero sin alcanzar los picos de los *frameworks* anteriores.
- DEAP exhibe el comportamiento más variable: desde valores excelentes (0,88 o 0,92) en configuraciones pequeñas hasta resultados pobres (0,62 o 0,84) en poblaciones grandes, manteniendo este patrón errático tanto en portátil como servidor.

Aunque las diferencias absolutas son pequeñas entre plataformas, se observa una ligera tendencia del servidor a obtener valores marginalmente superiores en algunas configuraciones, posiblemente debido a una convergencia más estable con mayores recursos computacionales.

El recuento de generaciones completadas revela la clara ventaja computacional del servidor:

- Para poblaciones pequeñas ( $N = 2^6$ ), el servidor completa aproximadamente 330.000 generaciones frente a las 134.000 del portátil, representando un factor de aceleración de 2,46.

- Con poblaciones intermedias ( $N = 2^{10}$ ), el servidor alcanza 7.000 y 7.100 generaciones frente a las 3.100 y 3.200 en el portátil, manteniendo una proporción aproximada de 2,2.
- En poblaciones grandes ( $N = 2^{14}$ ), ambas máquinas requieren significativamente menos generaciones (de 180 a 190 en portátil y de 430 a 440 en servidor), pero el servidor mantiene su ventaja con un factor de 2,3.

ECJ domina consistentemente el throughput en ambas plataformas, especialmente notable en el servidor donde alcanza los valores máximos de generaciones completadas.

Este patrón uniforme de aceleración (factor de 2,3 a 2,5) indica que el servidor mejora proporcionalmente el ritmo de iteración sin alterar fundamentalmente el número de generaciones necesario para que los algoritmos alcancen sus criterios de parada, confirmando que las diferencias observadas son puramente de rendimiento computacional y no de comportamiento algorítmico.

Tabla 8: Resumen aproximado de las generaciones alcanzadas de Rosenbrock

<i>N y p<sub>c</sub></i>	<i>portátil</i>	<i>servidor</i>	<i>Razón servidor/portátil</i>
$2^6, 0,01$	134.440	331.000	2,46
$2^6, 0,2$	130.200	334.800	2,57
$2^6, 0,8$	131.800	327.800	2,49
$2^{10}, 0,01$	3.129	7.030	2,25
$2^{10}, 0,2$	3.134	7.041	2,25
$2^{10}, 0,8$	3.145	7.110	2,26
$2^{14}, 0,01$	189	434	2,3
$2^{14}, 0,2$	188	430	2,29
$2^{14}, 0,8$	187	428	2,29

Fuente: elaboración propia

#### 4.3.10. Análisis de la eficiencia energética en el portátil

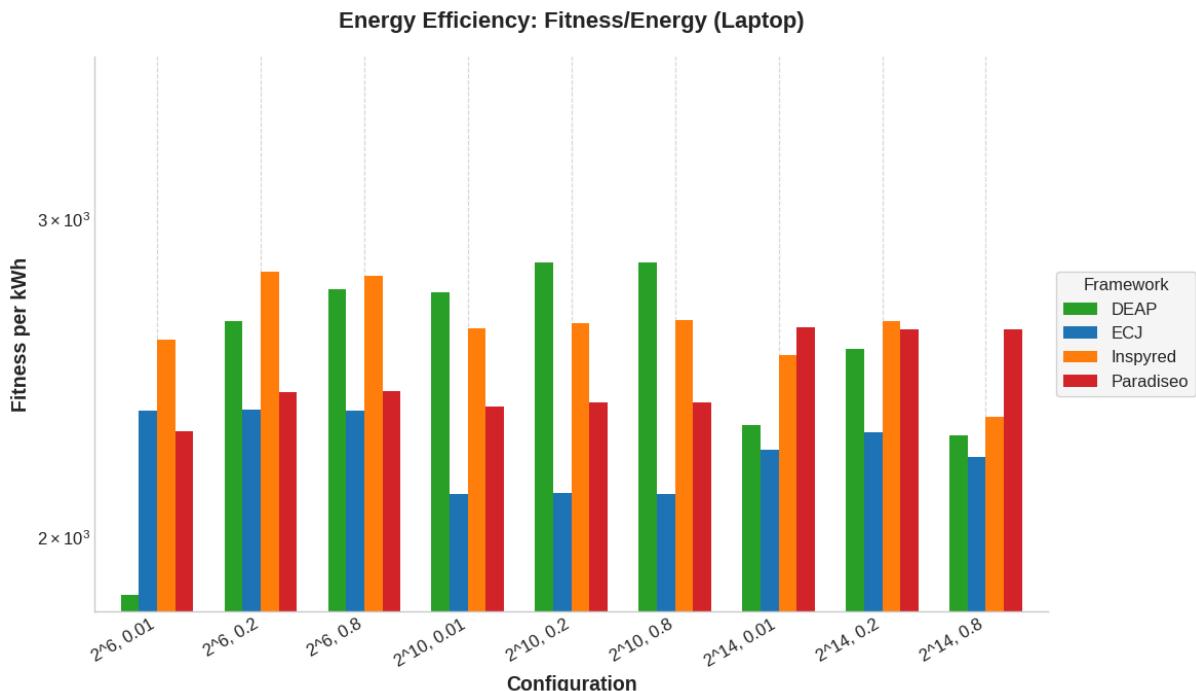
Para evaluar la eficiencia energética de cada *framework* en el portátil, se define la métrica

$$\eta = \frac{\text{Fitness máximo alcanzado}}{\text{Consumo (kWh)}}$$

que indica cuántas unidades de *fitness* se obtienen por cada kWh consumido. Así, gracias a la figura 77, se puede observar que:

- DEAP resulta ser el el *framework* más eficiente energéticamente en la mayoría de configuraciones, con valores de  $\eta$  que oscilan entre 1.700 y 2.850 *fitness/kWh*. Su desempeño sobresale, especialmente, en configuraciones de población intermedia ( $N = 2^{10}$ ), donde alcanza sus valores máximos cercanos a 2.850 *fitness/kWh*, demostrando una conversión óptima de energía en ganancia de *fitness*.
- Inspyred ocupa la segunda posición en el ranking general, con  $\eta$  comprendido entre 2.350 y 2.800 *fitness/kWh*. Muestra un comportamiento consolidado y consistente a través de diferentes configuraciones, manteniéndose competitivo especialmente en poblaciones pequeñas ( $N = 2^6$ ) donde alcanza valores cercanos a 2.800 *fitness/kWh*.
- Paradiseo presenta una eficiencia intermedia, con  $\eta$  fluctuando entre 2.300 y 2.650 *fitness/kWh*. Su implementación muestra estabilidad moderada, aunque no logra los picos de eficiencia de los *frameworks* superiores, manteniéndose en un rango consistente pero conservador.
- ECJ resulta ser el menos eficiente energéticamente, con  $\eta$  comprendido entre 2.100 y 2.370 *fitness/kWh*. A pesar de su estabilidad, su conversión energía-*fitness* es la más baja del grupo, especialmente visible en configuraciones de población grande ( $N = 2^{14}$ ) donde alcanza sus valores mínimos.

Figura 77: comparación entre *frameworks* del cálculo de  $\eta$  de Rosenbrock en el servidor



Fuente: elaboración propia

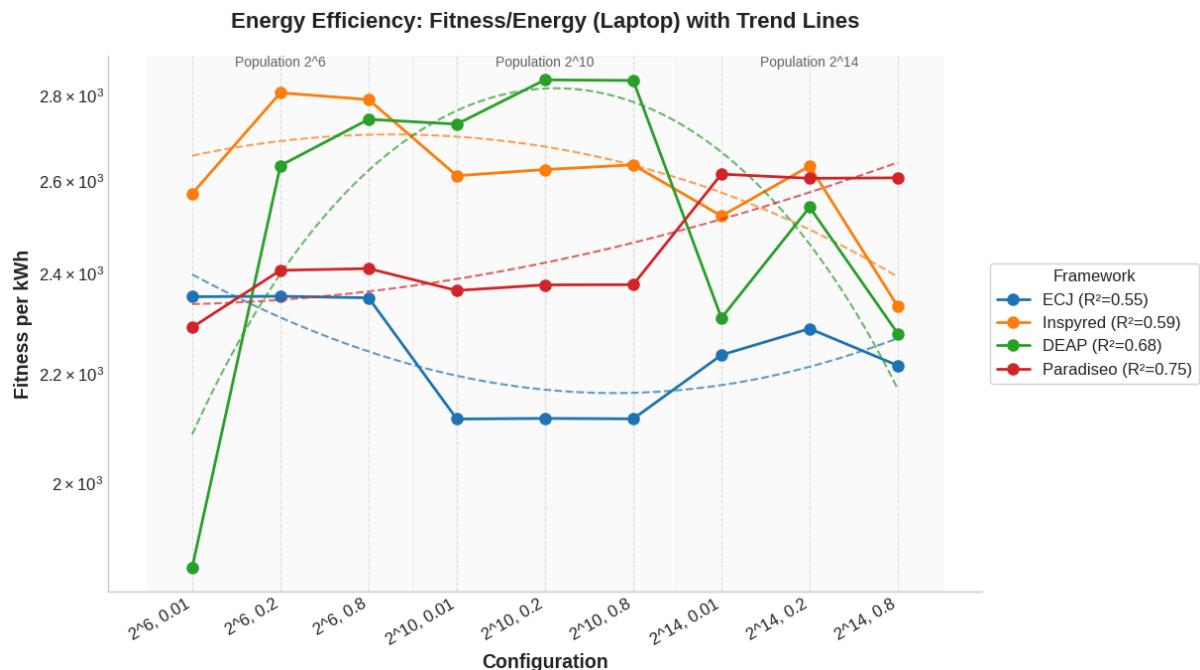
Las líneas de tendencia polinomiales visibles en la figura 78 revelan comportamientos característicos para cada *framework*:

- ECJ exhibe una tendencia moderadamente decreciente ( $R^2 = 0,55$ ), indicando que su eficiencia disminuye de manera gradual pero consistente al incrementarse el tamaño

de la población. Esta tendencia sugiere limitaciones en la escalabilidad energética del *framework*.

- Inspyred presenta la segunda mejor correlación ( $R^2 = 0,59$ ), mostrando un patrón de eficiencia que se mantiene relativamente estable en poblaciones pequeñas e intermedias, con una ligera tendencia descendente en configuraciones de mayor tamaño poblacional.
- DEAP demuestra la tendencia más marcada ( $R^2 = 0,68$ ), mostrando un comportamiento claramente optimizado para poblaciones intermedias. Su curva muestra un crecimiento inicial pronunciado desde  $N = 2^6$  hasta  $N = 2^{10}$ , seguido de una caída controlada en  $N = 2^{14}$ , confirmando un óptimo energético en configuraciones de tamaño medio.
- Paradiseo presenta la correlación más fuerte ( $R^2 = 0,75$ ), describiendo una tendencia ascendente sostenida que alcanza su máximo en configuraciones de población grande ( $N = 2^{14}$ ). Esta característica sugiere que Paradiseo optimiza mejor su eficiencia energética conforme aumenta la complejidad poblacional.

Figura 78: comparación entre tendencias de *frameworks* del cálculo de  $\eta$  de Rosenbrock en el portátil



Fuente: elaboración propia

#### 4.3.11. Análisis de la eficiencia energética en el servidor

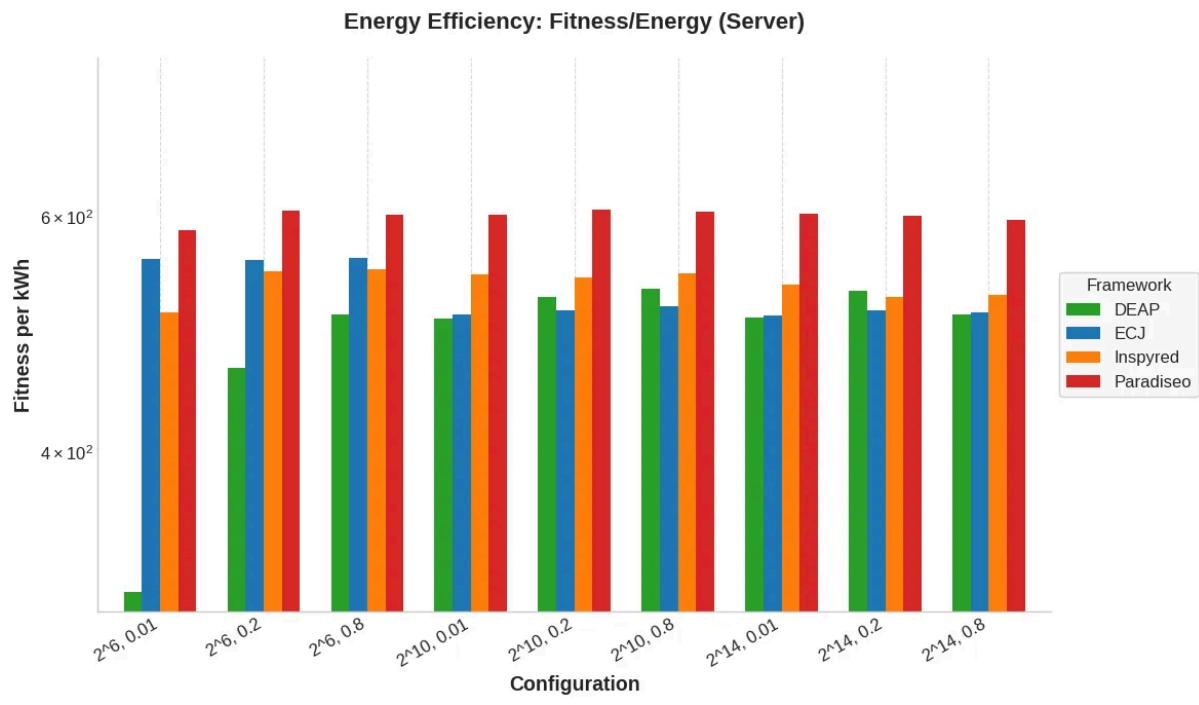
Usando la misma métrica que en el apartado anterior, se puede observar en la figura 79 que:

- Paradiseo emerge como el *framework* más eficiente energéticamente en todas las configuraciones, con valores de  $\eta$  que oscilan consistentemente entre 590 y 610 *fitness/kWh*. Su desempeño sobresale por su estabilidad excepcional, manteniendo una eficiencia prácticamente constante independientemente del tamaño poblacional o

configuración de parámetros, lo que revela una optimización superior en el aprovechamiento de recursos del servidor.

- ECJ ocupa la segunda posición en el ranking general, con  $\eta$  comprendido entre 510 y 570  $fitness/kWh$ . Muestra un comportamiento robusto en poblaciones pequeñas ( $N = 2^6$ ) donde alcanza sus valores máximos cercanos a 570  $fitness/kWh$ , pero experimenta una degradación conforme aumenta el tamaño de la población.
- Inspyred presenta una eficiencia intermedia, con  $\eta$  oscilando entre 510 y 550  $fitness/kWh$ . Su implementación demuestra mayor estabilidad en el entorno servidor comparado con portátil, manteniendo valores competitivos especialmente en configuraciones de población intermedia ( $N = 2^{10}$ ).
- DEAP resulta ser el menos eficiente energéticamente, con  $\eta$  comprendido entre 300 y 530  $fitness/kWh$ . Contrasta notablemente con su desempeño en portátil, mostrando la mayor variabilidad del grupo y alcanzando su eficiencia mínima en poblaciones pequeñas ( $N = 2^6, p_c = 0,01$ ) con apenas 300  $fitness/kWh$ .

Figura 79: comparación entre *frameworks* del cálculo de  $\eta$  de Rosenbrock en el servidor



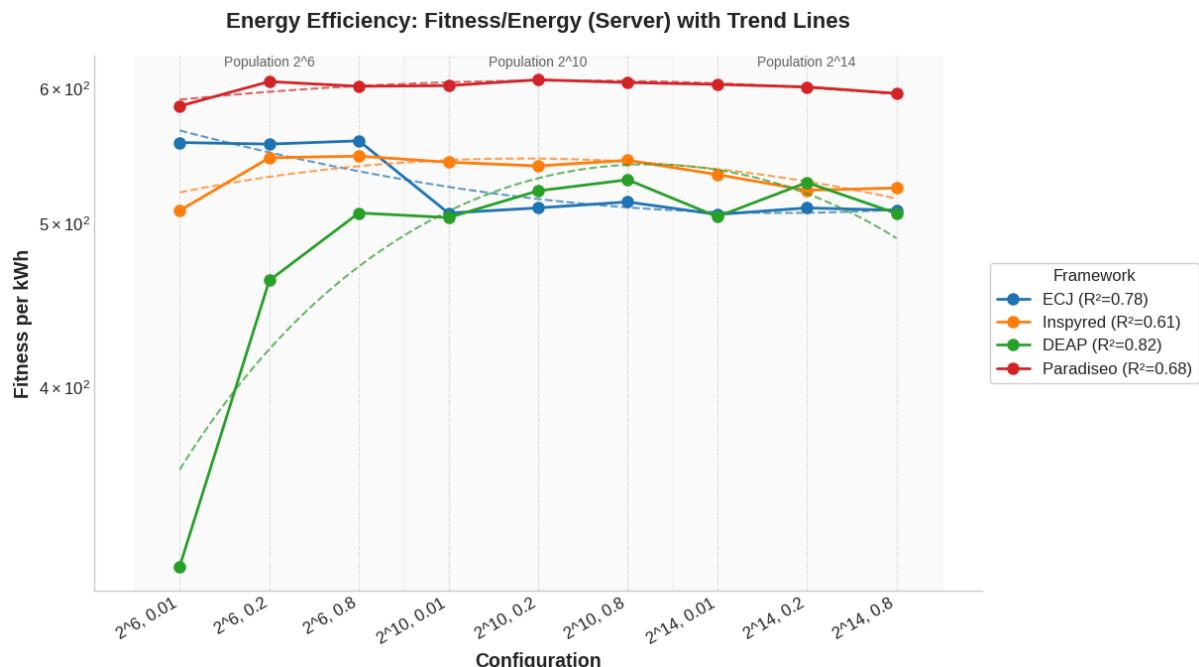
Fuente: elaboración propia

Además, las líneas de tendencia polinomiales presentes en la figura 80 revelan comportamientos característicos para cada *framework*:

- DEAP presenta la correlación más fuerte ( $R^2 = 0,82$ ), exhibiendo un comportamiento fuertemente ascendente desde poblaciones pequeñas hasta intermedias. Su curva muestra un crecimiento exponencial inicial desde  $N = 2^6$  hasta  $N = 2^{10}$ , seguido de una estabilización en  $N = 2^{14}$ , sugiriendo que requiere poblaciones de mayor tamaño para optimizar su eficiencia energética en servidor.

- ECJ demuestra una tendencia moderadamente decreciente ( $R^2 = 0,78$ ), indicando que su eficiencia disminuye de manera controlada pero consistente al incrementarse el tamaño poblacional. Esta característica sugiere que ECJ está optimizado para aprovechar mejor los recursos del servidor en configuraciones de población reducida.
- Paradiseo exhibe una tendencia prácticamente plana ( $R^2 = 0,68$ ), confirmando su estabilidad energética. Su línea de tendencia horizontal indica que mantiene una eficiencia constante independientemente del escalamiento poblacional, lo que representa una característica única entre los *frameworks* evaluados.
- Inspyred presenta la correlación más baja ( $R^2 = 0,61$ ), mostrando un patrón ligeramente descendente con mayor variabilidad. Su tendencia sugiere una adaptación moderada al entorno servidor, con eficiencia óptima en configuraciones de población pequeña e intermedia.

Figura 80: comparación entre tendencias de *frameworks* del cálculo de  $\eta$  de Rosenbrock en el servidor



Fuente: elaboración propia

#### 4.3.12. Análisis de la eficiencia energética del portátil frente al servidor

A continuación, se comparan los resultados obtenidos en el portátil y en el servidor usando la misma métrica que la expuesta en el punto 4.3.10. Como se puede observar en las figuras 81 y 82, en todas las configuraciones y *frameworks*, el portátil multiplica por cuatro o, incluso, casi por seis, la eficiencia de la misma combinación en el servidor. Por ejemplo, para la configuración  $N = 2^{10}$  y  $p_c = 0,8$ :

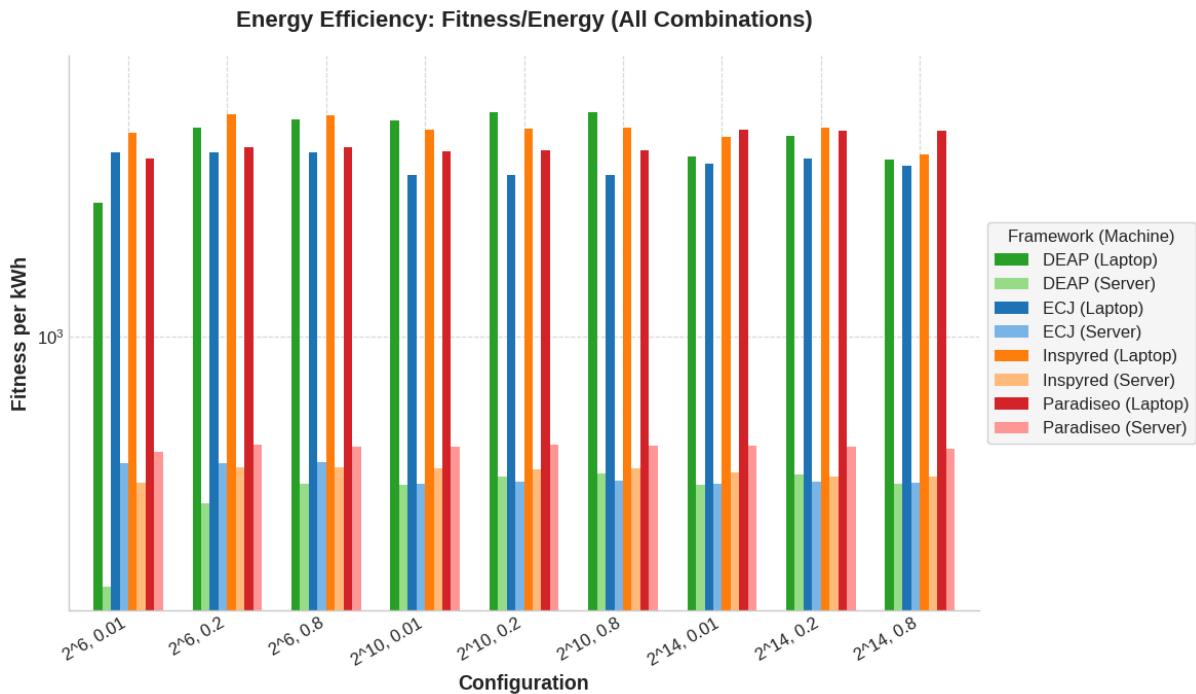
Tabla 9: Ejemplo de comparación de  $\eta$  de Rosenbrock para  $N = 2^{10}$  y  $p_c = 0,8$  en portátil y servidor

<b>Framework</b>	<b><math>\eta</math> (portátil)</b>	<b><math>\eta</math> (servidor)</b>	<b>Factor multiplicativo</b>
DEAP	2.850	530	5,38
ParadisEO	2.650	610	4,38
Inspyred	2.650	540	4,91
ECJ	2.120	520	4,08

Fuente: elaboración propia

Este salto refleja cómo la sobrecarga de gestión de paralelismo masivo en el servidor penaliza drásticamente la rentabilidad energética, pese a su mayor capacidad de procesamiento y recursos disponibles. La arquitectura del servidor, diseñada para maximizar el rendimiento computacional, introduce overhead energético significativo que no se traduce directamente en mejoras de eficiencia *fitness*/energía.

Figura 81: comparación entre *frameworks* del cálculo de  $\eta$  de Rosenbrock en portátil y servidor



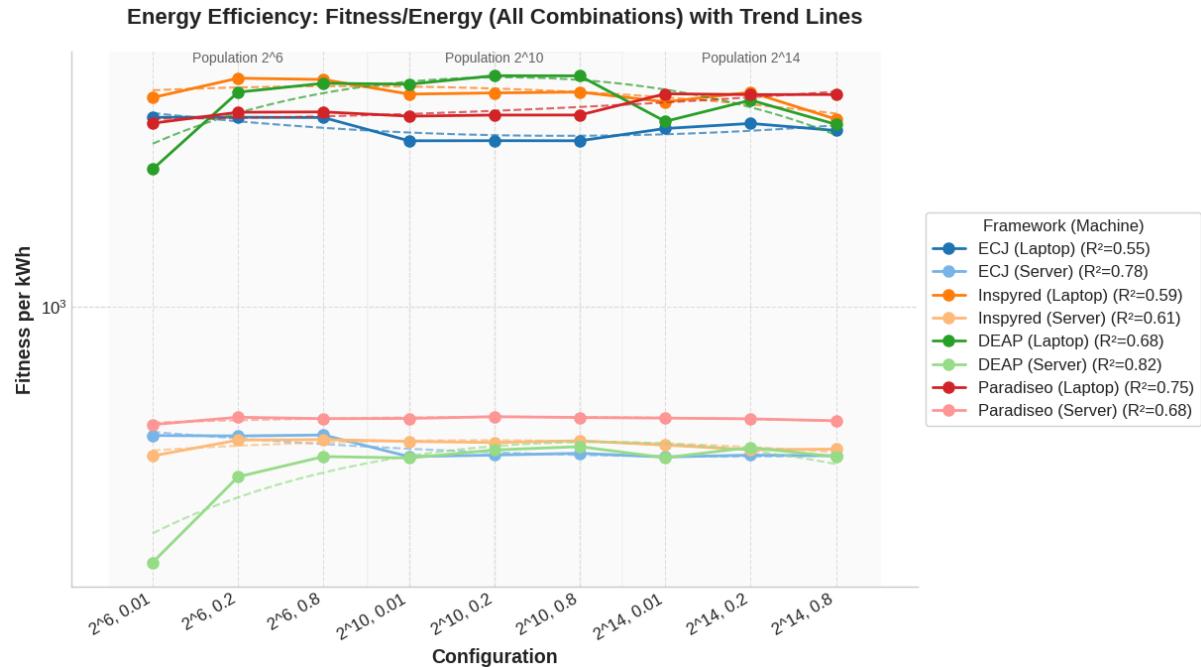
Fuente: elaboración propia

Las líneas de tendencia polinomiales revelan patrones diferenciados entre plataformas:

- En el portátil, los *frameworks* muestran correlaciones moderadas a fuertes ( $R^2$  entre 0,55 y 0,75), con Paradiseo exhibiendo la tendencia más estable ( $R^2 = 0,75$ ) y un comportamiento ascendente hacia poblaciones de mayor tamaño. DEAP presenta la correlación más pronunciada ( $R^2 = 0,68$ ) con un óptimo claro en poblaciones intermedias, mientras que ECJ e Inspyred muestran tendencias ligeramente decrecientes.

- En el servidor, las correlaciones se intensifican, con DEAP alcanzando la correlación más fuerte ( $R^2 = 0,82$ ) y un patrón dramáticamente ascendente que sugiere optimización para poblaciones grandes. ECJ mantiene una tendencia decreciente más pronunciada ( $R^2 = 0,78$ ), mientras que Paradiseo exhibe estabilidad excepcional ( $R^2 = 0,68$ ) con eficiencia prácticamente constante.

Figura 82: comparación entre tendencias de *frameworks* del cálculo de  $\eta$  de Rosenbrock en portátil y servidor



Fuente: elaboración propia

Además se puede observar, en primer lugar, que DEAP experimenta la transformación más fuerte entre plataformas: domina en portátil con eficiencia óptima en poblaciones intermedias, pero requiere poblaciones grandes para alcanzar eficiencia competitiva en servidor. Su factor multiplicativo de 5,38 refleja esta sensibilidad.

En segundo lugar, Paradiseo demuestra una adaptabilidad mayor, manteniendo una eficiencia competitiva en ambas plataformas, aunque con comportamientos distintos: ascendente en portátil y estable en servidor. Su menor factor multiplicativo (4,34) sugiere mejor optimización *cross-platform*.

ECJ muestra consistencia en su patrón decreciente en ambas plataformas, pero con eficiencia absoluta superior en portátil. Su factor multiplicativo de 4,08 es el más conservador del grupo.

Inspyred presenta estabilidad moderada en ambos entornos, con un factor multiplicativo intermedio (4,91) que refleja adaptación equilibrada pero sin optimizaciones específicas por plataforma.

#### 4.3.13. Distribución de eficiencia energética y de variación de *fitness*

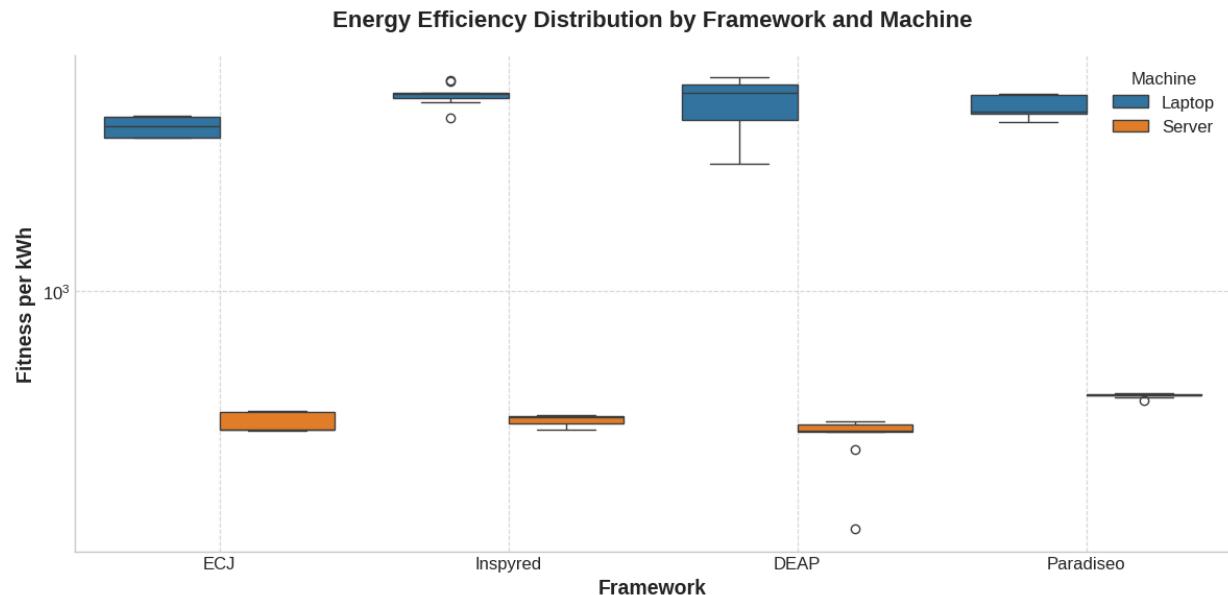
Para completar el análisis de eficiencia, se examina la distribución de las métricas de *fitness* por kWh y de variación de *fitness* por kWh. Los diagramas de caja que se pueden ver en las figuras 83 y 84 revelan patrones característicos de cada *framework* en ambas plataformas. Cabe así destacar, por *framework*:

- ECJ: En el portátil, presenta una caja muy estrecha comprendida entre 2.200 y 2.400 *fitness/kWh* con *whiskers* de apenas  $\pm 100$ , y su variación de *fitness* por kWh se concentra entre 650 y 750 *fitness/kWh* con dispersión mínima. Esta distribución indica un comportamiento altamente predecible independientemente de la configuración estudiada. En el servidor, la mediana de eficiencia desciende significativamente a aproximadamente 520 *fitness/kWh*, con *whiskers* aún más estrechos ( $\pm 30$ ), mientras que la variación/kWh se reduce drásticamente a un rango entre 120 y 140 *fitness/kWh*, también con dispersión mínima. Este bajo rango en ambas métricas refleja una implementación extremadamente estable pero poco sensible a optimizaciones específicas según el tamaño de población o probabilidad de cruce.
- Inspyred: En el portátil, muestra la caja más ancha en eficiencia energética, distribuyendo los valores entre 2.500 y 2.800 *fitness/kWh* con *whiskers* que se extienden desde 2.300 hasta 2.850, y su variación/kWh oscila ampliamente entre 700 y 900 *fitness/kWh* con *whiskers* que alcanzan desde 650 hasta 950. En el servidor, su eficiencia se comprime entre 510 y 540 *fitness/kWh* con *whiskers* más conservadores ( $\pm 30$ ), y la variación/kWh se reduce considerablemente al rango 140-160, confirmando que aunque mantiene cierta adaptabilidad, el entorno servidor limita su capacidad de optimización diferencial según configuración.
- DEAP: En el portátil, presenta la distribución más amplia, con eficiencia que se extiende desde 1.700 hasta 2.850 *fitness/kWh*, incluyendo *whiskers* que alcanzan desde 1.500 hasta 2.900, evidenciando la mayor sensibilidad a cambios configuracionales. Su variación/kWh se distribuye ampliamente entre 800 y 1.200 *fitness/kWh*. En el servidor, la eficiencia se concentra entre 480 y 520 *fitness/kWh* con algunos *outliers* que se extienden hasta 300 y 530, mientras que la variación/kWh se establece en un rango más estrecho de 180-220, mostrando capacidad de adaptación al paralelismo pero con comportamiento más conservador que en portátil.
- Paradiseo: En el portátil, es altamente predecible, con caja estrecha de 2.550 a 2.650 *fitness/kWh* y *whiskers* compactos de  $\pm 100$ , y variación/kWh concentrada entre 750 y 800 *fitness/kWh*. Esta distribución compacta indica implementación robusta con poca sensibilidad a variaciones configuracionales. En el servidor, mantiene su característica estabilidad variando mínimamente entre 590 y 610 *fitness/kWh* (*whiskers* de  $\pm 20$ ), siendo la implementación más homogénea, y variación/kWh entre 190 y 210, confirmando que Paradiseo logra el mejor balance entre eficiencia y predictibilidad.

En resumen, los patrones de distribución revelan que ECJ y Paradiseo priorizan estabilidad sobre optimización, mientras que DEAP e Inspyred muestran mayor sensibilidad configuracional. En portátil, esta variabilidad permite optimizaciones específicas, pero en servidor, la reducción generalizada de dispersión sugiere que las limitaciones comprimen las diferencias de comportamiento entre configuraciones.

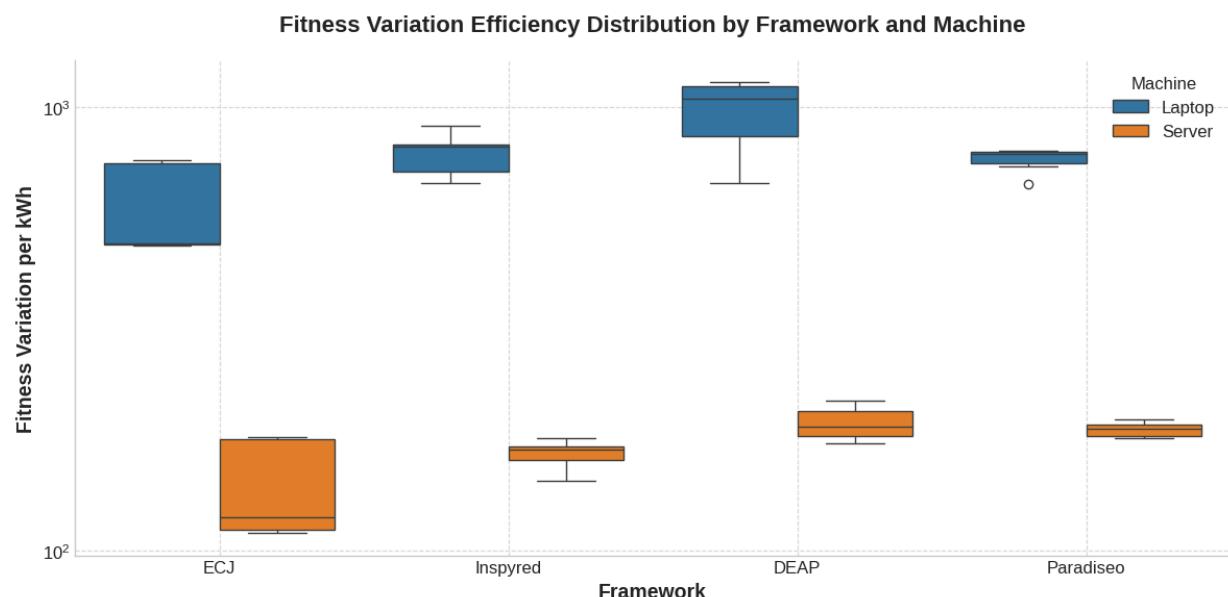
La consistencia de Paradiseo en ambas plataformas, combinada con su eficiencia superior en servidor, lo posiciona como la opción más robusta para implementaciones *cross-platform* donde la predictibilidad energética es prioritaria.

Figura 83: distribución de la eficiencia energética de Rosenbrock en portátil y servidor



Fuente: elaboración propia

Figura 84: distribución de la variación del *fitness* de Rosenbrock en portátil y servidor



Fuente: elaboración propia

## 4.4. Schwefel

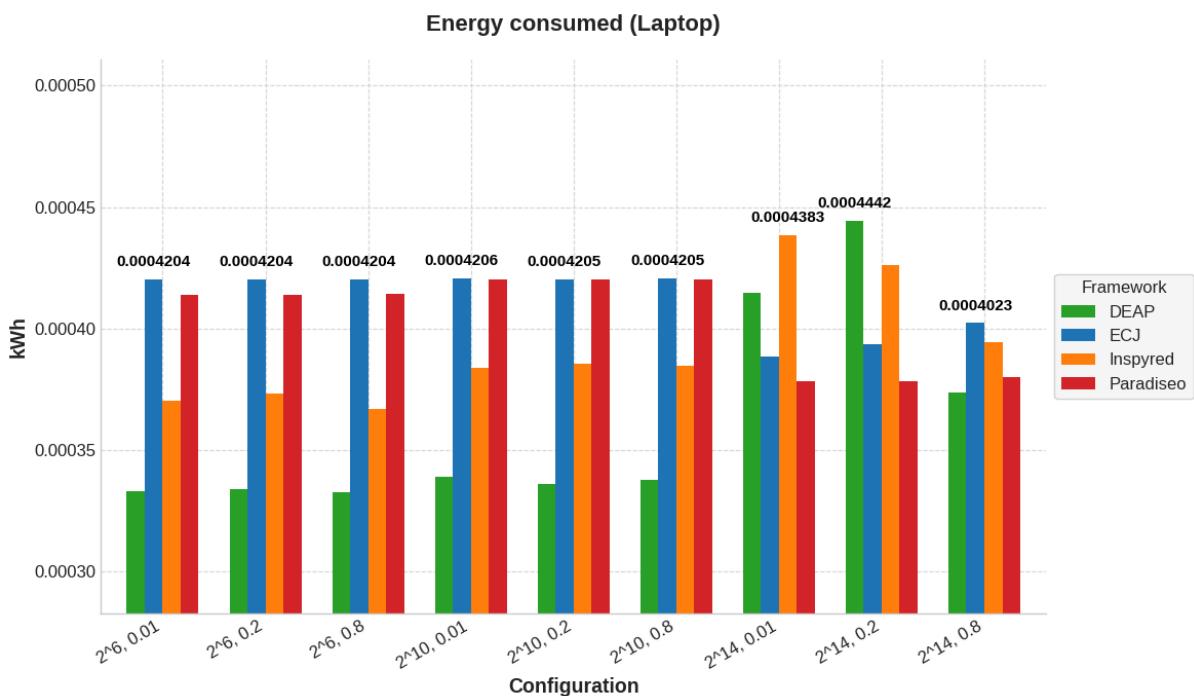
En esta subsección se repetirá la misma estructura de análisis del experimento Rosenbrock, pero aplicándolo al *benchmark* Schwefel.

### 4.4.1. Consumo energético en el portátil

Como se puede observar en la figura 85, la energía consumida presenta variaciones significativas entre *frameworks*:

- DEAP oscila entre, aproximadamente, 0,000333 kWh ( $N = 2^6, p_c = 0,01$ ) y 0,000442 kWh ( $N = 2^{14}, p_c = 0,2$ ), mostrando el comportamiento más irregular pero alcanzando el pico máximo de consumo en configuraciones de población grande.
- ECJ mantiene valores relativamente estables entre 0,000389 kWh ( $N = 2^{14}, p_c = 0,2$ ) y 0,000420 kWh ( $N = 2^6, p_c = 0,01$ ), con una consistencia notable en poblaciones pequeñas e intermedias, pero experimentando una reducción significativa en  $N = 2^{14}$ .
- Inspyred registra un rango de 0,000367 kWh ( $N = 2^6, p_c = 0,01$ ) hasta 0,000438 kWh ( $N = 2^{14}, p_c = 0,01$ ), posicionándose en valores intermedios con un incremento notable hacia poblaciones grandes.
- ParadisEO presenta valores entre 0,000377 kWh ( $N = 2^{14}, p_c = 0,2$ ) y 0,000421 kWh ( $N = 2^6, p_c = 0,01$ ), manteniendo estabilidad en configuraciones pequeñas e intermedias, pero con descenso pronunciado en poblaciones grandes.

Figura 85: energía consumida por el portátil en las ejecuciones de Schwefel

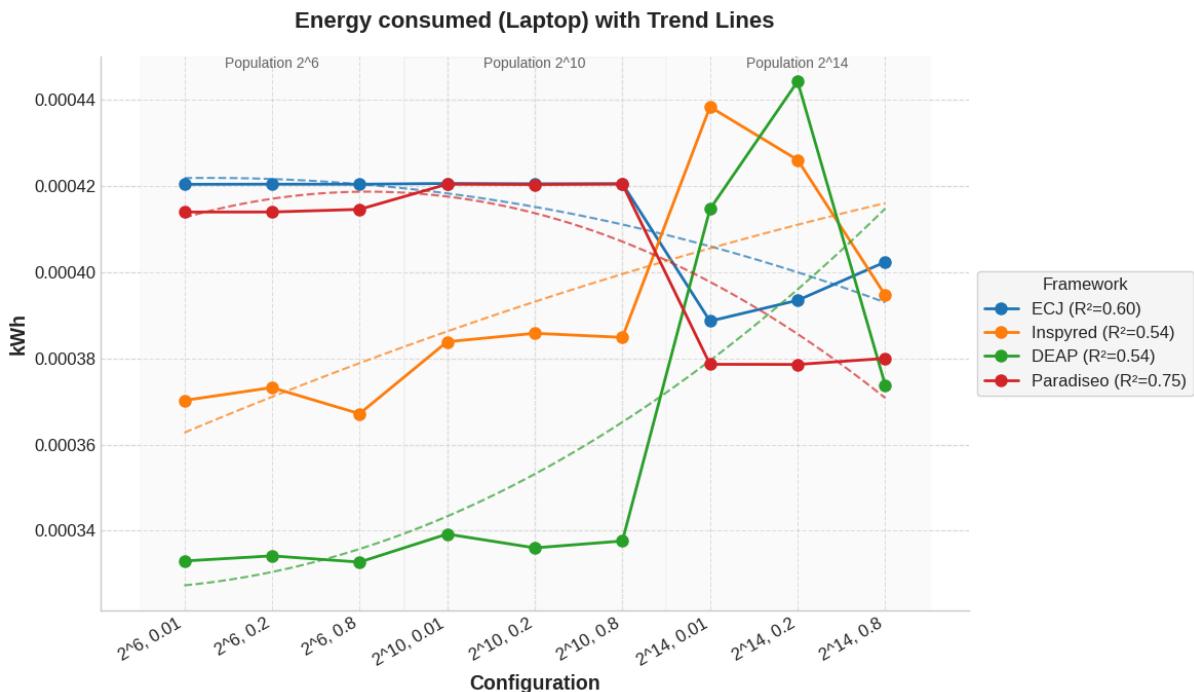


Fuente: elaboración propia

Al atender a las líneas de tendencia polinómicas visibles en la figura 86 se revelan comportamientos energéticos diferenciados según el tamaño de la población:

- ECJ ( $R^2 = 0,6$ ) evidencia una tendencia descendente suave, sugiriendo una mejor optimización energética conforme aumenta el tamaño poblacional.
- Inspyred ( $R^2 = 0,54$ ) muestra una curvatura convexa pronunciada, con un mínimo en poblaciones intermedias ( $N = 2^{10}$ ) y un incremento exponencial hacia  $N = 2^{14}$ , indicando posibles sobrecostes de gestión en configuraciones extremas.
- DEAP ( $R^2 = 0,54$ ) presenta la tendencia más pronunciada, con un crecimiento exponencial que alcanza su máximo en  $N = 2^{14}$ ,  $p_c = 0,2$ , seguido de una caída pronunciada, sugiriendo umbrales críticos de eficiencia energética.
- ParadisEO ( $R^2 = 0,75$ ) registra el mejor ajuste lineal con tendencia ligeramente descendente, indicando un comportamiento energéticamente proporcional y predecible, con eficiencia creciente al aumentar la población.

Figura 86: tendencia de la energía consumida por el portátil en las ejecuciones de Schwefel



Fuente: elaboración propia

Para poner cifras en contexto:

- Poblaciones pequeñas ( $N = 2^6$ ): consumo aproximado de 0,000333 kWh (DEAP), 0,000367 kWh (Inspyred), 0,000420 kWh (ECJ) y 0,000415 kWh (ParadisEO).
- Poblaciones intermedias ( $N = 2^{10}$ ): DEAP incrementa un 2,1 %, Inspyred aumenta un 4,6 %, ECJ se mantiene estable (-0,1 %) y ParadisEO sube marginalmente un 0,5 %.

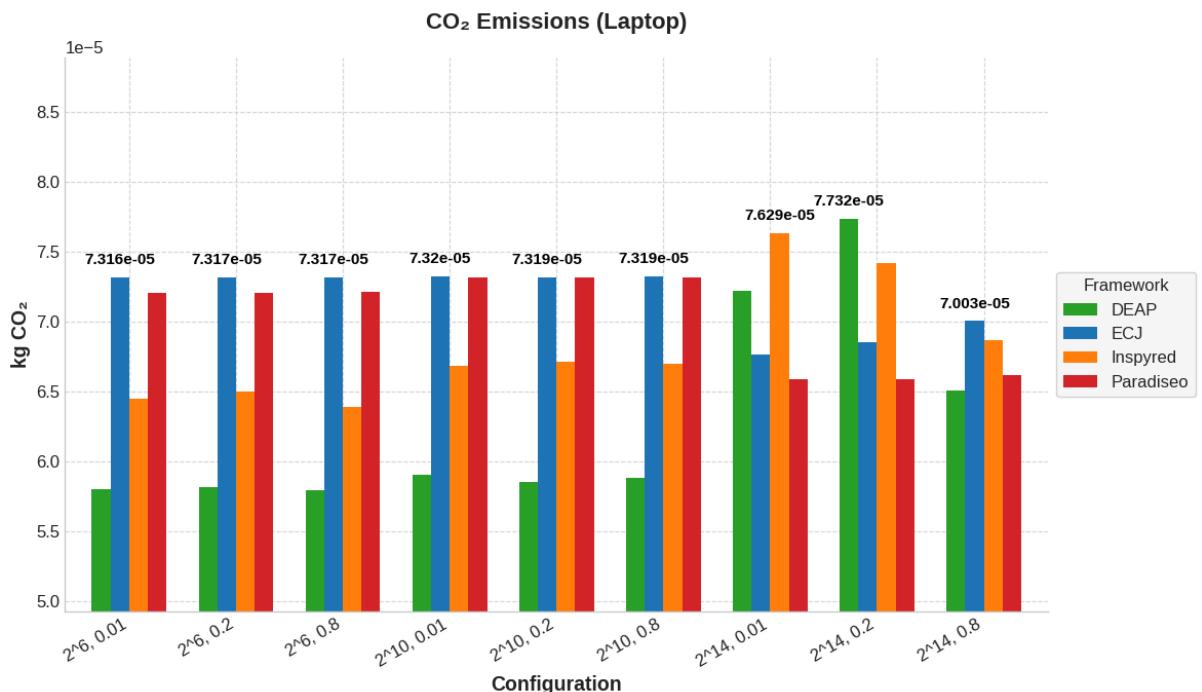
- Poblaciones grandes ( $N = 2^{14}$ ): DEAP experimenta el mayor incremento (+32,7 % respecto a poblaciones pequeñas), Inspyred aumenta un 19,4 %, mientras ECJ y ParadisEO reducen su consumo un 7,4 % y 9,2 % respectivamente.

En síntesis, el portátil demuestra que ParadisEO ofrece el comportamiento más predecible y energéticamente eficiente ( $R^2 = 0,75$ ), mientras que DEAP, pese a ser el más eficiente en configuraciones pequeñas, presenta la mayor variabilidad energética en poblaciones grandes. ECJ muestra una mejora progresiva de eficiencia con el aumento poblacional, contrario al patrón típico de sobrecarga computacional.

#### 4.4.2. Emisiones totales de CO<sub>2</sub> en el portátil

Según se puede observar en la figura 87, las emisiones de CO<sub>2</sub> reproducen fielmente el patrón de consumo energético visto en la sección anterior.

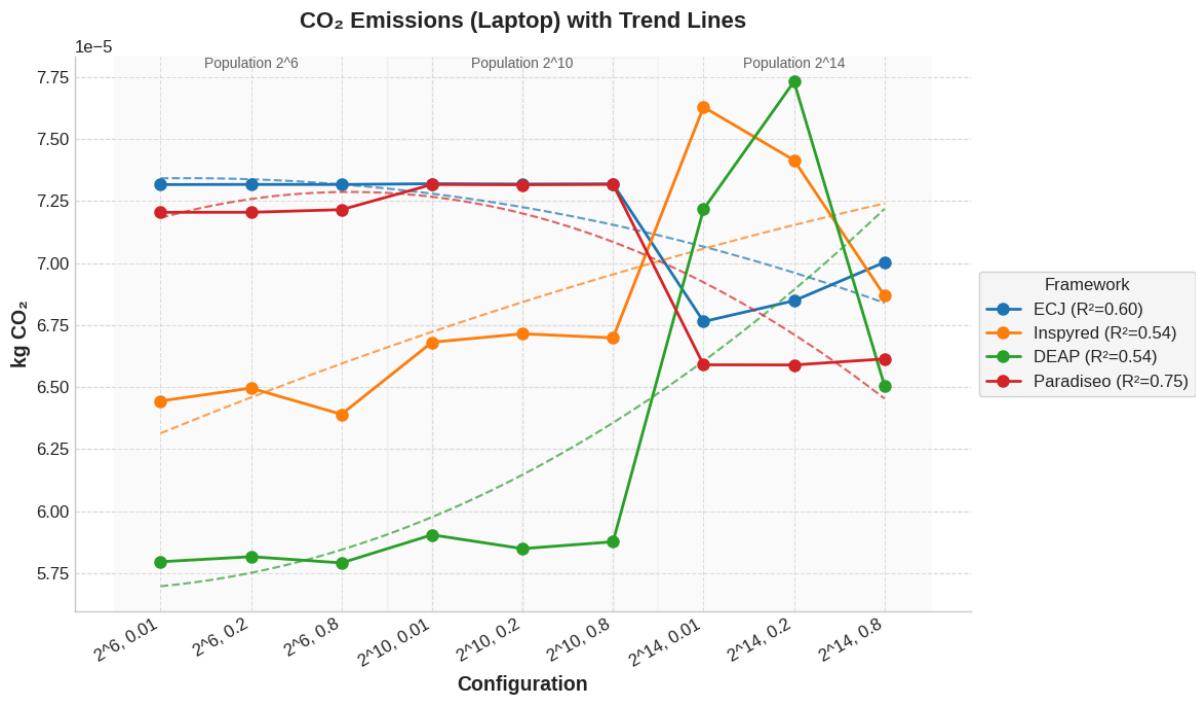
Figura 87: emisiones totales de CO<sub>2</sub> por el portátil en las ejecuciones de Schwefel



Fuente: elaboración propia

En primer lugar, DEAP registra las emisiones más bajas del conjunto, oscilando entre 0,0000578 kg CO<sub>2</sub> ( $N = 2^6$ ,  $p_c = 0,01$ ) y 0,0000773 kg CO<sub>2</sub> ( $N = 2^{14}$ ,  $p_c = 0,2$ ). Su comportamiento mantiene valores relativamente estables en poblaciones pequeñas e intermedias (incremento marginal del 1,9 % de  $N = 2^6$  a  $N = 2^{10}$ ), pero muestra un incremento pronunciado del 33,8 % hacia poblaciones grandes. La tendencia ( $R^2 = 0,54$ ) evidencia una curvatura convexa extrema, alcanzando un máximo crítico en  $N = 2^{14}$ ,  $p_c = 0,2$  seguido de una caída abrupta, sugiriendo umbrales críticos de eficiencia ambiental donde los costes de gestión poblacional superan los beneficios de paralelización. Esta curvatura respalda la amortización del overhead de inicialización que se distribuye eficientemente hasta que las poblaciones muy grandes generan sobrecostos no lineales.

Figura 88: tendencia de las emisiones totales de CO<sub>2</sub> por el portátil en las ejecuciones de Schwefel



Fuente: elaboración propia

En segundo lugar, Inspyred ocupa la segunda posición en términos de sostenibilidad ambiental, con emisiones que van de 0,0000644 kg CO<sub>2</sub> ( $N = 2^6, p_c = 0,01$ ) hasta 0,0000763 kg CO<sub>2</sub> ( $N = 2^{14}, p_c = 0,01$ ). Presenta un comportamiento intermedio con incrementos graduales del 4,5 % hacia poblaciones intermedias y del 18,4 % hacia poblaciones grandes, mostrando mayor estabilidad que DEAP en configuraciones extremas. Su tendencia ( $R^2 = 0,54$ ) muestra una curvatura convexa menos pronunciada que DEAP, con un comportamiento más suave pero ascendente. El pico en  $N = 2^{14}, p_c = 0,01$  indica cierta sensibilidad a la combinación específica de parámetros poblacionales y probabilísticos, sugiriendo una gestión no lineal de accesos a memoria y optimizaciones del intérprete Python.

En tercer lugar, ECJ mantiene emisiones consistentes entre 0,0000677 kg CO<sub>2</sub> ( $N = 2^{14}, p_c = 0,2$ ) y 0,0000732 kg CO<sub>2</sub> ( $N = 2^6, p_c = 0,01$ ), con cierta uniformidad en poblaciones pequeñas e intermedias (variación prácticamente nula) pero experimentando una reducción significativa del 7,5 % en poblaciones grandes. Su tendencia ( $R^2 = 0,6$ ) presenta un comportamiento descendente suave, indicando una mejora progresiva de la eficiencia ambiental conforme aumenta el tamaño poblacional.

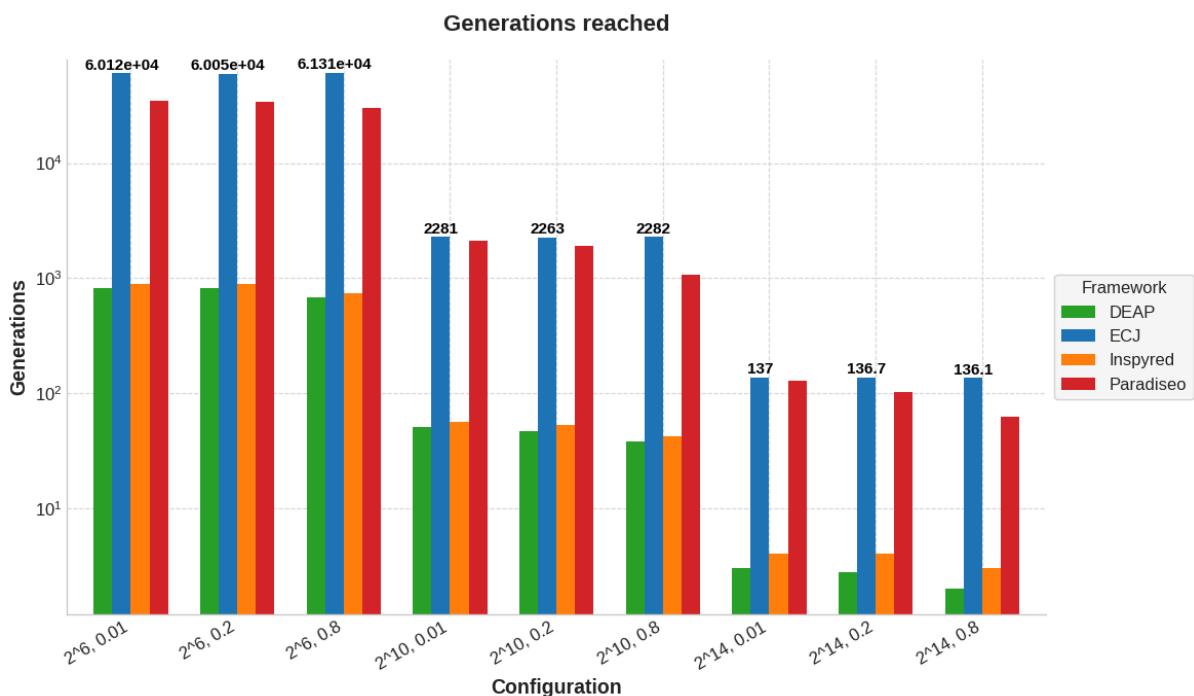
Por último, ParadisEO registra las emisiones más elevadas en la mayoría de configuraciones, oscilando entre 0,0000656 kg CO<sub>2</sub> ( $N = 2^{14}, p_c = 0,2$ ) y 0,0000732 kg CO<sub>2</sub> ( $N = 2^6, p_c = 0,01$ ). Mantiene valores altos y estables en poblaciones pequeñas e intermedias, con un descenso pronunciado del 10,4 % hacia poblaciones grandes. Su tendencia ( $R^2 = 0,75$ ) registra el mejor ajuste lineal con pendiente ligeramente descendente, confirmando su predictibilidad en términos de huella de carbono y comportamiento energéticamente proporcional.

En conjunto, los resultados revelan diferencias de hasta un 34 % en emisiones de CO<sub>2</sub> entre configuraciones, con DEAP e Inspyred constituyendo las alternativas más sostenibles para poblaciones pequeñas e intermedias, ECJ demostrando el mejor comportamiento ambiental en poblaciones grandes, y ParadisEO manteniendo la mayor huella de carbono absoluta pese a su comportamiento más predecible.

#### 4.4.3. Evolución del *fitness* en el portátil

Para representar el comportamiento evolutivo y la calidad de las soluciones se han propuesto cuatro métricas a evaluar: *fitness* inicial, variación del *fitness*, *fitness* máximo alcanzado y número de generaciones alcanzadas.

Figura 89: número máximo de generaciones alcanzadas de Schwefel



Fuente: elaboración propia

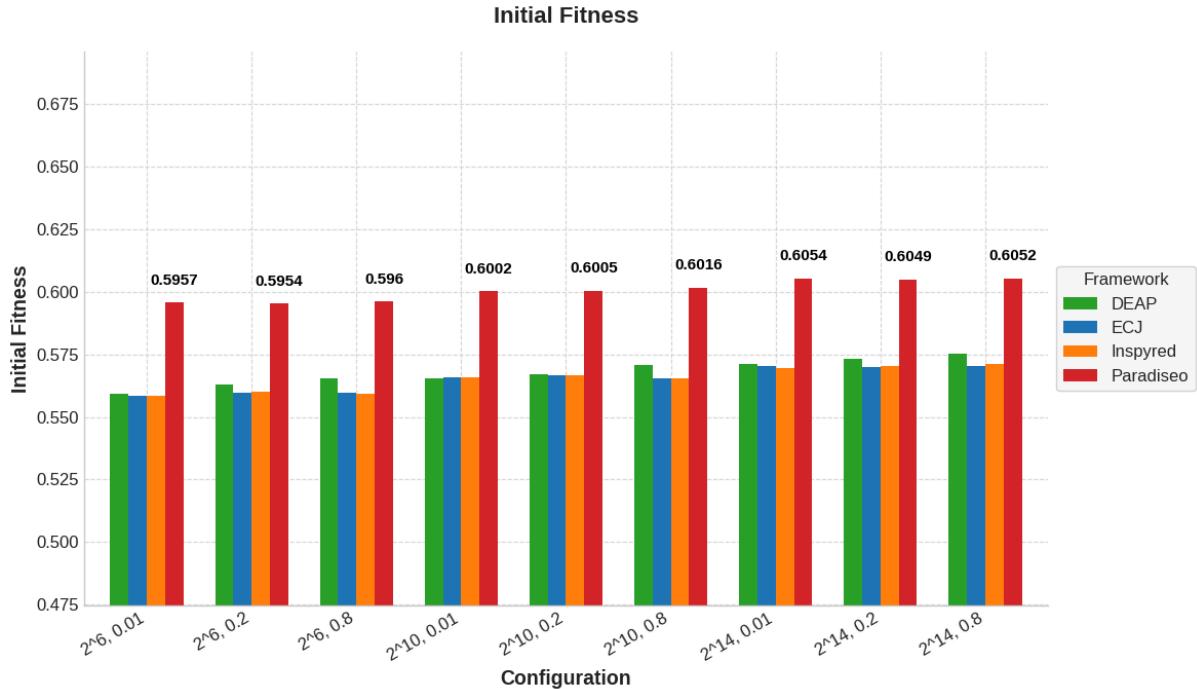
Los resultados visibles en la figura 89 muestran tres órdenes de magnitud de diferencia en el número de generaciones completadas antes del límite temporal de dos minutos. ECJ demuestra la mayor capacidad iterativa, completando aproximadamente 60.120 generaciones en  $N = 2^6$ ,  $p_c = 0,01$ , reduciéndose progresivamente a 2.281 generaciones en  $N = 2^{10}$ ,  $p_c = 0,01$  y finalizando en 137 generaciones para  $N = 2^{14}$ ,  $p_c = 0,01$ . Esta reducción sistemática confirma la relación inversa entre tamaño poblacional y capacidad iterativa.

DEAP presenta el comportamiento más conservador, oscilando entre 850 generaciones ( $N = 2^6$ ,  $p_c = 0,01$ ) y apenas 3 generaciones ( $N = 2^{14}$ ,  $p_c = 0,2$ ), evidenciando un coste computacional elevado por iteración que limita severamente su capacidad de exploración temporal.

Inspyred mantiene un rango intermedio, desde 900 generaciones ( $N = 2^6$ ,  $p_c = 0,01$ ) hasta 4 generaciones ( $N = 2^{14}$ ,  $p_c = 0,2$ ), siguiendo un patrón similar a DEAP pero con ligera ventaja en configuraciones pequeñas.

ParadisEO registra valores intermedios entre ECJ y los *frameworks* Python, completando desde 25.000 generaciones ( $N = 2^6$ ,  $p_c = 0,01$ ) hasta 65 generaciones ( $N = 2^{14}$ ,  $p_c = 0,8$ ), manteniendo una capacidad iterativa superior a DEAP e Inspyred en todos los escenarios evaluados.

Figura 90: *fitness* inicial promedio de Schwefel



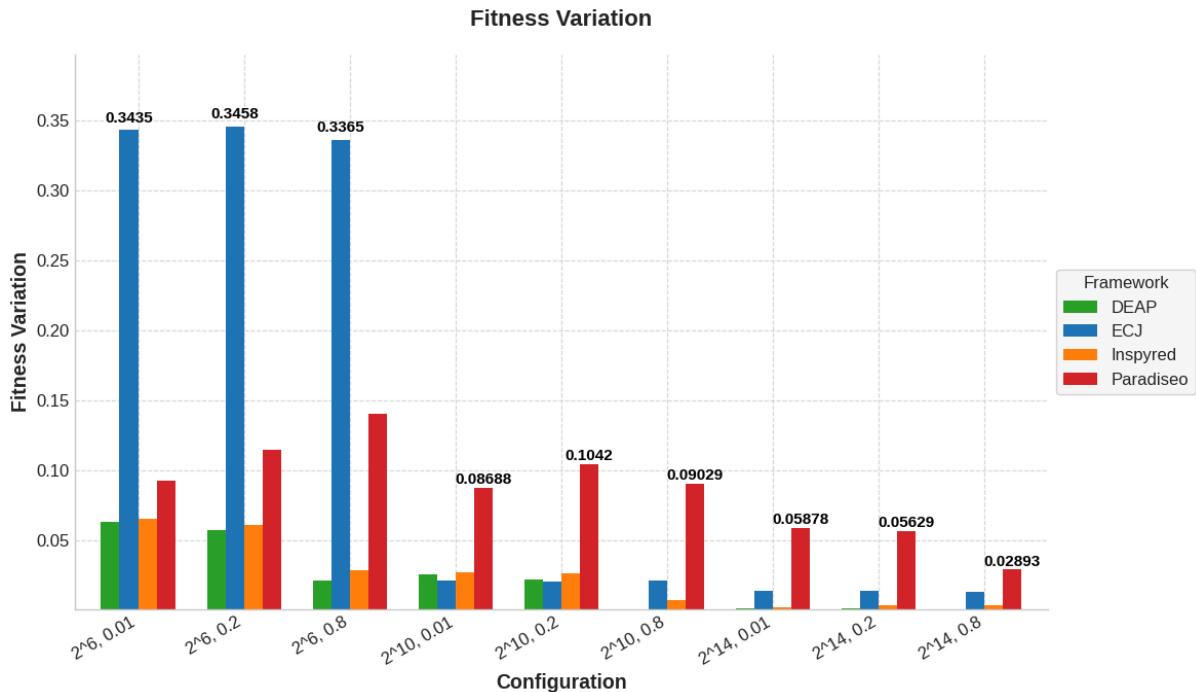
Fuente: elaboración propia

Los valores iniciales mostrados en la figura 90 revelan diferentes estrategias de inicialización poblacional. ParadisEO presenta los valores iniciales más altos y estables, oscilando entre 0,5957 ( $N = 2^6$ ,  $p_c = 0,01$ ) y 0,6052 ( $N = 2^{14}$ ,  $p_c = 0,8$ ), sugiriendo una inicialización más conservadora pero de mayor calidad. ECJ e Inspyred muestran comportamientos similares, con valores entre 0,5560 y 0,5720, indicando estrategias de inicialización comparable que priorizan la diversidad sobre la calidad inicial. DEAP registra los valores iniciales más bajos, variando de 0,5560 ( $N = 2^6$ ,  $p_c = 0,01$ ) a 0,5750 ( $N = 2^{14}$ ,  $p_c = 0,8$ ), reflejando una inicialización más exploratoria que sacrifica calidad inicial por diversidad poblacional.

La mejora evolutiva reflejada en la figura 91 muestra diferencias significativas entre *frameworks*. ECJ exhibe las mayores ganancias evolutivas, alcanzando picos de 0,3458 ( $N = 2^6$ ,  $p_c = 0,2$ ) y manteniéndose consistentemente por encima de 0,33 en poblaciones pequeñas, descendiendo gradualmente hasta 0,02893 ( $N = 2^{14}$ ,  $p_c = 0,8$ ). Esta capacidad de mejora refleja la eficiencia de sus operadores evolutivos y su alta capacidad iterativa. ParadisEO presenta ganancias moderadas pero estables, oscilando entre 0,02893 ( $N = 2^{14}$ ,  $p_c = 0,8$ ) y

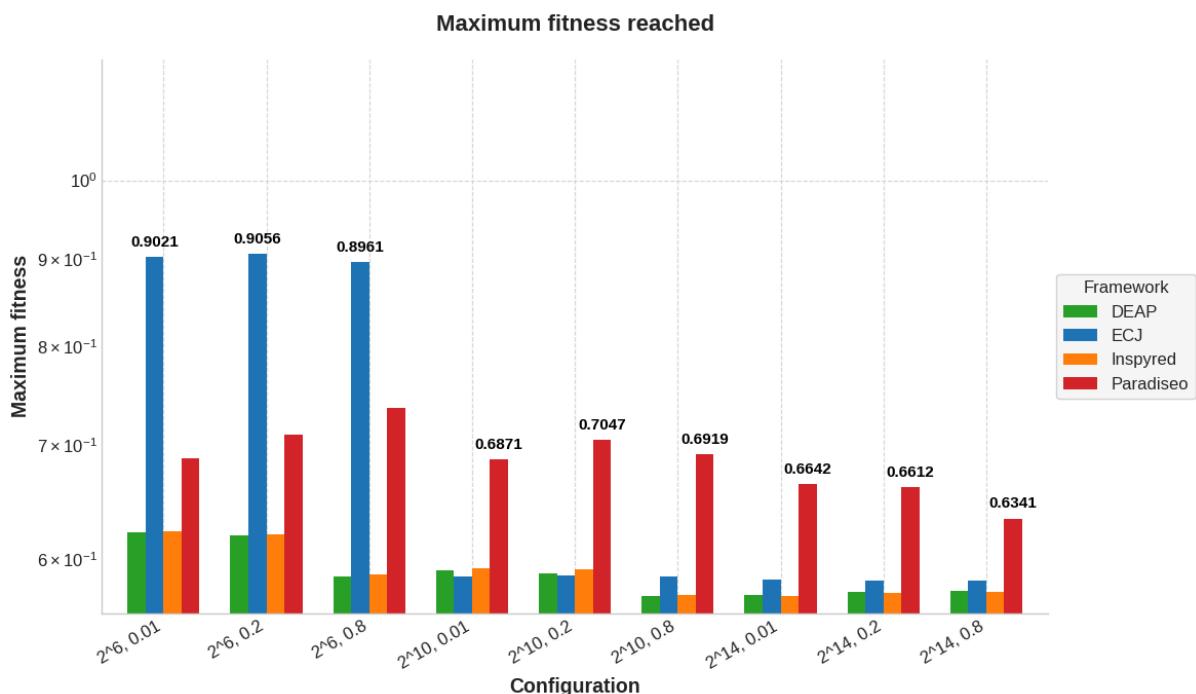
0,1042 ( $N = 2^{10}$ ,  $p_c = 0,2$ ), con tendencia descendente al aumentar el tamaño poblacional. DEAP e Inspyred registran las menores variaciones, manteniéndose generalmente por debajo de 0,07, lo que sugiere limitaciones en su capacidad de convergencia debido al bajo número de generaciones alcanzadas o ineficiencias en sus operadores evolutivos.

Figura 91: variación promedio del *fitness* de Schwefel



Fuente: elaboración propia

Figura 92: *fitness* máximo promedio alcanzado de Schwefel



Fuente: elaboración propia

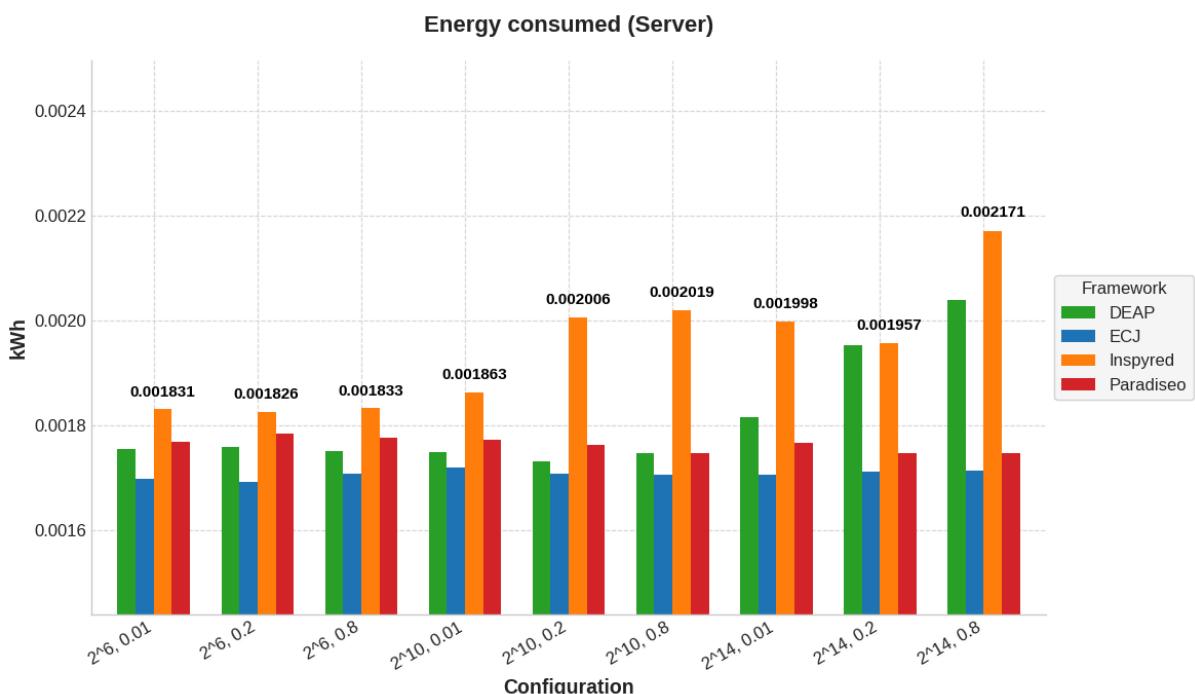
La calidad final de las soluciones, expuesta en la figura 92, revela la eficacia global de cada *framework*. ECJ alcanza los mejores resultados absolutos, superando 0,90 en poblaciones pequeñas ( $N = 2^6$ ) con máximos de 0,9021 ( $p_c = 0,01$ ), aunque experimenta degradación al aumentar el tamaño poblacional, descendiendo hasta 0,5780 ( $N = 2^{14}$ ,  $p_c = 0,8$ ). ParadisEO demuestra mayor estabilidad en poblaciones grandes, manteniendo valores entre 0,6341 ( $N = 2^{14}$ ,  $p_c = 0,8$ ) y 0,7411 ( $N = 2^6$ ,  $p_c = 0,8$ ), con mejor consistencia que ECJ en configuraciones extremas. DEAP e Inspyred presentan rendimientos similares y más conservadores, oscilando entre 0,5650 y 0,6250, reflejando las limitaciones impuestas por su baja capacidad iterativa y menor eficiencia de convergencia.

En conjunto, los resultados confirman que ECJ optimiza la relación iteraciones-calidad en poblaciones pequeñas, ParadisEO ofrece el mejor equilibrio entre estabilidad y rendimiento en todas las configuraciones, mientras que DEAP e Inspyred, pese a su facilidad de uso, presentan limitaciones significativas en capacidad iterativa y calidad de convergencia que comprometen su eficacia en optimización continua.

#### 4.4.4. Consumo energético en servidor

Como se observa en las figuras 93 y 94, el orden jerárquico de consumo energético se mantiene relativamente estable a lo largo de las configuraciones, con sólo intercambios menores entre algunos *frameworks*, lo que permite una comparación directa y consistente de su eficiencia energética.

Figura 93: energía consumida por el servidor en las ejecuciones de Schwefel



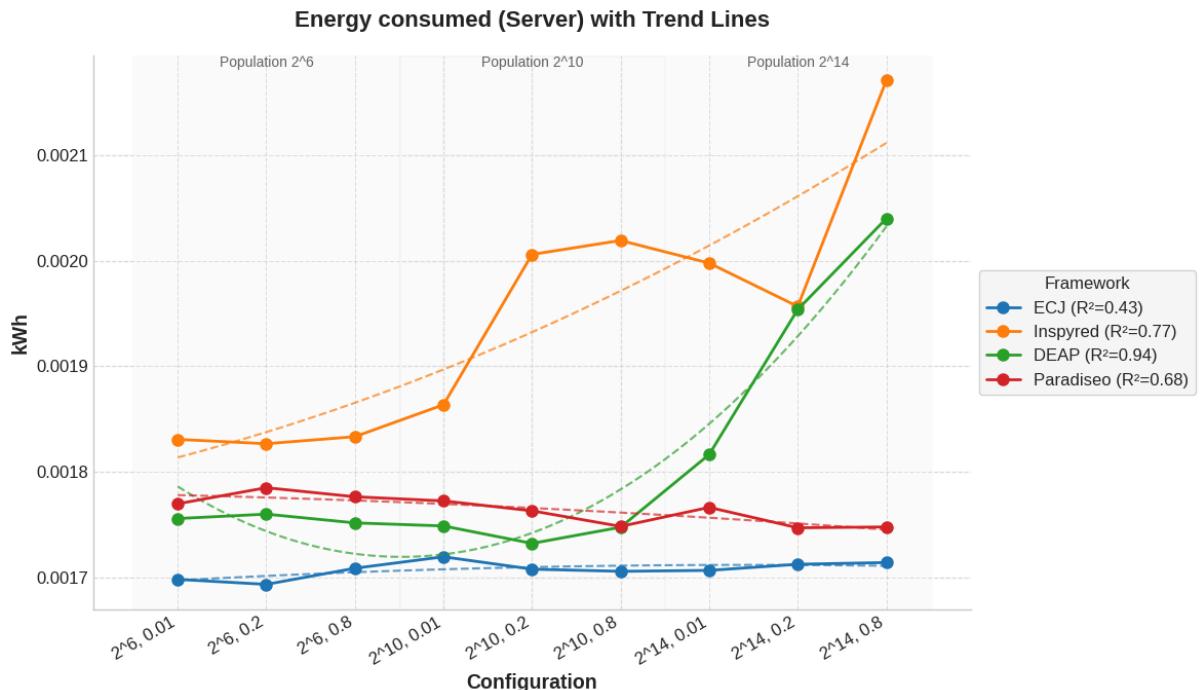
Fuente: elaboración propia

Inspyred aparece como el *framework* más consumidor de energía de forma sistemática, con valores que oscilan desde 0,001831 kWh ( $N = 2^6$ ,  $p_c = 0,01$ ) hasta alcanzar

su pico máximo de 0,002171 kWh ( $N = 2^{14}$ ,  $p_c = 0,8$ ), representando este último el valor más alto registrado en todo el conjunto de pruebas. Su comportamiento muestra una tendencia claramente ascendente conforme aumenta el tamaño poblacional.

DEAP se posiciona, de media, como el segundo *framework* en mayor consumo energético, iniciando en 0,001762 kWh ( $N = 2^6$ ,  $p_c = 0,01$ ) y culminando en 0,002049 kWh ( $N = 2^{14}$ ,  $p_c = 0,8$ ). Su comportamiento es particularmente interesante, ya que muestra una notable mejora en eficiencia durante las poblaciones intermedias antes de experimentar un incremento pronunciado hacia las configuraciones de mayor población.

Figura 94: tendencia de la energía consumida por el servidor en las ejecuciones de Schwefel



Fuente: elaboración propia

Paradiseo mantiene un perfil energético intermedio, con registros que van desde 0,001772 kWh ( $N = 2^6$ ,  $p_c = 0,01$ ) hasta 0,001751 kWh ( $N = 2^{14}$ ,  $p_c = 0,8$ ). Su comportamiento se mantiene relativamente estable a lo largo de todas las configuraciones, ocupando consistentemente la tercera posición en eficiencia energética.

ECJ se distingue como el *framework* más eficiente energéticamente, con valores consistentemente bajos que oscilan entre, aproximadamente, 0,001698 kWh y 0,001720 kWh a lo largo de todas las configuraciones evaluadas. Mantiene el consumo más bajo de forma consistente, posicionándose como la opción más eficiente en términos energéticos.

La diferencia absoluta entre el *framework* más eficiente (ECJ) y el más costoso (Inspyred en  $N = 2^{14}$ ,  $p_c = 0,8$ ) alcanza aproximadamente 0,000450 kWh, lo que representa cerca del 26 % de margen relativo, una diferencia significativa que se mantiene o incluso se amplifica en el entorno de servidor.

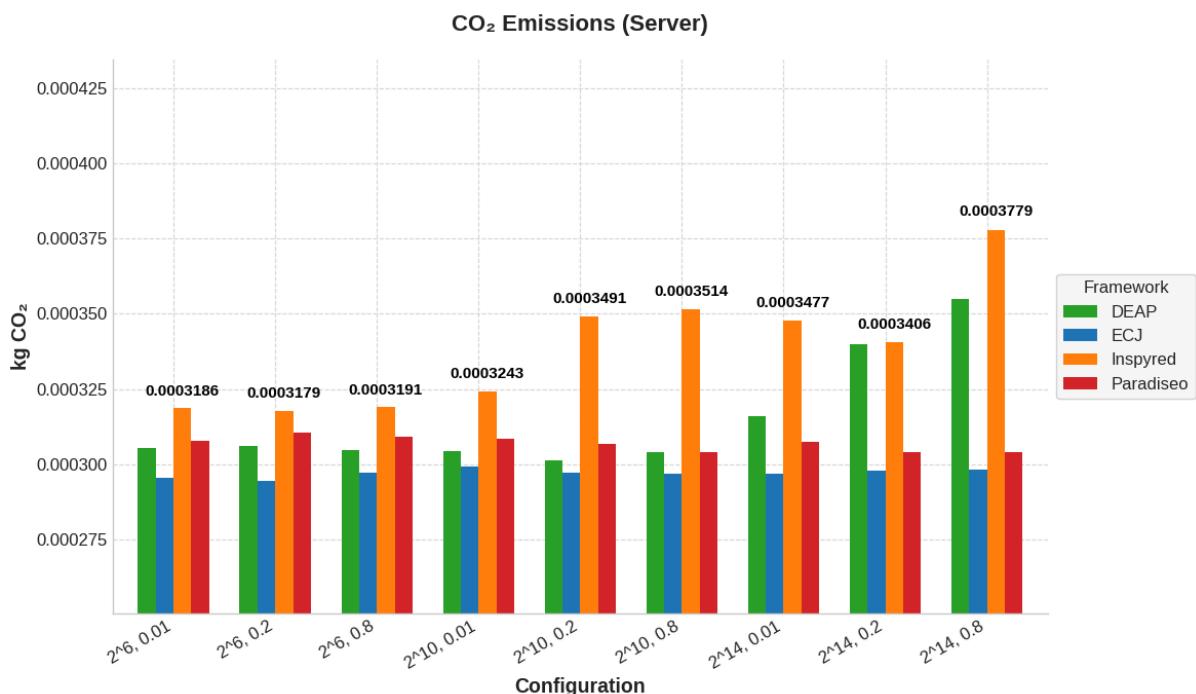
Además, la figura de líneas con tendencia polinomial muestra con claridad las dinámicas de escalado energético:

- ECJ ( $R^2 = 0,43$ ) exhibe la correlación más débil con la tendencia poblacional y mantiene el consumo más bajo, sugiriendo un comportamiento energético excepcionalmente estable e independiente del tamaño de la población. Esto evidencia una arquitectura altamente optimizada que mantiene su eficiencia superior a través de diferentes escalas computacionales.
- Paradiseo ( $R^2 = 0,68$ ) mantiene una línea de tendencia relativamente estable con variaciones moderadas, reflejando un perfil energético consistente en la zona intermedia de consumo.
- DEAP ( $R^2 = 0,94$ ) presenta la correlación más fuerte con una curva característica en forma de U invertida, donde la eficiencia mejora en poblaciones intermedias antes de deteriorarse marcadamente con poblaciones muy grandes.
- Inspyred ( $R^2 = 0,77$ ) muestra una tendencia ascendente consistente y pronunciada, confirmando su posición como el i menos eficiente energéticamente.

En conclusión, ECJ emerge claramente como la opción más eficiente energéticamente, seguido por Paradiseo, mientras que DEAP e Inspyred presentan costos energéticos considerablemente superiores, especialmente en configuraciones de alta población.

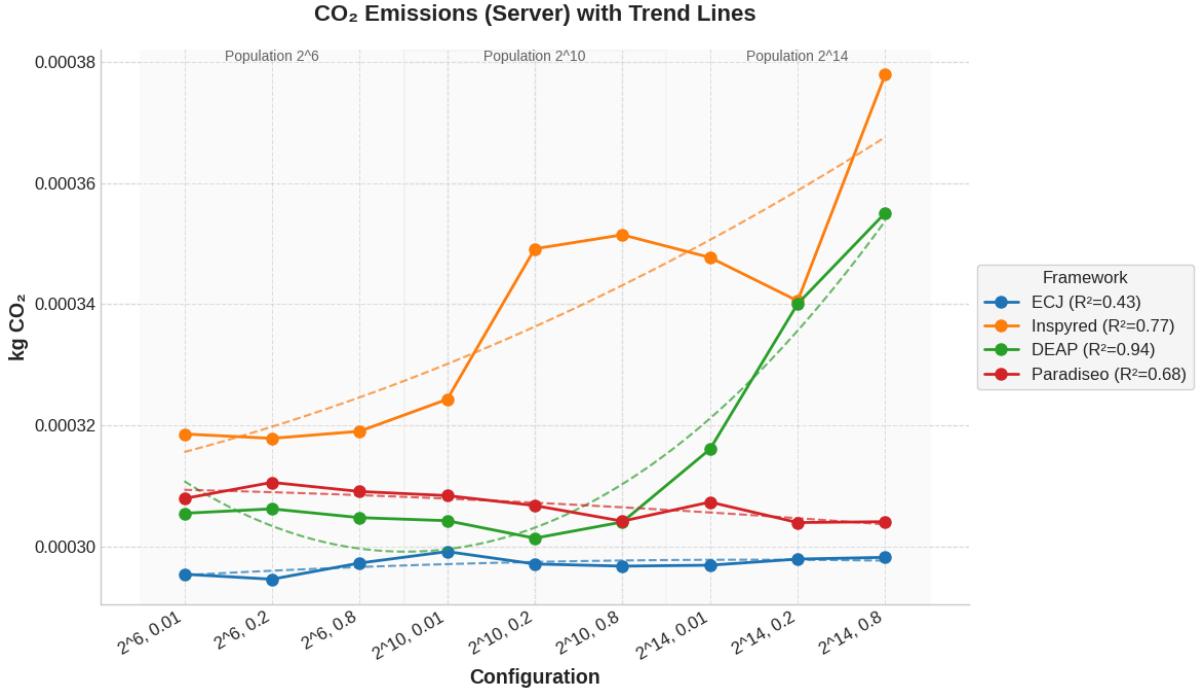
#### 4.4.5. Emisiones totales de CO<sub>2</sub> en servidor

Figura 95: emisiones totales de CO<sub>2</sub> por el servidor en las ejecuciones de Schwefel



Fuente: elaboración propia

Figura 96: tendencia de las emisiones totales de CO<sub>2</sub> por el servidor en las ejecuciones de Schwefel



Fuente: elaboración propia

En las figuras 95 y 96 se expone el orden jerárquico de emisiones de CO<sub>2</sub> en el servidor. En las nueve configuraciones ejecutadas ( $N \in \{2^6, 2^{10}, 2^{14}\}$  y probabilidad de cruce  $p_c \in \{0.01, 0.2, 0.8\}$ ), se observa que:

- Inspyred se posiciona como el *framework* más emisivo en todas las configuraciones evaluadas, con emisiones que oscilan desde 0,0003179 kg CO<sub>2</sub> ( $N = 2^6, p_c = 0,2$ ) hasta alcanzar su pico máximo de 0,0003779 kg CO<sub>2</sub> ( $N = 2^{14}, p_c = 0,8$ ), representando este último el valor más alto registrado en todo el conjunto de pruebas. Su tendencia polinomial ( $R^2 = 0,77$ ) confirma una curva ascendente pronunciada que refleja el creciente overhead interpretativo de Python conforme aumenta la complejidad poblacional, penalizando progresivamente tanto en consumo energético como en emisiones de carbono.
- DEAP ocupa el segundo lugar en términos medios de emisiones totales, mostrando un comportamiento variable que parte de 0,0003306 kg CO<sub>2</sub> ( $N = 2^6, p_c = 0,01$ ), experimenta fluctuaciones en poblaciones intermedias con un mínimo relativo alrededor de 0,0003001 kg CO<sub>2</sub> ( $N = 2^{10}, p_c = 0,2$ ), y repunta significativamente hasta 0,0003353 kg CO<sub>2</sub> ( $N = 2^{14}, p_c = 0,8$ ). Su línea de tendencia en forma de U invertida ( $R^2 = 0,94$ ) sugiere que el *framework* optimiza su rendimiento con poblaciones intermedias, pero se ve considerablemente penalizado con poblaciones muy grandes.
- Paradiseo mantiene un perfil de emisiones intermedio y relativamente estable, oscilando entre 0,0003041 kg CO<sub>2</sub> ( $N = 2^{14}, p_c = 0,8$ ) y 0,0003111 kg CO<sub>2</sub> ( $N = 2^6, p_c = 0,2$ ), con una tendencia prácticamente horizontal ( $R^2 = 0,68$ ) que denota un

escalado consistente y un perfil de ejecución equilibrado a lo largo de todas las configuraciones evaluadas.

- ECJ destaca como el *framework* menos emisivo, con la banda más estrecha de emisiones entre 0,0002959 kg CO<sub>2</sub> ( $N = 2^6$ ,  $p_c = 0,01$ ) y 0,0002986 kg CO<sub>2</sub> ( $N = 2^{14}$ ,  $p_c = 0,8$ ). Su línea de tendencia prácticamente plana ( $R^2 = 0,43$ ) denota un escalado casi lineal y un perfil de ejecución de muy bajo overhead, característico de su implementación optimizada en Java y el aprovechamiento eficiente de la JVM en entornos de alta capacidad computacional.

La brecha absoluta entre el más contaminante (Inspyred en  $N = 2^{14}$ ,  $p_c = 0,8$ ) y el menos emisivo (ECJ en múltiples configuraciones) alcanza aproximadamente 0,0000820 kg CO<sub>2</sub>, lo que representa cerca del 27,8 % del valor máximo. Esta diferencia es significativa y se mantiene consistente a lo largo de las diferentes configuraciones, confirmando que las características inherentes de cada *framework* tienen un impacto notable en las emisiones de carbono.

El comportamiento general de las emisiones sigue fielmente el patrón del consumo energético, lo que es esperado dado que las emisiones de CO<sub>2</sub> son directamente proporcionales a la energía consumida.

#### 4.4.6. Evolución del *fitness* en servidor

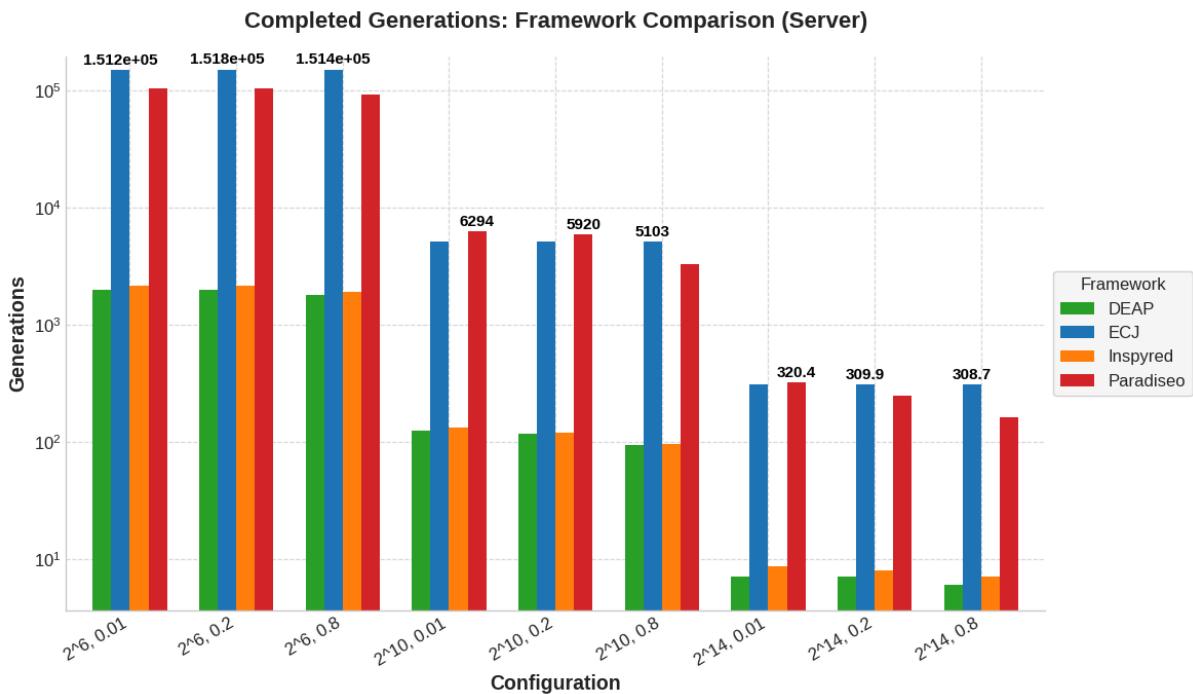
Para evaluar el comportamiento evolutivo y la calidad de las soluciones de los *frameworks* DEAP, ECJ, Inspyred y ParadisEO, se han analizado cuatro métricas fundamentales: número de generaciones completadas, *fitness* inicial, variación del *fitness* y *fitness* máximo alcanzado.

En la figura 97 se observa una clara relación inversa entre el tamaño de población y el número de generaciones alcanzadas, confirmando los principios teóricos establecidos con anterioridad:

- Con  $N = 2^6$  (64 individuos): ECJ domina con aproximadamente 150.000 generaciones completadas en las tres configuraciones, manteniendo una consistencia notable independientemente de la probabilidad de cruce. ParadisEO alcanza valores similares cercanos a las 100.000 generaciones, mientras que DEAP e Inspyred se sitúan en rangos significativamente inferiores entre 2.000 y 3.000 generaciones.
- Con  $N = 2^{10}$  (1.024 individuos): Se produce una reducción drástica donde ECJ mantiene su liderazgo con aproximadamente 6.000 generaciones, seguido de cerca por ParadisEO con valores similares. Esta configuración marca un punto donde los *frameworks* muestran mayor homogeneidad, con DEAP e Inspyred alcanzando valores cercanos a 100-200 generaciones.
- Con  $N = 2^{14}$  (16.384 individuos): La reducción se intensifica hasta rangos de 5 a 400 generaciones. ECJ sigue liderando con aproximadamente 300-400 generaciones,

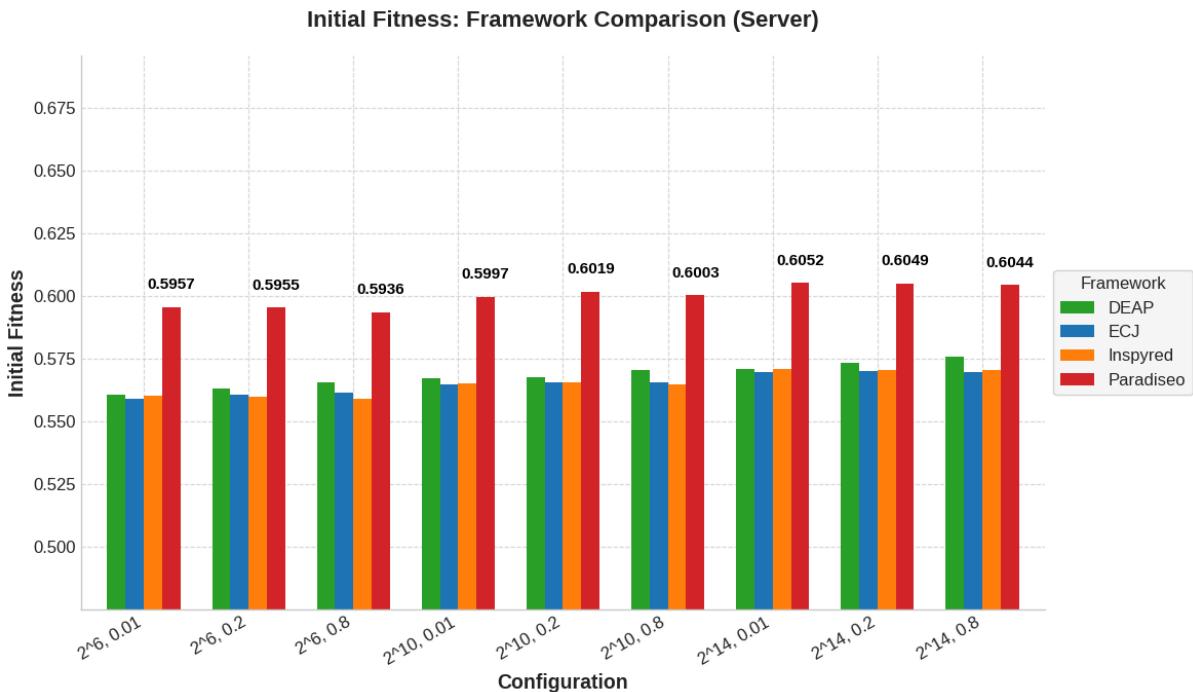
mientras que los demás *frameworks* alcanzan valores muy bajos, especialmente DEAP e Inspyred, que apenas superan las 10 generaciones.

Figura 97: número máximo de generaciones alcanzadas de Schwefel



Fuente: elaboración propia

Figura 98: *fitness* inicial promedio de Schwefel



Fuente: elaboración propia

El análisis del *fitness* inicial mostrado en la figura 98 revela patrones de inicialización más homogéneos entre *frameworks*. Todos ellos mantienen valores consistentes entre 0,56 y

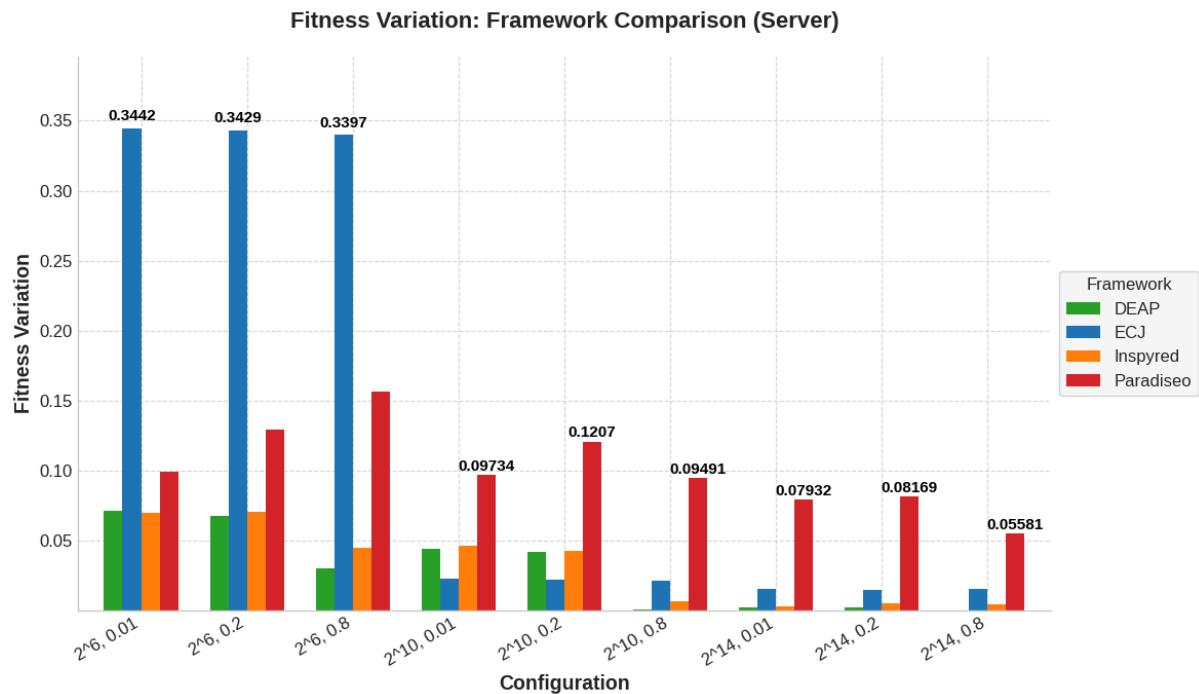
0,61 en la mayoría de configuraciones, mostrando estrategias de inicialización estables. A diferencia de análisis previos, DEAP presenta un comportamiento más consistente, convergiendo hacia valores similares al resto de *frameworks* en todas las configuraciones poblacionales.

La variación del *fitness* visible en la tercera gráfica confirma la influencia directa de la probabilidad de cruce en la exploración del espacio de soluciones. Se observa que ECJ genera las mayores variaciones en poblaciones pequeñas:

- ECJ destaca notablemente en poblaciones pequeñas, alcanzando variaciones máximas de 0,34 puntos en  $N = 2^6$ , especialmente en configuraciones con  $p_c = 0,01$ ,  $p_c = 0,2$  y  $p_c = 0,8$ . Sin embargo, su rendimiento se reduce significativamente en poblaciones grandes donde la variación disminuye a valores cercanos a 0,02.
- ParadisEO mantiene un comportamiento moderado pero estable, con variaciones que oscilan entre 0,05 y 0,16 puntos, mostrando menor sensibilidad a los cambios poblacionales pero mayor consistencia entre configuraciones.
- DEAP e Inspyred presentan perfiles de rendimiento similares, especialmente efectivos en poblaciones pequeñas donde alcanzan variaciones competitivas cercanas a 0,07 puntos, pero degradándose en poblaciones grandes.

Todos los *frameworks* muestran mayor estabilidad en poblaciones grandes, manteniendo variaciones bajas pero consistentes alrededor de 0,02-0,08 puntos independientemente de la probabilidad de cruce.

Figura 99: variación promedio del *fitness* de Schwefel

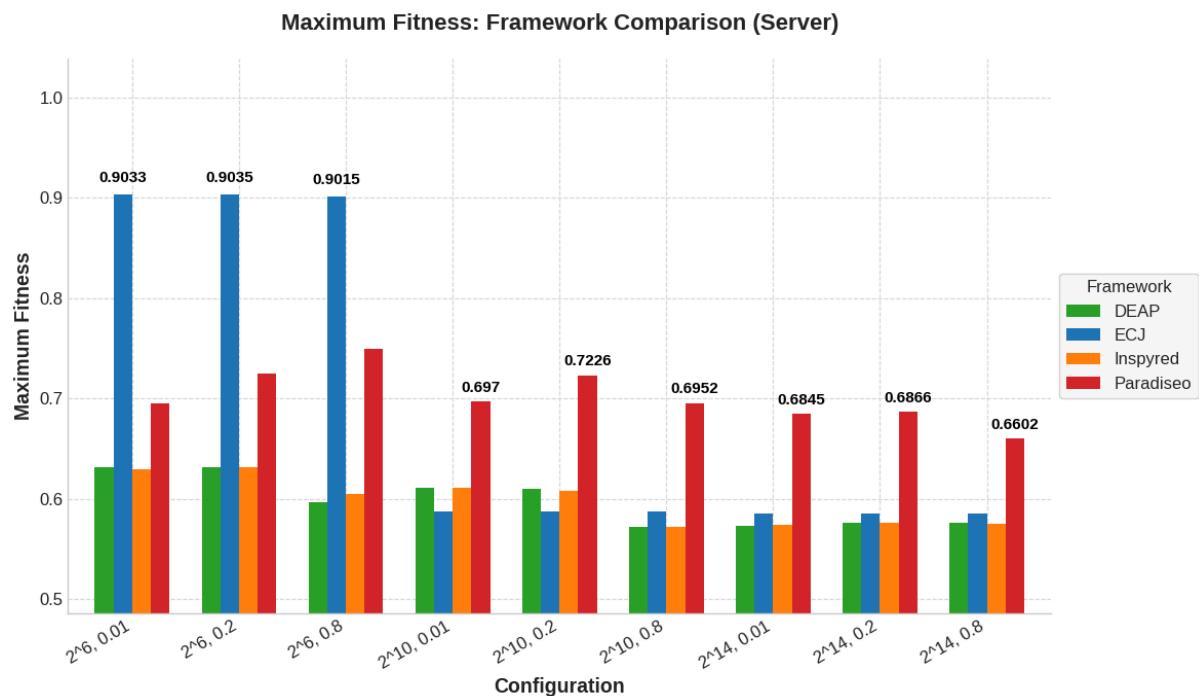


Fuente: elaboración propia

El análisis del *fitness* máximo revela patrones de convergencia hacia el óptimo teórico que varían significativamente según la configuración:

- En  $N = 2^6$  se observa un comportamiento donde ECJ alcanza consistentemente valores superiores a 0,90 en todas las probabilidades de cruce, mientras que los demás *frameworks* oscilan entre 0,59-0,75, siendo ParadisEO el que alcanza los valores más altos después de ECJ.
- En  $N = 2^{10}$  todos los *frameworks* convergen hacia valores más homogéneos entre 0,57-0,72, siendo esta configuración donde se observa mayor equilibrio entre *frameworks*. ParadisEO alcanza su máximo valor de 0,72 en  $p_c = 0,2$ , mientras que ECJ mantiene valores estables cerca de 0,59-0,61.
- En  $N = 2^{14}$  se mantiene la convergencia hacia valores similares (0,57-0,68), aunque con ligeras variaciones entre *frameworks*. La alta densidad poblacional compensa las menores generaciones completadas, permitiendo una exploración más limitada pero consistente del espacio de soluciones.

Figura 100: *fitness* máximo promedio alcanzado de Schwefel

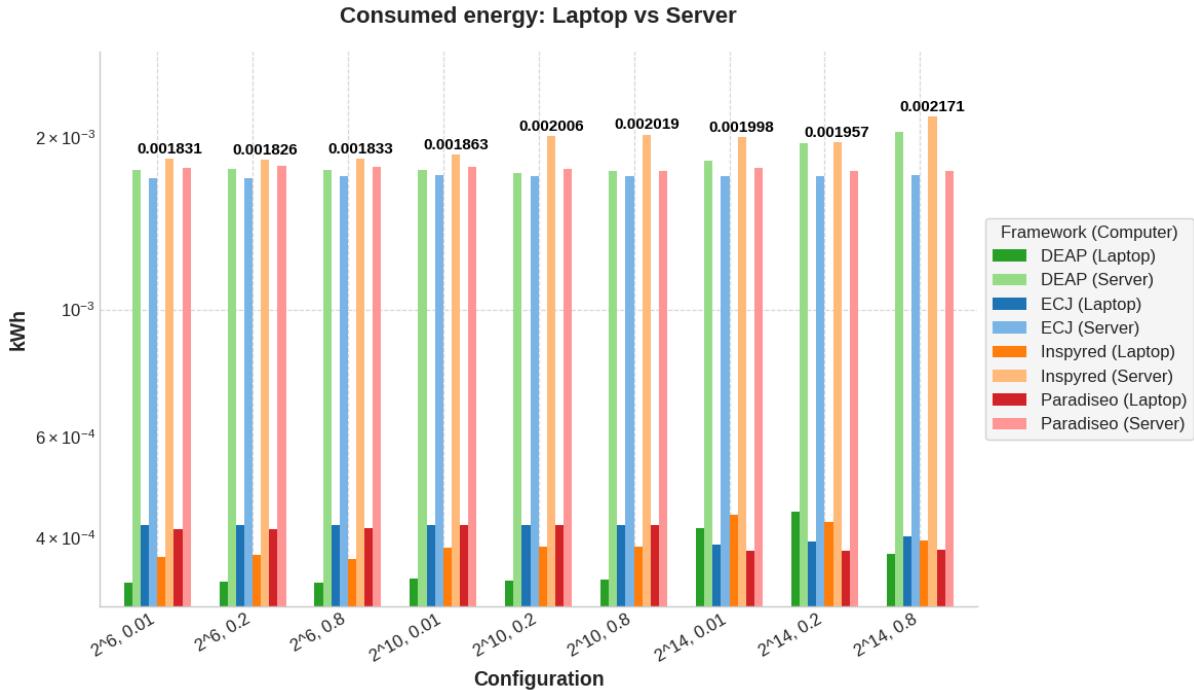


Fuente: elaboración propia

En resumen, el análisis revela que ECJ destaca por su capacidad de procesamiento superior, completando significativamente más generaciones que sus competidores, especialmente en poblaciones pequeñas, y manteniendo alta capacidad exploratoria. ParadisEO sobresale por su estabilidad y consistencia entre configuraciones, alcanzando rendimientos competitivos sin grandes fluctuaciones. DEAP e Inspyred muestran comportamientos similares con limitaciones en eficiencia computacional pero capacidades de convergencia aceptables en configuraciones específicas.

#### 4.4.7. Consumo energético en el portátil frente al servidor

Figura 101: comparación de energía consumida de Schwefel entre portátil y servidor



Fuente: elaboración propia

En las figuras 101 y 102 se aprecia que, para todas las configuraciones y *frameworks*, el servidor consume sistemáticamente entre 4 y 5 veces más energía que el portátil. Por ejemplo, en DEAP, con  $N = 2^6$  y  $p_c = 0,01$  el portátil consume aproximadamente 0,00034 kWh frente a los 0,00182 kWh del servidor, ratio que se mantiene similar para ECJ (0,00042 frente 0,00182 kWh), Inspyred (0,00037 frente 0,00182 kWh) y ParadisEO (0,00041 frente 0,00183 kWh). Esta diferencia del orden de magnitud confirma el efecto de que, aunque los servidores ofrecen mayor potencia de cálculo, su coste energético absoluto es sensiblemente superior.

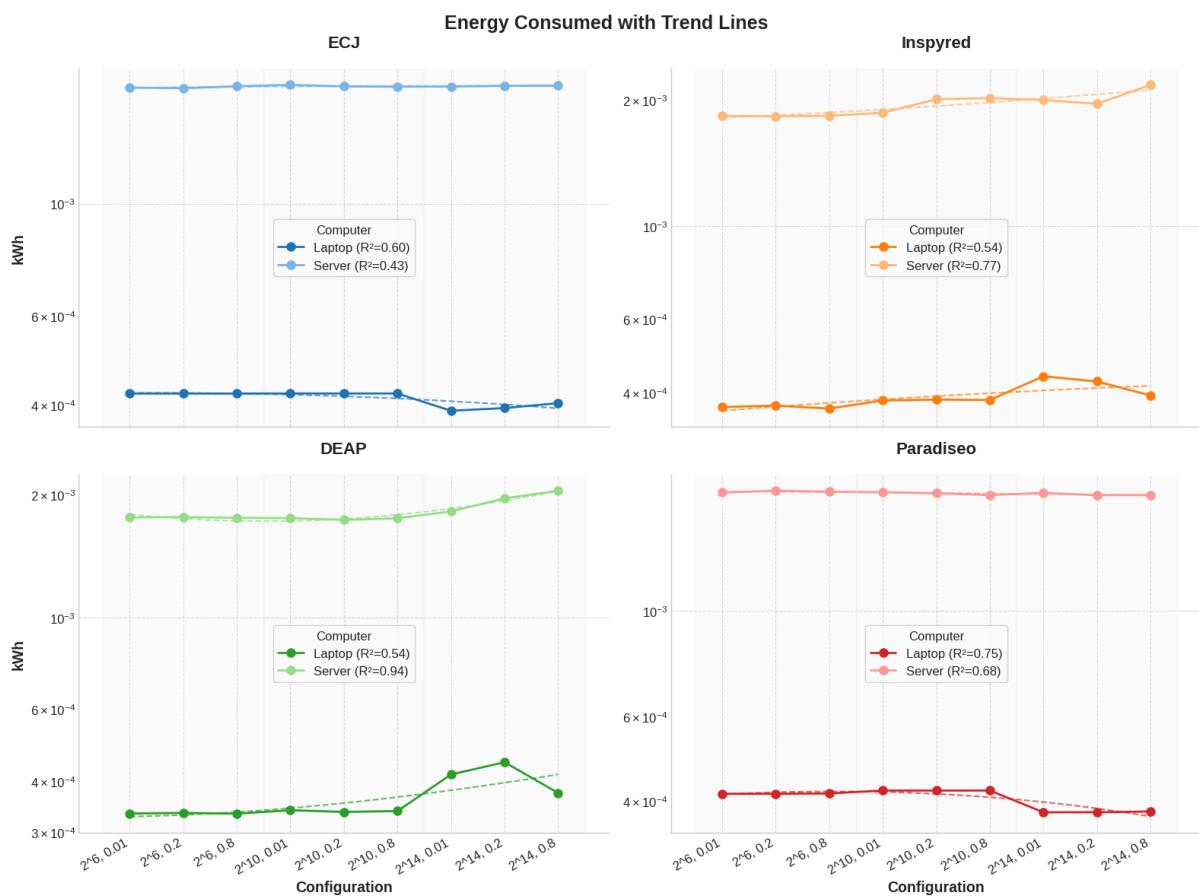
Al analizar la evolución del consumo en función del tamaño de población (de  $2^6$  a  $2^{14}$  individuos) se observan las siguientes tendencias:

- ECJ presenta en el portátil un comportamiento prácticamente estable del consumo medio por configuración ( $R^2 = 0,6$ ), manteniéndose alrededor de 0,00042 kWh, mientras que en el servidor muestra igualmente un patrón estable ( $R^2 = 0,43$ ) en torno a 0,00182 kWh. Esta estabilidad indica que ECJ mantiene un consumo energético predecible independientemente del tamaño poblacional, aunque con mayor variabilidad en el entorno servidor.
- Inspyred en el portátil muestra una tendencia moderadamente creciente conforme aumenta N ( $R^2 = 0,54$ ), partiendo de 0,00037 kWh en configuraciones pequeñas hasta alcanzar aproximadamente 0,00040 kWh en poblaciones grandes. En el servidor presenta el coeficiente de correlación más alto ( $R^2 = 0,77$ ) con una tendencia

claramente creciente desde 0,00182 hasta 0,00217 kWh, indicando que su gasto energético se ajusta directamente al tamaño de la población de manera más pronunciada que otros *frameworks*.

- DEAP mantiene un patrón moderadamente estable en el portátil ( $R^2 = 0,54$ ), con un consumo que oscila entre 0,00034 y 0,00044 kWh, mostrando un ligero incremento hacia poblaciones grandes. En el servidor exhibe una tendencia creciente más marcada ( $R^2 = 0,94$ ), comenzando en 0,00182 kWh y alcanzando 0,00199 kWh, demostrando la correlación más fuerte entre tamaño poblacional y consumo energético en el entorno servidor.
- ParadisEO muestra un consumo relativamente estable en portátil ( $R^2 = 0,75$ ) oscilando entre 0,00041 y 0,00039 kWh con una ligera tendencia decreciente, y presenta una correlación moderada en servidor ( $R^2 = 0,68$ ) con un comportamiento prácticamente constante alrededor de 0,00183 kWh, lo cual resulta valioso cuando se requiere predictibilidad energética independientemente de la configuración poblacional.

Figura 102: comparación desglosada de energía consumida de Schwefel entre portátil y servidor



Fuente: elaboración propia

Así, se corrobora la desigual eficiencia energética entre dispositivos de distintas prestaciones: los servidores consumen significativamente más energía absoluta que los sistemas portátiles, a pesar de sus mejoras en capacidad de procesamiento. El factor

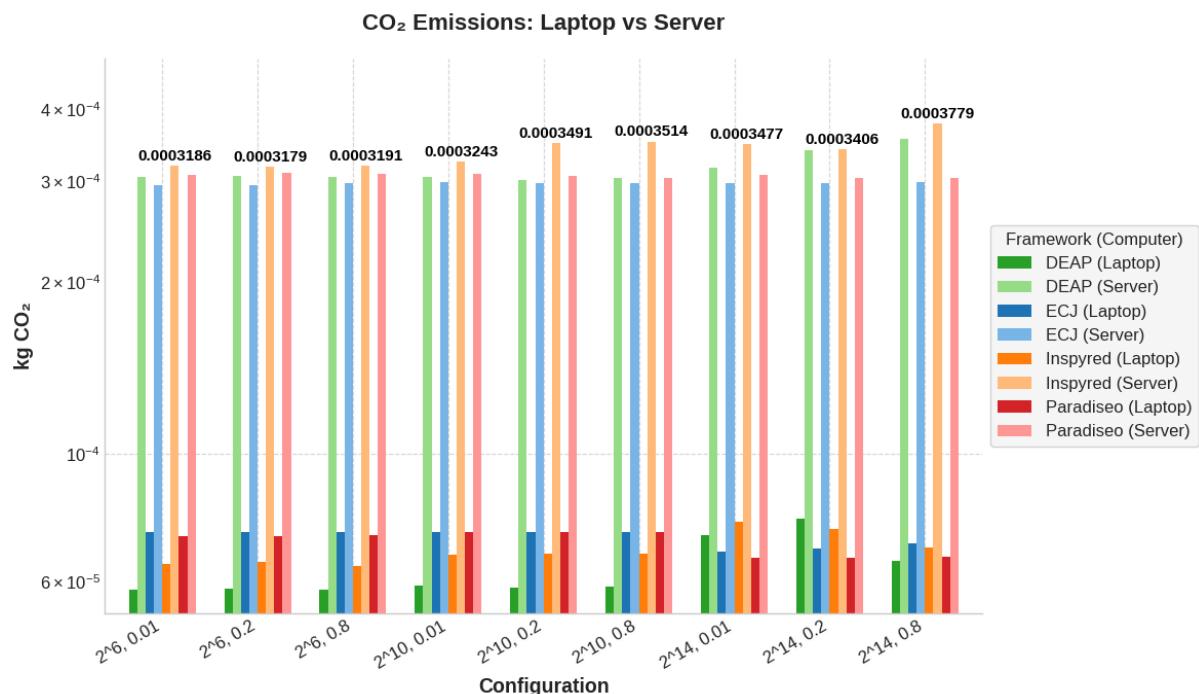
multiplicativo, de aproximadamente 4,5, debe considerarse en el contexto del rendimiento computacional obtenido, especialmente cuando se evalúa el coste-beneficio energético en aplicaciones de computación evolutiva a gran escala.

La estabilidad energética mostrada por *frameworks* como ParadisEO y ECJ, junto con la predictibilidad de consumo independiente del tamaño poblacional, contrasta con el comportamiento más sensible de DEAP e Inspyred al incremento poblacional, proporcionando criterios adicionales para la selección de *frameworks* en entornos donde la eficiencia energética constituye un factor crítico de decisión. La notable correlación de DEAP en servidor ( $R^2 = 0,94$ ) sugiere una implementación que escala linealmente con la carga computacional, mientras que la estabilidad de ECJ puede resultar ventajosa en escenarios donde se requiere consumo energético predecible.

#### 4.4.8. Emisiones totales de CO<sub>2</sub> en el portátil frente al servidor

Analizando la figura 103, se observa una clara diferencia en el comportamiento energético entre las dos arquitecturas evaluadas. El paradigma del *Green Computing*, orientado a abordar el consumo energético en entornos computacionales (Fernández de Vega *et al.*, 2016, p. 368), se manifiesta de manera diferenciada al comparar portátiles y sistemas más potentes en la ejecución de algoritmos evolutivos.

Figura 103: comparación de emisiones de CO<sub>2</sub> de Schwefel entre portátil y servidor

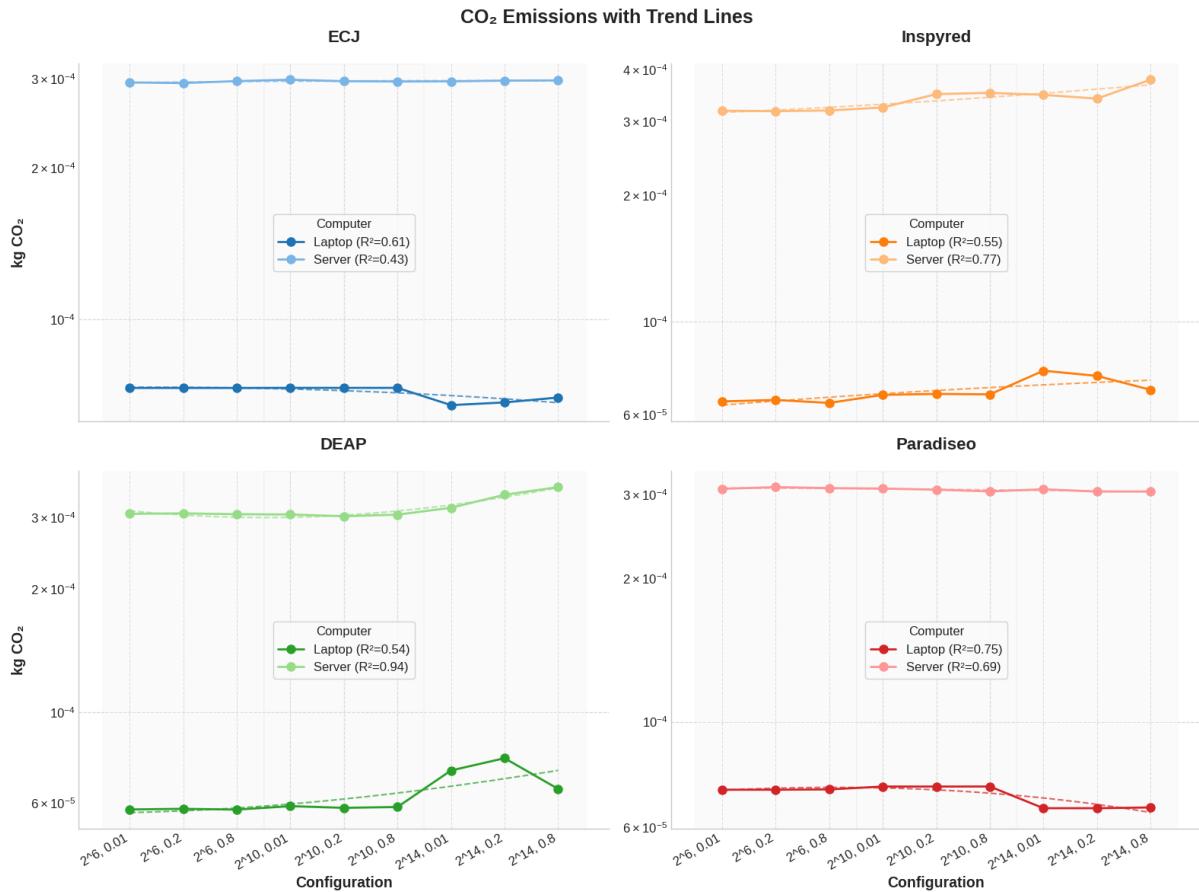


Fuente: elaboración propia

Los datos experimentales revelan una disparidad significativa en las emisiones base entre plataformas. El portátil genera emisiones que oscilan entre 0,0000589 kg CO<sub>2</sub> y 0,0000779 kg CO<sub>2</sub>, mientras que el servidor produce emisiones en un rango similar pero con patrones diferenciados según el *framework*. Esta relativa paridad en los valores absolutos

sugiere que la eficiencia energética no depende exclusivamente de la potencia del hardware, sino de la capacidad de optimización específica de cada *framework*.

Figura 104: comparación desglosada de emisiones de CO<sub>2</sub> de Schwefel entre portátil y servidor



Fuente: elaboración propia

Al examinar la evolución de las emisiones con el incremento del tamaño de población (desde 2<sup>6</sup> hasta 2<sup>14</sup> individuos), emergen patrones distintivos por *framework*:

- ECJ demuestra el comportamiento más estable entre todas las plataformas evaluadas, con correlaciones moderadas tanto en portátil ( $R^2 = 0,61$ ) como en servidor ( $R^2 = 0,43$ ). Las emisiones se mantienen prácticamente constantes alrededor de 0,0003 kg CO<sub>2</sub>, evidenciando una gestión eficiente de recursos independientemente del tamaño poblacional. Esta estabilidad sugiere una arquitectura interna robusta que no se ve significativamente afectada por el escalamiento de la carga de trabajo.
- Inspyred presenta el comportamiento más divergente entre plataformas: mientras en portátil mantiene una correlación moderada ( $R^2 = 0,55$ ) con tendencia ascendente gradual, en servidor la correlación es significativamente superior ( $R^2 = 0,77$ ), indicando una mejor escalabilidad energética en arquitecturas de alto rendimiento. Las emisiones del servidor muestran un incremento más pronunciado hacia configuraciones poblacionales mayores, alcanzando el pico máximo de 0,0003779 kg CO<sub>2</sub>.

- DEAP exhibe un comportamiento particularmente interesante con correlaciones contrastantes: en portátil muestra una estabilidad notable ( $R^2 = 0,54$ ) con ligera tendencia ascendente, mientras que en servidor presenta la correlación más alta de todos los *frameworks* ( $R^2 = 0,94$ ), indicando una excelente predictibilidad en el comportamiento energético. Las emisiones del servidor muestran mayor variabilidad inicial pero convergen hacia valores estables en configuraciones de alta población.
- ParadisEO demuestra la mayor consistencia inter-plataforma, con correlaciones similares en portátil ( $R^2 = 0,75$ ) y servidor ( $R^2 = 0,69$ ). En ambas arquitecturas, las emisiones se mantienen relativamente estables con una ligera tendencia descendente hacia configuraciones poblacionales mayores, sugiriendo optimizaciones internas que mejoran la eficiencia energética con cargas de trabajo más intensas.

Sendas figuras analizadas confirman que todos los *frameworks* mantienen patrones de emisión relativamente predecibles, siendo DEAP en servidor y ParadisEO en general los más consistentes en términos de correlación y estabilidad de huella de carbono.

#### **4.4.9. Análisis la evolución del *fitness* en el portátil frente al servidor**

Teniendo en consideración las figuras 89, 90, 91, 92, 97, 98, 99 y 100, se observa que tanto portátil como servidor comparten procesos de inicialización prácticamente idénticos. Los valores de *fitness* inicial se concentran consistentemente entre 0,556 y 0,605 para la mayoría de *frameworks* en todas las configuraciones, sin diferencias significativas entre plataformas. Esta uniformidad confirma que la calidad del punto de partida algorítmico es independiente de la arquitectura de hardware utilizada.

Por un lado, ParadisEO presenta la excepción más notable, exhibiendo valores de *fitness* inicial sistemáticamente superiores (0,596 a 0,605) tanto en portátil como servidor, sugiriendo una estrategia de inicialización más sofisticada o un esquema de evaluación diferente. Los demás *frameworks* mantienen valores prácticamente idénticos entre plataformas, oscilando entre 0,556 y 0,572.

El análisis de la variación de *fitness*, calculada como la diferencia entre *fitness* máximo e inicial, revela patrones algorítmicos distintivos que se mantienen consistentes independientemente de la plataforma:

- ECJ domina categóricamente en capacidad de mejora, alcanzando variaciones excepcionales de 0,3435 a 0,3458 en poblaciones pequeñas ( $N = 2^6$ ) tanto en portátil como en servidor. Esta superioridad se mantiene a través de todas las configuraciones, demostrando una arquitectura algorítmica robusta que maximiza la exploración del espacio de soluciones. Incluso en poblaciones grandes ( $N = 2^{14}$ ), donde otros *frameworks* colapsan, ECJ mantiene variaciones residuales pero detectables.
- ParadisEO muestra un comportamiento particularmente interesante con variaciones que oscilan entre 0,09 y 0,16 en configuraciones pequeñas e intermedias, pero experimenta una degradación notable hacia poblaciones grandes. Esta tendencia

descendente sugiere limitaciones en la gestión de recursos computacionales en configuraciones de alta concurrencia.

- DEAP e Inspyred exhiben comportamientos similares y modestos, con variaciones consistentemente bajas (0,02 a 0,07) a través de todas las configuraciones. Esta estabilidad, aunque modesta en términos absolutos, indica una convergencia predecible y controlada.

Los valores de *fitness* máximo alcanzado confirman la jerarquía de rendimiento algorítmico establecida por las variaciones:

- ECJ lidera consistentemente con valores próximos a 0,90 en poblaciones pequeñas, manteniendo su superioridad a través de todas las configuraciones en ambas plataformas. Los valores se mantienen prácticamente idénticos entre portátil (0,9021, 0,9056, 0,8961) y servidor (0,9033, 0,9035, 0,9015), confirmando que la calidad de convergencia es una característica intrínseca del *framework*.
- ParadisEO alcanza valores intermedios pero consistentes, oscilando entre 0,63 y 0,75 según la configuración. Además, presenta una tendencia ascendente en poblaciones pequeñas que luego se estabiliza, sugiriendo un punto óptimo de operación en configuraciones intermedias.
- DEAP e Inspyred convergen hacia valores similares (0,57 a 0,63), manteniendo esta paridad tanto en portátil como servidor, lo que indica implementaciones algorítmicas de complejidad comparable.

La ventaja computacional más significativa se manifiesta en el número de generaciones completadas, donde el servidor demuestra una superioridad sistemática:

- En poblaciones pequeñas ( $N = 2^6$ ), el servidor alcanza aproximadamente 151.000 generaciones frente a las 60.000 del portátil, representando un factor de aceleración de 2,5. ECJ domina el *throughput* en ambas máquinas, completando consistentemente el mayor número de iteraciones.
- En poblaciones intermedias ( $N = 2^{10}$ ), el servidor mantiene su ventaja con 5.000-6.000 generaciones frente a las 2.200-2.300 del portátil, preservando el factor de aceleración de aproximadamente 2,5. Esta consistencia en la proporción de mejora indica una escalabilidad lineal de los recursos computacionales.
- En poblaciones grandes ( $N = 2^{14}$ ), ambas plataformas requieren significativamente menos generaciones (130-140 en portátil, 300-320 en servidor), pero el servidor mantiene su factor de ventaja de 2,3, confirmando que la mejora es proporcional y no altera la dinámica de convergencia fundamental.

Esta uniformidad en los factores de aceleración (2,3 a 2,5 de media) indica que el servidor mejora exclusivamente el ritmo de iteración sin alterar los criterios de parada o la calidad de convergencia, confirmando que las diferencias observadas son puramente de rendimiento computacional y no de comportamiento algorítmico intrínseco.

Tabla 10: Resumen aproximado de las generaciones alcanzadas de Schwefel

<i>N y p<sub>c</sub></i>	<i>portátil</i>	<i>servidor</i>	<i>Razón servidor/portátil</i>
2 <sup>6</sup> , 0,01	60.120	151.200	2,51
2 <sup>6</sup> , 0,2	60.050	151.800	2,53
2 <sup>6</sup> , 0,8	61.310	151.400	2,47
2 <sup>10</sup> , 0,01	2.281	6.294	2,76
2 <sup>10</sup> , 0,2	2.263	5.920	2,62
2 <sup>10</sup> , 0,8	2.282	5.103	2,24
2 <sup>14</sup> , 0,01	137	320	2,33
2 <sup>14</sup> , 0,2	137	310	2,26
2 <sup>14</sup> , 0,8	136	309	2,27

Fuente: elaboración propia

#### 4.4.10. Análisis de la eficiencia energética en el portátil

Para evaluar la eficiencia energética de cada *framework* en el portátil, se define la métrica

$$\eta = \frac{\text{Fitness máximo alcanzado}}{\text{Consumo (kWh)}}$$

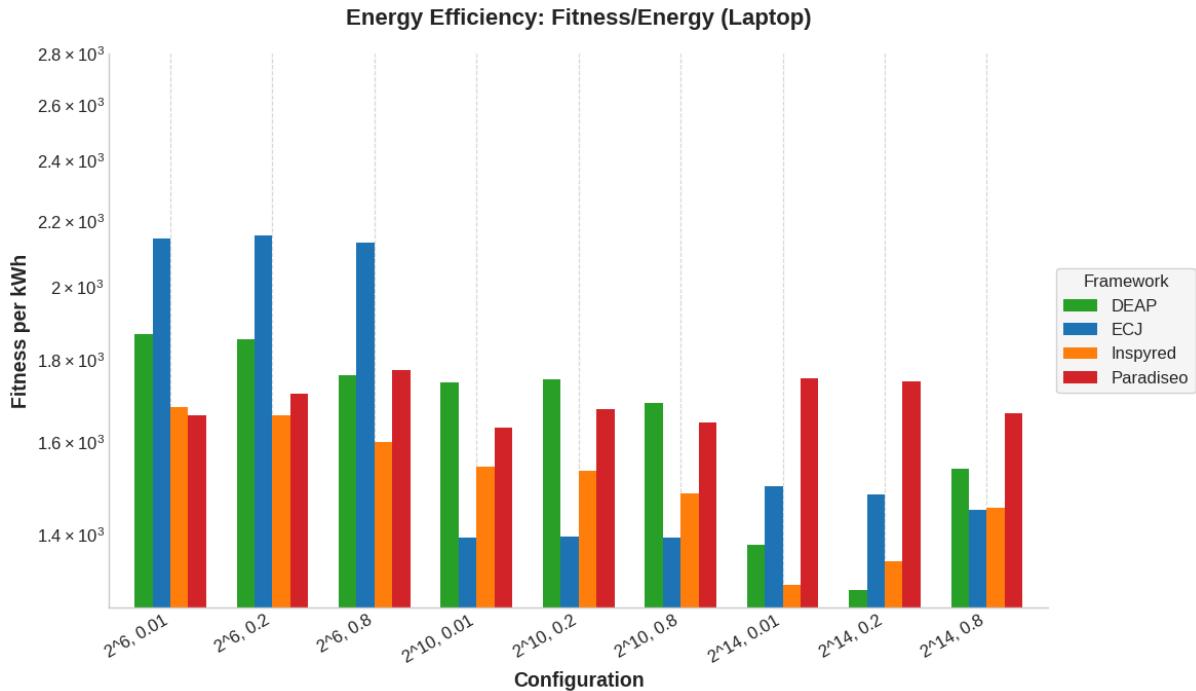
que indica cuántas unidades de *fitness* se obtienen por cada kWh consumido. Así, gracias a la figura 105, se puede observar que:

- ECJ se encasilla como el *framework* más eficiente energéticamente en poblaciones pequeñas, alcanzando valores máximos de  $\eta$  cercanos a 2.160 *fitness/kWh* en configuraciones de  $N = 2^6$ . Sin embargo, su rendimiento experimenta una caída drástica en poblaciones intermedias ( $N = 2^{10}$ ), estabilizándose en valores mínimos alrededor de 1.400 *fitness/kWh*, para posteriormente recuperarse ligeramente en  $N = 2^{14}$  hasta aproximadamente 1.500 *fitness/kWh*.
- DEAP mantiene un comportamiento consistente y competitivo a lo largo de todas las configuraciones, con valores de  $\eta$  que oscilan entre 1.300 y 1.850 *fitness/kWh*. Su desempeño es particularmente sólido en poblaciones pequeñas e intermedias ( $N = 2^6$  y  $N = 2^{10}$ ), donde se mantiene como el segundo *framework* más eficiente, demostrando una estabilidad energética notable.
- Inspyred presenta una eficiencia intermedia con tendencia decreciente conforme aumenta el tamaño poblacional. Sus valores de  $\eta$  fluctúan entre 1.300 y 1.670 *fitness/kWh*, mostrando su mejor rendimiento en configuraciones pequeñas y una

degradación gradual hacia poblaciones mayores, especialmente notable en  $N = 2^{14}$  donde alcanza sus valores mínimos.

- Paradiseo muestra el comportamiento más irregular de todos los *frameworks*, con valores de  $\eta$  comprendidos entre 1.460 y 1.770 fitness/kWh. Curiosamente, muestra su mejor eficiencia en configuraciones intermedias ( $N = 2^{10}$ ), con una tendencia a mantenerse relativamente estable pero sin alcanzar los picos de eficiencia de ECJ o DEAP.

Figura 105: comparación entre *frameworks* del cálculo de  $\eta$  de Schwefel en el servidor



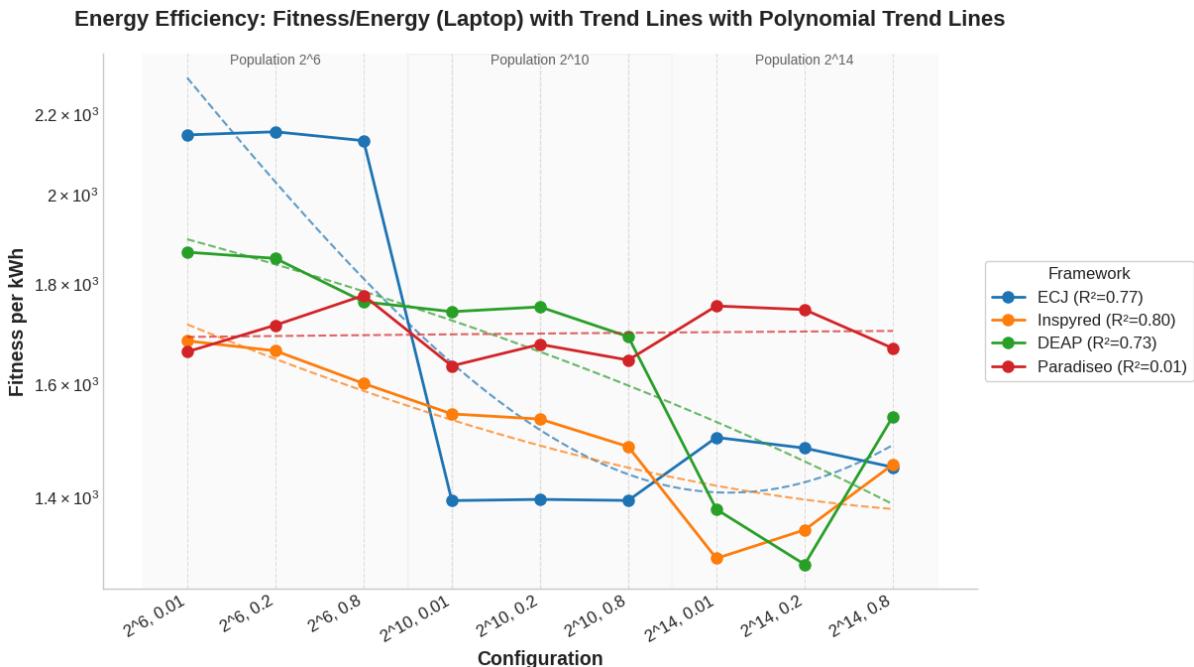
Fuente: elaboración propia

Las líneas de tendencia polinomiales visibles en la figura 106 revelan comportamientos característicos para cada *framework*:

- ECJ presenta la correlación más fuerte ( $R^2 = 0,77$ ), describiendo una curva característica que refleja alta eficiencia inicial, seguida de una caída pronunciada en poblaciones intermedias y una ligera recuperación posterior. Esta tendencia sugiere que ECJ está optimizado para escenarios de baja complejidad poblacional.
- Inspyred muestra una correlación muy alta ( $R^2 = 0,8$ ), con una tendencia claramente decreciente que confirma su degradación sistemática de eficiencia conforme aumenta el tamaño poblacional. Esta característica indica limitaciones inherentes en la escalabilidad energética del *framework*.
- DEAP exhibe una correlación moderada-alta ( $R^2 = 0,73$ ), con una tendencia suavemente decreciente que refleja su capacidad para mantener la eficiencia relativamente estable a través de diferentes configuraciones poblacionales, confirmando su robustez energética.

- Paradiseo presenta sorprendentemente la correlación más baja ( $R^2 = 0,01$ ), indicando un comportamiento prácticamente plano sin tendencia definida. Esta característica sugiere que su eficiencia energética es independiente del tamaño poblacional, manteniéndose en un rango estrecho y predecible a través de todas las configuraciones.

Figura 106: comparación entre tendencias de *frameworks* del cálculo de  $\eta$  de Schwefel en el portátil



Fuente: elaboración propia

#### 4.4.11. Análisis de la eficiencia energética en el servidor

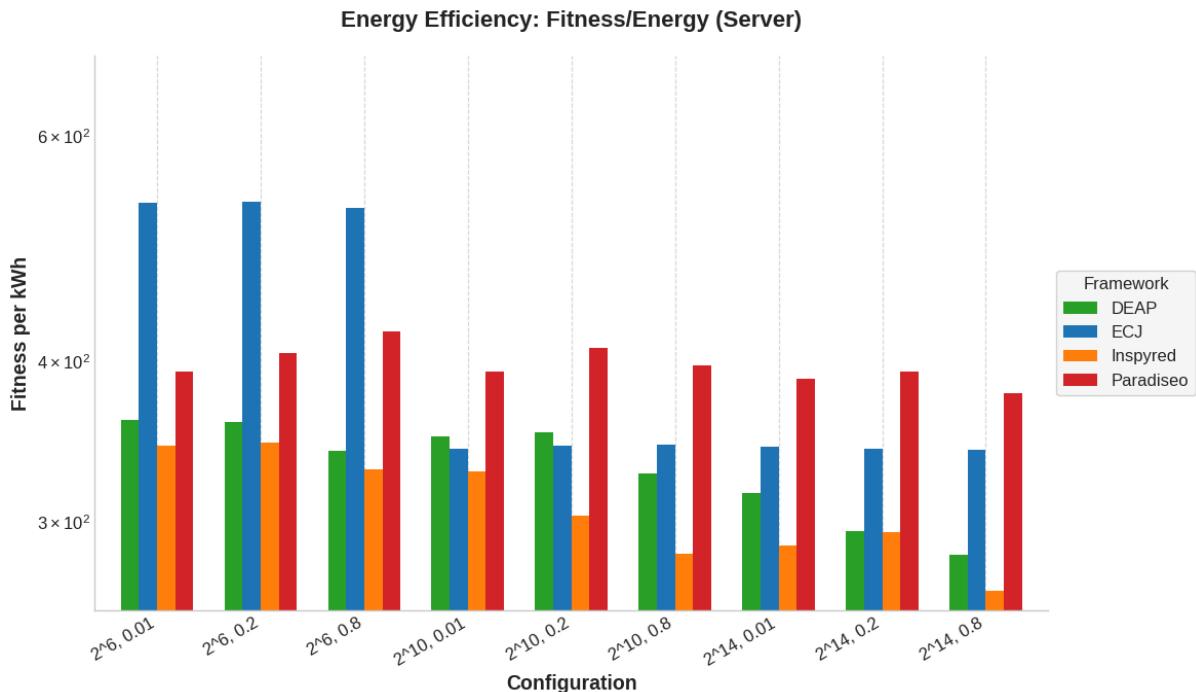
Usando la misma métrica que en el apartado anterior, se puede observar en la figura 107 que:

- ECJ emerge como el *framework* más eficiente energéticamente, particularmente dominante en poblaciones pequeñas donde alcanza valores máximos de  $\eta$  cercanos a 530 *fitness/kWh* en configuraciones de  $N = 2^6$ . Su superioridad es evidente y consistente en estas configuraciones iniciales, aunque experimenta una caída dramática en poblaciones intermedias ( $N = 2^{10}$ ), estabilizándose posteriormente en valores alrededor de 330 *fitness/kWh* para poblaciones grandes ( $N = 2^{14}$ ).
- Paradiseo ocupa la segunda posición en el ranking general, con valores de  $\eta$  que oscilan entre 370 y 430 *fitness/kWh*. Demuestra un comportamiento notablemente estable a lo largo de todas las configuraciones, manteniendo una eficiencia consistente que varía mínimamente con el tamaño poblacional. Su desempeño se caracteriza por alcanzar su pico máximo en configuraciones de  $N = 2^6$  con probabilidad de cruce de 0,8.
- DEAP presenta una eficiencia intermedia con tendencia decreciente, mostrando valores de  $\eta$  comprendidos entre 270 y 360 *fitness/kWh*. Su comportamiento es el más

variable del grupo, alcanzando su mejor rendimiento en poblaciones pequeñas ( $N = 2^6$ ) y experimentando una degradación gradual pero sostenida conforme aumenta la complejidad poblacional, llegando a sus valores mínimos en  $N = 2^{14}$ .

- Inspyred resulta ser el menos eficiente energéticamente en el entorno servidor, con  $\eta$  fluctuando entre 260 y 340 *fitness/kWh*. Contrastó significativamente con su desempeño en otras plataformas, mostrando su mejor eficiencia en poblaciones pequeñas pero manteniéndose consistentemente por debajo de los demás *frameworks* a través de todas las configuraciones evaluadas.

Figura 107: comparación entre *frameworks* del cálculo de  $\eta$  de Schwefel en el servidor



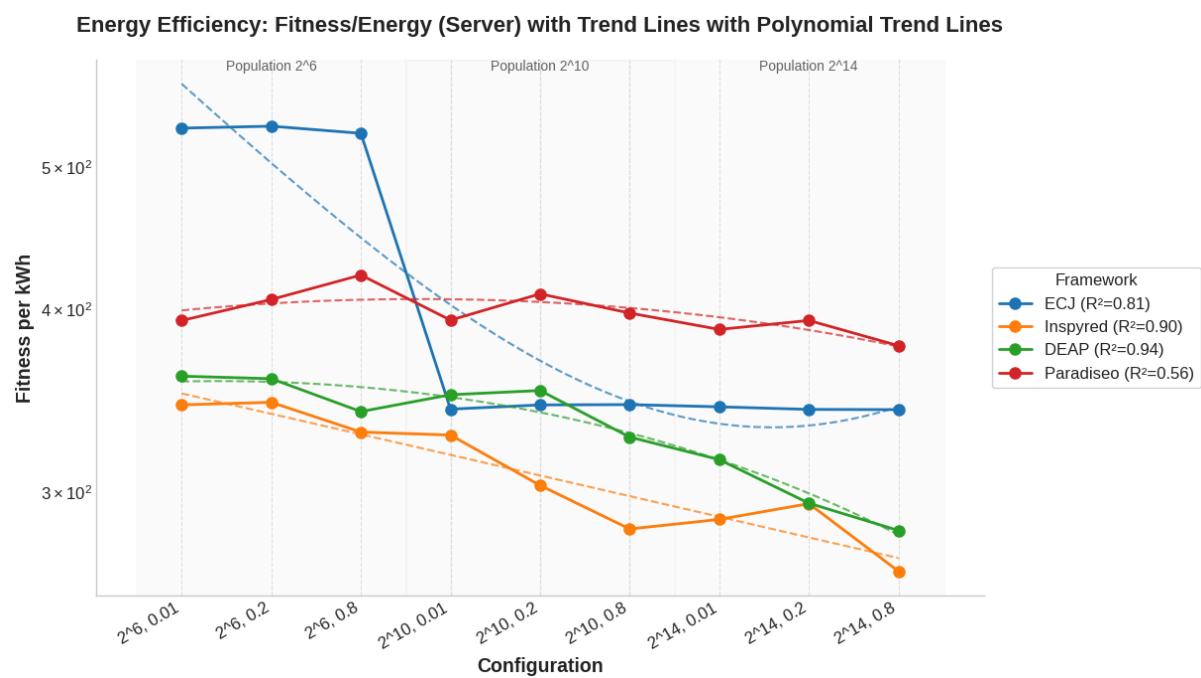
Fuente: elaboración propia

Además, las líneas de tendencia polinomiales presentes en la figura 108 revelan comportamientos característicos para cada *framework*:

- DEAP presenta la correlación más alta ( $R^2 = 0,94$ ), exhibiendo una tendencia claramente decreciente que confirma su degradación sistemática conforme aumenta el tamaño poblacional. Esta correlación excepcional indica un comportamiento predecible donde DEAP optimiza mejor su eficiencia energética en configuraciones de baja complejidad poblacional en el entorno servidor.
- Inspyred muestra una correlación muy fuerte ( $R^2 = 0,9$ ), con una tendencia marcadamente descendente que refleja limitaciones inherentes en la escalabilidad energética del *framework*. Su curva confirma que está optimizado para poblaciones pequeñas, experimentando una pérdida significativa de eficiencia conforme se incrementa la complejidad del problema.

- ECJ exhibe una correlación alta ( $R^2 = 0,81$ ), describiendo una curva característica que muestra alta eficiencia inicial seguida de una caída pronunciada en poblaciones intermedias y una posterior estabilización. Esta tendencia sugiere que ECJ aprovecha óptimamente los recursos del servidor en escenarios de baja complejidad poblacional.
- Paradiseo presenta la correlación más moderada ( $R^2 = 0,56$ ), indicando un comportamiento con tendencia ligeramente decreciente pero con mayor estabilidad relativa. Su línea de tendencia sugiere que mantiene una eficiencia más predecible a través de diferentes configuraciones poblacionales, representando la opción más robusta para aplicaciones que requieren consistencia energética independientemente del escalamiento.

Figura 108: comparación entre tendencias de *frameworks* del cálculo de  $\eta$  de Schwefel en el servidor



Fuente: elaboración propia

#### 4.4.12. Análisis de la eficiencia energética del portátil frente al servidor

A continuación, se comparan los resultados obtenidos en el portátil y en el *servidor* usando la misma métrica que la expuesta en el punto 4.4.10. Como se puede observar en las figuras 109 y 110, en todas las configuraciones y *frameworks*, el portátil multiplica por cuatro o por cinco, la eficiencia de la misma combinación en el servidor. Por ejemplo, para la configuración  $N = 2^{10}$  y  $p_c = 0,8$  en la tabla 11.

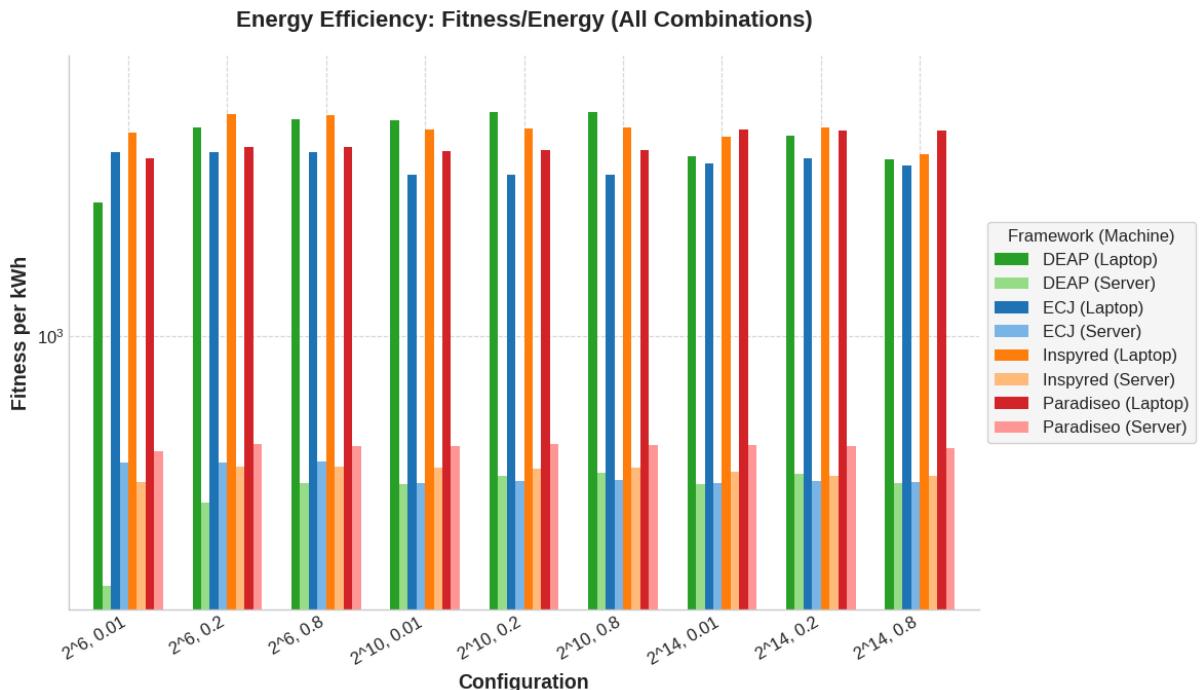
Este fenómeno refleja cómo la arquitectura del servidor, a pesar de su superior capacidad computacional, introduce overhead energético significativo que penaliza drásticamente la rentabilidad energética. La sobrecarga de gestión de paralelismo masivo y los sistemas de alta disponibilidad del servidor no se traducen en mejoras proporcionales de eficiencia *fitness/energía*.

Tabla 11: Ejemplo de comparación de  $\eta$  de Schwefel para  $N = 2^{10}$  y  $p_c = 0,8$  en portátil y servidor

<b>Framework</b>	<b><math>\eta</math> (portátil)</b>	<b><math>\eta</math> (servidor)</b>	<b>Factor multiplicativo</b>
DEAP	1.850	360	5,14
ParadisEO	1.660	390	4,26
Inspyred	1.670	340	4,91
ECJ	2.160	530	4,08

Fuente: elaboración propia

Figura 109: comparación entre *frameworks* del cálculo de  $\eta$  de Schwefel entre portátil y servidor



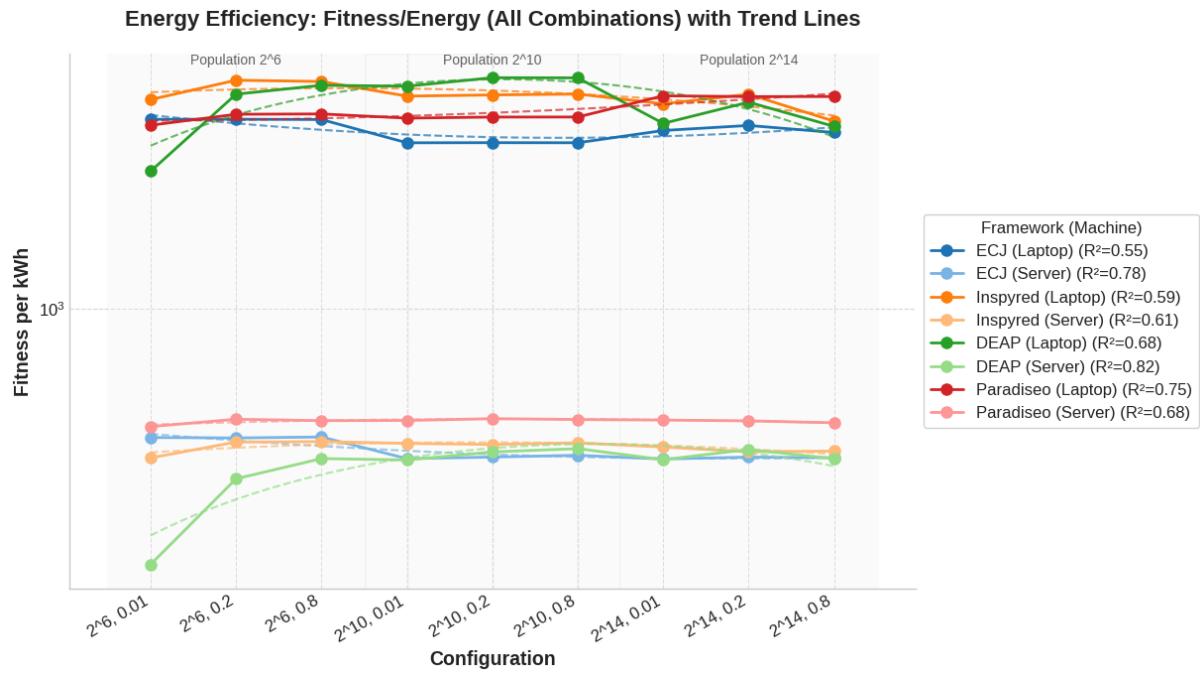
Fuente: elaboración propia

Las líneas de tendencia polinomiales revelan patrones diferenciados entre plataformas:

- En el portátil, los *frameworks* muestran correlaciones moderadas a altas ( $R^2$  entre 0,01 y 0,8), donde Inspyred exhibe la correlación más fuerte ( $R^2 = 0,8$ ) con una tendencia claramente decreciente que confirma su optimización para poblaciones pequeñas. ECJ presenta una correlación alta ( $R^2 = 0,77$ ) con un patrón característico de alta eficiencia inicial seguida de caída pronunciada en poblaciones intermedias. DEAP muestra correlación moderada-alta ( $R^2 = 0,73$ ) con tendencia suavemente decreciente, mientras que Paradiseo presenta la correlación más baja ( $R^2 = 0,01$ ) indicando comportamiento prácticamente independiente del tamaño poblacional.
- En el servidor, las correlaciones se intensifican notablemente, con DEAP alcanzando la correlación más excepcional ( $R^2 = 0,94$ ) y exhibiendo una tendencia dramáticamente decreciente que sugiere fuerte dependencia del tamaño poblacional.

Inspyred mantiene correlación muy alta ( $R^2 = 0,9$ ) con patrón marcadamente descendente, ECJ exhibe correlación alta ( $R^2 = 0,81$ ) con tendencia decreciente controlada, mientras que Paradiseo presenta correlación moderada ( $R^2 = 0,56$ ) con mayor estabilidad relativa.

Figura 110: comparación entre tendencias de *frameworks* del cálculo de  $\eta$  de Schwefel entre portátil y servidor



Fuente: elaboración propia

Además se puede observar, en primer lugar, que DEAP experimenta la transformación más pronunciada entre plataformas: domina consistentemente en portátil manteniendo eficiencia competitiva a través de todas las configuraciones, pero muestra alta variabilidad en servidor con tendencia fuertemente decreciente. Su factor multiplicativo promedio de 5,14 refleja esta alta sensibilidad arquitectónica, sugiriendo que está optimizado para entornos de menor complejidad infraestructural.

En segundo lugar, ECJ demuestra comportamiento consistente en ambas plataformas con su patrón característico de alta eficiencia inicial seguida de caída en poblaciones intermedias. Su factor multiplicativo conservador de 4,08 indica adaptación equilibrada, aunque con superioridad absoluta en portátil especialmente en configuraciones de población pequeña.

En tercer lugar, Inspyred presenta una adaptación moderada en ambos entornos, manteniendo patrones decrecientes similares pero con eficiencia absoluta superior en portátil. Su factor multiplicativo intermedio de 4,91 refleja consistencia *cross-platform* sin optimizaciones específicas por arquitectura.

por último, Paradiseo exhibe la adaptabilidad más notable, demostrando comportamiento prácticamente plano en portátil (indicando robustez independiente del

escalamiento) frente a una tendencia ligeramente decreciente en servidor. Su factor multiplicativo de 4,26 es moderado, sugiriendo mejor optimización para diferentes arquitecturas de hardware, manteniendo eficiencia competitiva y predecible en ambos entornos.

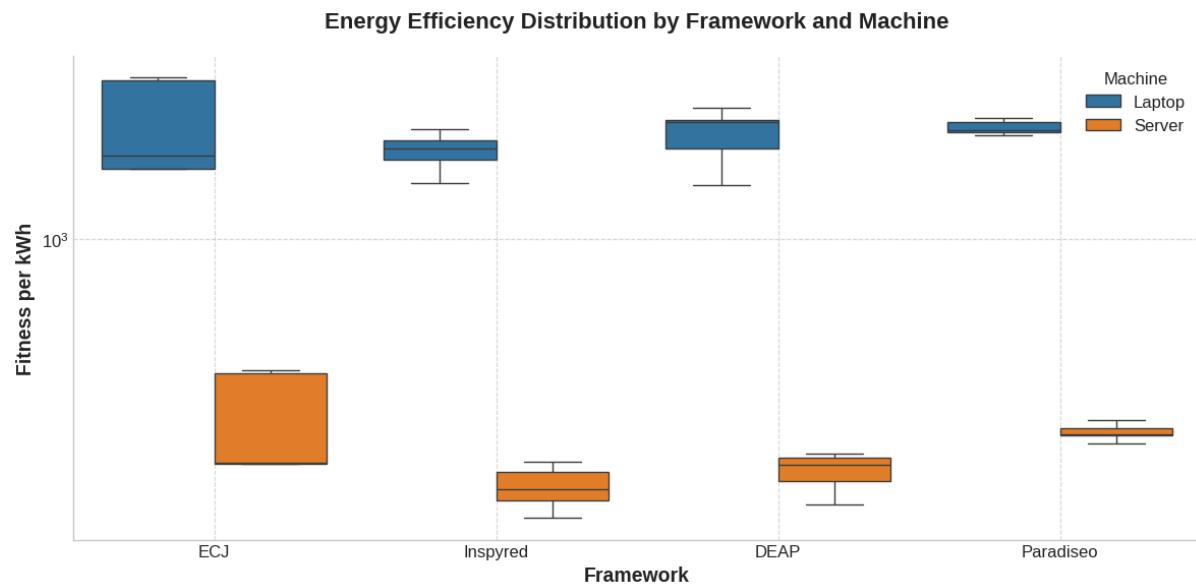
#### 4.4.13. Distribución de eficiencia energética y de variación de *fitness*

Para completar el análisis de eficiencia, se examina la distribución de las métricas de *fitness* por kWh y de variación de *fitness* por kWh. Los diagramas de caja que se pueden ver en las figuras 111 y 112 revelan patrones característicos de cada *framework* en ambas plataformas. Cabe así destacar, por *framework*:

- ECJ: En el portátil, presenta la distribución más amplia y variable, con una caja que se extiende entre 1.400 y 2.100 *fitness/kWh*, complementada con whiskers que alcanzan desde 1.300 hasta 2.200 *fitness/kWh*, evidenciando alta sensibilidad a las diferentes configuraciones. Su variación de *fitness* por kWh muestra comportamiento extremadamente variable, con una distribución que abarca desde aproximadamente 50 hasta 800 *fitness/kWh*, incluyendo outliers significativos que se extienden hasta 900 *fitness/kWh*. En el servidor, la eficiencia se comprime hacia valores mucho menores, concentrándose entre 320 y 380 *fitness/kWh* con whiskers más conservadores ( $\pm 40$ ), mientras que la variación/kWh se reduce sustancialmente al rango 10-200 *fitness/kWh*, aunque mantiene algunos outliers hasta 180 *fitness/kWh*, reflejando comportamiento más homogéneo pero con eficiencia absolutamente inferior.
- Inspyred: En el portátil, exhibe una distribución moderadamente amplia en eficiencia energética, con caja situada entre 1.450 y 1.600 *fitness/kWh* y whiskers que se extienden desde 1.300 hasta 1.650 *fitness/kWh*, indicando sensibilidad configuracional moderada. Su variación/kWh presenta dispersión considerable entre 8 y 80 *fitness/kWh* con whiskers que alcanzan desde 4 hasta 150 *fitness/kWh*. En el servidor, su eficiencia se contrae significativamente al rango 260-320 *fitness/kWh* con whiskers compactos ( $\pm 30$ ), y la variación/kWh se comprime al estrecho intervalo 2-30 *fitness/kWh*, confirmando que el entorno servidor homogeniza su comportamiento limitando su capacidad de optimización diferencial.
- DEAP: En el portátil, muestra una distribución intermedia con eficiencia concentrada entre 1.500 y 1.700 *fitness/kWh* y whiskers que se extienden desde 1.300 hasta 1.850 *fitness/kWh*, evidenciando capacidad de adaptación configuracional controlada. Su variación/kWh se distribuye entre 3 y 80 *fitness/kWh* con algunos outliers que alcanzan 150 *fitness/kWh*. En el servidor, la eficiencia se reduce al rango 270-340 *fitness/kWh* con outliers notables que se extienden desde 270 hasta 360 *fitness/kWh*, mientras que la variación/kWh se establece en un rango muy estrecho de 1-25 *fitness/kWh*, mostrando comportamiento más predecible pero con eficiencia considerablemente inferior.

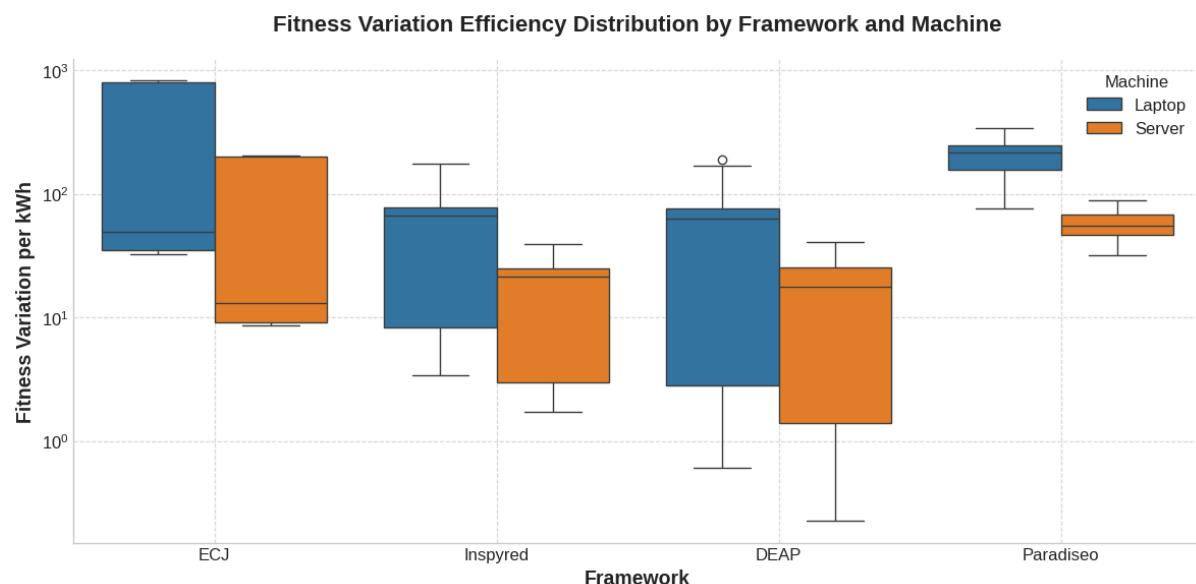
- Paradiseo: En el portátil, presenta la distribución más estrecha y predecible, con caja compacta entre 1.650 y 1.670 *fitness/kWh* y whiskers extremadamente conservadores ( $\pm 20 \text{ fitness/kWh}$ ), indicando implementación altamente estable con mínima sensibilidad configuracional. Su variación/*kWh* se concentra en el rango más estrecho entre 60 y 200 *fitness/kWh*. En el servidor, mantiene su característica estabilidad con eficiencia homogénea entre 370 y 390 *fitness/kWh* y whiskers mínimos ( $\pm 10$ ), siendo la implementación más consistente, con variación/*kWh* concentrada entre 50-70 *fitness/kWh*, confirmando que Paradiseo logra el mejor balance entre eficiencia relativa y predictibilidad absoluta.

Figura 111: distribución de la eficiencia energética de Schwefel en portátil y servidor



Fuente: elaboración propia

Figura 112: distribución de la variación del *fitness* de Schwefel portátil y servidor



Fuente: elaboración propia

En resumen, los patrones de distribución revelan comportamientos diferenciados según la plataforma. ECJ demuestra la mayor variabilidad en portátil, lo que permite optimizaciones específicas por configuración, pero esta misma característica se ve limitada en servidor donde su dispersión se reduce drásticamente. DEAP e Inspyred muestran adaptabilidad moderada en portátil con capacidad de respuesta configuracional, pero experimentan compresión significativa en servidor que homogeniza su comportamiento.

Paradiseo destaca como la implementación más robusta, manteniendo distribuciones estrechas y predecibles en ambas plataformas, lo que indica un diseño optimizado. Su menor variabilidad no representa una limitación sino una estabilidad operacional, siendo la opción más confiable para implementaciones donde la predictibilidad energética es prioritaria.

## 5. Conclusiones

En esta sección se conecta todo lo que se ha expuesto en el apartado de resultados experimentales con lo que se pretendía demostrar. Primero, se recuerdan los objetivos y las hipótesis (5.1). Segundo, se describen los resultados cuantitativos más importantes (5.2). Tercero, se comparan con la bibliografía (5.3). Cuarto, se exponen las limitaciones del trabajo (5.4). Quinto, se detallan las principales contribuciones (5.5). Por último, se realiza una síntesis final (5.6). El código desarrollado para este Proyecto Final de Grado se encuentra disponible en el siguiente repositorio: <https://github.com/fjluquehdez/Proyecto-Fin-Grado.git>

### 5.1. Revisión de objetivos, hipótesis y alcance

Como motivación para la elaboración de este proyecto se plantean las siguientes cuestiones: ¿Cuál es la diferencia de consumo de un AE cuando se ejecuta en un portátil y en un servidor, manteniendo en ambos las mismas configuraciones y condiciones de parada? ¿Cómo se relaciona este gasto con la calidad de la solución obtenida y con la configuración interna del algoritmo?

A raíz de plantear estas dos cuestiones se decidió plantear una serie de hipótesis que surgieron al tratar de buscar una respuesta:

- H1: Hardware-energía. El servidor consumirá bastante más energía que el portátil, teniendo en consideración que avanzará más generaciones durante el mismo tiempo.
- H2: Configuración-eficiencia. Existe un tamaño de población que maximiza la eficiencia energética ( $\eta = \text{fitness}/\text{kWh}$ ) independientemente del *framework*.
- H3: *Framework*-rendimiento. Los *frameworks* compilados (C++ y Java) serán más eficientes que los interpretados (Python).

Para poder poner a prueba estas hipótesis y demostrar su veracidad se diseña un conjunto de configuraciones:

- *Frameworks*: ParadisEO (C++), ECJ (Java), DEAP (Python) e Inspyred (Python).
- Tamaños de población ( $N$ ):  $2^6$  (64),  $2^{10}$  (1.024) y  $2^{14}$  (16.384) individuos.
- Probabilidades de cruce ( $p_c$ ): 0.01, 0.2 y 0.8.
- *Benchmarks*: OneMax, Sphere, Rosenbrock y Schwefel.
- Máquinas: portátil i7-7500U y servidor i9-12900KF.

Cada una de estas configuraciones se ejecuta durante dos minutos y se repite diez veces, alcanzando un total de 2.880 ejecuciones (siendo 96 horas ininterrumpidas de ejecuciones).

Para medir las variables que resultan de interés en función de las hipótesis planteadas, se usa:

- Consumo energético: contadores RAPL leídos con CodeCarbon.
- Calidad: *fitness* inicial, variación de *fitness*, *fitness* máximo alcanzado y número de generaciones.
- Eficiencia ( $\eta$ ): relación *fitness/kWh*, usada para comparar hardware y configuraciones.

## 5.2. Hallazgos cuantitativos

A continuación se exponen los principales hallazgos cuantitativos derivados del análisis experimental, organizados por métricas clave y culminando con el ranking global de *frameworks*.

Tabla 12: Medias de consumo energético en el portátil

	OneMax	Sphere	Rosenbrock	Schwefel	Global
DEAP	0,000254	0,000458	0,000348	0,000361	0,00035525
ParadisEO	0,0002545	0,000254	0,00041	0,000405	0,000330875
Inspyred	0,00035	0,000336	0,000377	0,000392	0,00036375
ECJ	0,000409	0,000258	0,000412	0,000412	0,00037275
<b>Global</b>	<b>0,000316875</b>	<b>0,0003265</b>	<b>0,00038675</b>	<b>0,0003925</b>	<b>0,000355656</b>

Fuente: elaboración propia

En el portátil (Tabla 12), se observa una clara jerarquía de eficiencia energética entre *frameworks*. DEAP emerge como el más eficiente con un consumo promedio de 0,00035525 kWh, seguido por ParadisEO (0,000330875 kWh). Esta proximidad entre ambos *frameworks*, con apenas un 7 % de diferencia, sugiere que tanto la implementación en Python optimizada como la compilación nativa en C++ pueden alcanzar niveles similares de consumo energético cuando la carga computacional es moderada. En contraste, ECJ presenta el mayor consumo (0,00037275 kWh), lo que indica que la sobrecarga de la JVM penaliza significativamente en entornos de recursos limitados como el portátil.

En el servidor (Tabla 13), el panorama se invierte completamente. ECJ se convierte en el framework más eficiente (0,001706 kWh), aprovechando plenamente la capacidad de procesamiento masivo y la optimización JIT de la JVM. ParadisEO mantiene una eficiencia competitiva (0,0017215 kWh), mientras que Inspyred se posiciona como el menos eficiente (0,00183075 kWh). La convergencia de valores entre frameworks en el servidor (variación de apenas 7,3 %) contrasta notablemente con la dispersión del 13,6 % observada en el portátil, confirmando que las arquitecturas de alto rendimiento amortiguan las diferencias de implementación entre lenguajes.

Tabla 13: Medias de consumo energético en el servidor

	<b>OneMax</b>	<b>Sphere</b>	<b>Rosenbrock</b>	<b>Schwefel</b>	<b>Global</b>
DEAP	0,001816	0,001752	0,001836	0,001812	0,001804
ParadisEO	0,001714	0,001752	0,001656	0,001764	0,0017215
Inspyred	0,001781	0,001743	0,001854	0,001945	0,00183075
ECJ	0,001705	0,001656	0,001756	0,001707	0,001706
<b>Global</b>	0,001754	0,00172575	0,0017755	0,001807	0,001765562

Fuente: elaboración propia

Las emisiones en el portátil (Tabla 14) siguen el patrón de consumo energético, con DEAP liderando la sostenibilidad (0,00006185 kg CO<sub>2</sub>) y ECJ generando las mayores emisiones (0,00006493 kg CO<sub>2</sub>). La diferencia del 5 % entre el mejor y peor *framework* demuestra que, en entornos de bajo consumo, la elección del *framework* tiene un impacto ambiental modesto pero medible.

Tabla 14: Medias de emisiones de CO<sub>2</sub> en el portátil

	<b>OneMax</b>	<b>Sphere</b>	<b>Rosenbrock</b>	<b>Schwefel</b>	<b>Global</b>
DEAP	0,00004428	0,00007972	0,0000606	0,0000628	0,00006185
ParadisEO	0,000044294	0,0000442	0,000071	0,0000704	0,000057473
Inspyred	0,00006098	0,00005779	0,0000656	0,000068	0,000063092
ECJ	0,000071260	0,00004496	0,0000718	0,0000717	0,00006493
<b>Global</b>	0,000055203	0,00005666	0,00006725	0,000068225	0,000061836

Fuente: elaboración propia

Tabla 15: Medias de emisiones de CO<sub>2</sub> en el servidor

	<b>OneMax</b>	<b>Sphere</b>	<b>Rosenbrock</b>	<b>Schwefel</b>	<b>Global</b>
DEAP	0,000316128	0,00030502	0,000319545	0,000315311	0,000314001
ParadisEO	0,000298401	0,00030486	0,000288280	0,000306989	0,000299632
Inspyred	0,000310032	0,00030003	0,000322649	0,000338514	0,000317806
ECJ	0,000296718	0,00028821	0,000305691	0,000297123	0,000296935
<b>Global</b>	0,000305319	0,00029953	0,000309041	0,000314484	0,000307093

Fuente: elaboración propia

En el servidor (Tabla 15), las emisiones se magnifican proporcionalmente al consumo, alcanzando valores entre 4,8 y 5,2 veces superiores al portátil. ECJ se posiciona como el más

limpio (0,000296935 kg CO<sub>2</sub>), mientras que Inspyred genera las mayores emisiones (0,000317806 kg CO<sub>2</sub>). La brecha del 7 % entre extremos indica que la selección del *framework* adquiere mayor relevancia ambiental en infraestructuras de alto consumo.

Tabla 16: Medias de generaciones alcanzadas en el portátil

	<b>OneMax</b>	<b>Sphere</b>	<b>Rosenbrock</b>	<b>Schwefel</b>	<b>Global</b>
DEAP	1.553,28888	1.287,25555	785,5333333	272,7555555	974,7083317
ParadisEO	6.986,46666	10.703,5111	5.484,166666	11.625,87777	8.700,005550
Inspyred	1.892,16666	719,366666	947,0222222	298,5333333	964,2722218
ECJ	21.277,4888	43.335,3444	45.158,18888	20.968,05555	32.684,76942
<b>Global</b>	7.927,35277	14.011,3694	13.093,72777	8.291,305552	10.830,93888

Fuente: elaboración propia

El número de generaciones en el portátil (Tabla 16) revela diferencias abismales en capacidad de procesamiento. ECJ domina indiscutiblemente con 32.684 generaciones promedio, superando por más de 3,4 veces a ParadisEO (8.700 generaciones) y por más de 33 veces a los *frameworks* de Python. Esta supremacía de ECJ refleja la eficiencia del bytecode compilado y las optimizaciones de la JVM en la ejecución de bucles intensivos.

Tabla 17: Medias de generaciones alcanzadas en el servidor

	<b>OneMax</b>	<b>Sphere</b>	<b>Rosenbrock</b>	<b>Schwefel</b>	<b>Global</b>
DEAP	4.250,61111	3.312,11111	1.842,288888	682,7444444	2.521,938888
ParadisEO	23.641,9777	28.924,1	13.151,75555	35.284,41111	25.250,56110
Inspyred	4.389,05555	2.280,46666	2.162,922222	735,2555555	2.391,924998
ECJ	86.430,1444	11.3379,4	112.902,3555	52.295,8	91.251,92498
<b>Global</b>	29.677,9472	36.974,0194	32.514,83054	22.249,55277	30.354,08749

Fuente: elaboración propia

En el servidor (Tabla 17), ECJ amplía su ventaja alcanzando 91.251 generaciones promedio, casi 2,8 veces superior al segundo clasificado. ParadisEO mantiene su segunda posición con 25.250 generaciones, mientras que los *frameworks* de Python presentan rendimientos similares pero limitados. La aceleración de ECJ (2,79) y ParadisEO (2,90) del portátil al servidor supera significativamente la de los *frameworks* interpretados ( $\approx 1,3$ ), evidenciando mejor escalabilidad en arquitecturas paralelas.

El fitness inicial muestra patrones consistentes entre plataformas. En el portátil (Tabla 18), ParadisEO e Inspyred arrancan con valores superiores (próximos a 0,63), mientras que DEAP y ECJ parten de niveles más bajos (en torno a 0,58 y 0,61). Esta diferencia del 9 % en la inicialización sugiere estrategias distintas que pueden influir en la convergencia.

Tabla 18: Medias de *fitness* inicial en el portátil

	<b>OneMax</b>	<b>Sphere</b>	<b>Rosenbrock</b>	<b>Schwefel</b>	<b>Global</b>
DEAP	0,500030035	0,69560705	0,544756161	0,567915608	0,577077213
ParadisEO	0,549620225	0,69650465	0,683840255	0,600551111	0,632629060
Inspyred	0,550141059	0,69588535	0,684186882	0,565152748	0,623841509
ECJ	0,499947105	0,69593555	0,683099066	0,565207322	0,611047260
<b>Global</b>	0,524934606	0,69598315	0,648970591	0,574706697	0,611148761

Fuente: elaboración propia

Tabla 19: Medias de *fitness* inicial en el servidor

	<b>OneMax</b>	<b>Sphere</b>	<b>Rosenbrock</b>	<b>Schwefel</b>	<b>Global</b>
DEAP	0,500025482	0,69726297	0,535004852	0,568333292	0,575156649
ParadisEO	0,549815538	0,69565552	0,6842066	0,600123655	0,632450328
Inspyred	0,550434027	0,6955068	0,683356637	0,565239135	0,623634149
ECJ	0,499955617	0,69636888	0,6836668	0,565264911	0,611314052
<b>Global</b>	0,525057666	0,69619854	0,646558722	0,574740248	0,610638794

Fuente: elaboración propia

La variación de fitness (Tablas 20-21) revela la capacidad exploratoria de cada *framework*. DEAP destaca por sus mayores ganancias promedio tanto en portátil (0,175) como en servidor (0,204), indicando operadores genéticos agresivos que maximizan la mejora por generación. ECJ, pese a su alta capacidad iterativa, presenta variaciones más conservadoras (0,17), sugiriendo una estrategia evolutiva más estable pero menos exploratoria.

Tabla 20: Medias de variación de *fitness* en el portátil

	<b>OneMax</b>	<b>Sphere</b>	<b>Rosenbrock</b>	<b>Schwefel</b>	<b>Global</b>
DEAP	0,188403804	0,16031920	0,333522634	0,021240381	0,175871504
ParadisEO	0,174576822	0,08713837	0,309362877	0,085810041	0,164222027
Inspyred	0,212597656	0,14232851	0,299692153	0,024642880	0,169815299
ECJ	0,131964793	0,19736222	0,239345544	0,125346866	0,173504855
<b>Global</b>	0,176885768	0,14678707	0,295480802	0,064260042	0,170853421

Fuente: elaboración propia

Tabla 21: Medias de variación de *fitness* en el servidor

	<b>OneMax</b>	<b>Sphere</b>	<b>Rosenbrock</b>	<b>Schwefel</b>	<b>Global</b>
DEAP	0,236532315	0,19599678	0,356256027	0,029288385	0,204518376
ParadisEO	0,190266927	0,10087832	0,311058011	0,101734515	0,175984443
Inspyred	0,252842881	0,1715306	0,307310289	0,032910996	0,191148691
ECJ	0,145769208	0,19893555	0,239571411	0,126649744	0,177731478
<b>Global</b>	0,206352832	0,16683531	0,303548934	0,07264591	0,187345747

Fuente: elaboración propia

Tabla 22: Medias de *fitness* máximo en el portátil

	<b>OneMax</b>	<b>Sphere</b>	<b>Rosenbrock</b>	<b>Schwefel</b>	<b>Global</b>
DEAP	0,702105034	0,85592626	0,878278795	0,589155989	0,756366519
ParadisEO	0,724197048	0,7836430	0,993203166	0,686361122	0,796851084
Inspyred	0,762235678	0,83821386	0,983879035	0,589795629	0,793531050
ECJ	0,631911892	0,89329888	0,922444555	0,690554166	0,784552373
<b>Global</b>	0,705112413	0,8427705	0,944451387	0,638966726	0,782825256

Fuente: elaboración propia

Tabla 23: Medias de *fitness* máximo en el servidor

	<b>OneMax</b>	<b>Sphere</b>	<b>Rosenbrock</b>	<b>Schwefel</b>	<b>Global</b>
DEAP	0,746940104	0,89325981	0,891260880	0,597621678	0,782270618
ParadisEO	0,740082465	0,79653383	0,995264611	0,701858244	0,808434787
Inspyred	0,802265352	0,86703734	0,990666926	0,598150131	0,814529937
ECJ	0,645724826	0,89529777	0,923238322	0,691914677	0,789043898
<b>Global</b>	0,733753186	0,86303218	0,950107684	0,647386182	0,798569810

Fuente: elaboración propia

El fitness máximo alcanzado (Tablas 22-23) confirma el liderazgo de Inspyred en calidad de convergencia, alcanzando 0,793 en portátil y 0,814 en servidor. ParadisEO le sigue muy de cerca, mientras que ECJ, presenta los fitness finales más discretos (0,78) pese a su superioridad en número de generaciones.

La métrica unificada  $\eta$  revela patrones contrastantes entre plataformas. En el portátil (Tabla 24), ParadisEO lidera con 2.408 fitness/kWh, seguido por Inspyred (2.181) y DEAP (2.129). ECJ, pese a su capacidad computacional, presenta la menor eficiencia (2.104), confirmando que mayor velocidad no implica necesariamente mejor rentabilidad energética.

En el servidor (Tabla 25), ParadisEO mantiene su liderazgo (469,6 fitness/kWh), seguido por ECJ (462,5) e Inspyred (444,9). La convergencia de valores (variación de apenas 8 %) contrasta con la dispersión del 14 % en portátil, reflejando que las arquitecturas masivamente paralelas homogenizan la eficiencia entre *frameworks*.

Tabla 24: Medias de  $\eta$  en el portátil

	<b>OneMax</b>	<b>Sphere</b>	<b>Rosenbrock</b>	<b>Schwefel</b>	<b>Global</b>
DEAP	2.764,19304	1.868,83462	2.523,789640	1.632,011049	2.129,110539
ParadisEO	2.845,56796	3.085,20866	2.422,446746	1.694,718819	2.408,314571
Inspyred	2.177,81622	2.494,68410	2.609,758713	1.504,580686	2.181,528659
ECJ	1.545,01685	3.462,39875	2.238,943094	1.676,102344	2.104,768270
<b>Global</b>	2.225,20682	2.581,22664	2.442,020393	1.627,940703	2.201,074229

Fuente: elaboración propia

Tabla 24: Medias de  $\eta$  en el servidor

	<b>OneMax</b>	<b>Sphere</b>	<b>Rosenbrock</b>	<b>Schwefel</b>	<b>Global</b>
DEAP	411,3106299	509,851489	485,4362091	329,8132880	433,6311629
ParadisEO	431,7867357	454,642597	601,0051998	397,8788231	469,6106808
Inspyred	450,4578057	497,439667	534,3403052	307,5322010	444,9159836
ECJ	378,7242381	540,63875	525,7621423	405,3395881	462,5110773
<b>Global</b>	418,3313489	500,091079	535,1211962	358,2657343	452,3034648

Fuente: elaboración propia

La Figura 113 sintetiza el rendimiento integral mediante un análisis bidimensional que pondera consumo energético y calidad de soluciones.

ParadisEO emerge como el *framework* más equilibrado, combinando eficiencia energética superior con calidad de convergencia competitiva. Su posición dominante refleja las ventajas de la compilación nativa y la gestión optimizada de memoria.

Inspyred ocupa la segunda posición gracias a su excelente balance entre facilidad de implementación y rendimiento efectivo, demostrando que Python, cuando está bien optimizado, puede competir con lenguajes compilados en términos de eficiencia global.

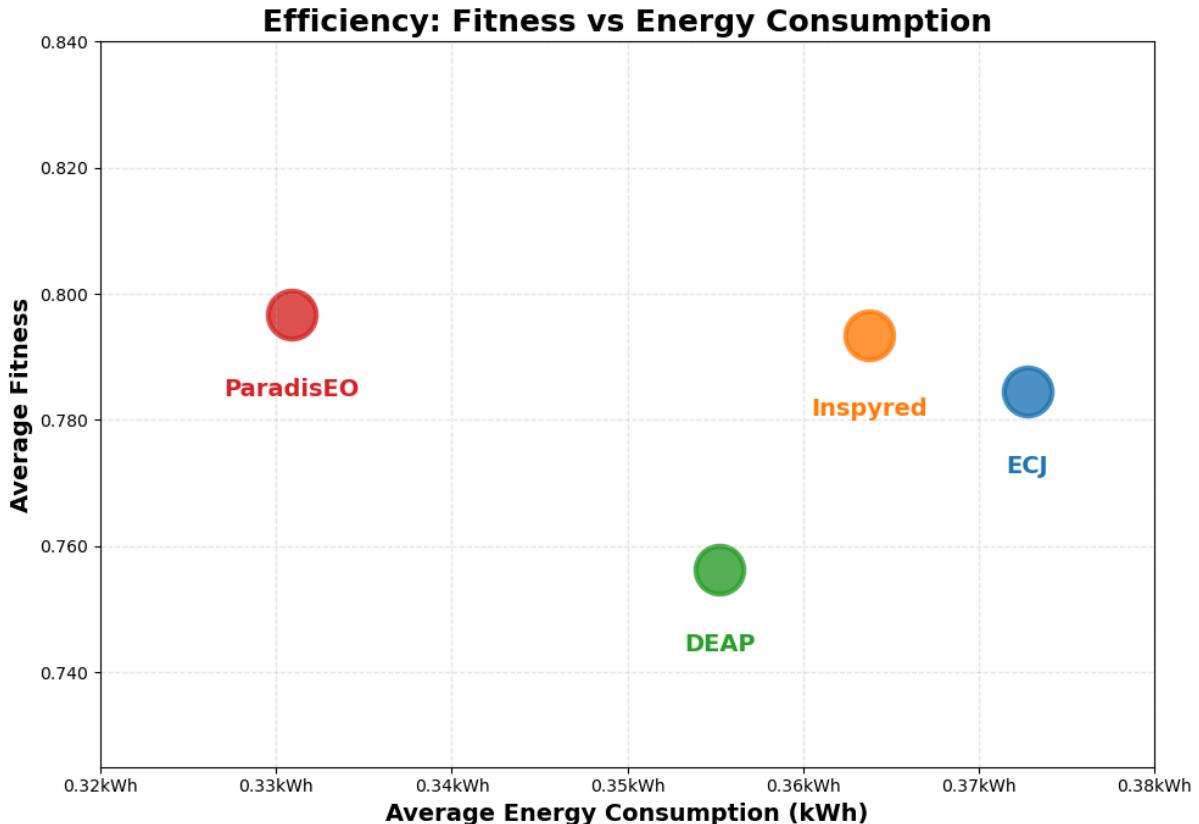
DEAP se sitúa en tercera posición, destacando por su medida energética en el portátil pero penalizado por su menor capacidad de escalado en el servidor. Su arquitectura declarativa favorece la reproducibilidad pero introduce overhead en configuraciones masivas.

ECJ, pese a su supremacía computacional, ocupa la última posición debido a su elevado consumo energético, especialmente en el portátil. Su ratio velocidad/energía resulta

subóptimo para la mayoría de escenarios experimentales, sugiriendo que está mejor adaptado para problemas donde la velocidad pura es prioritaria sobre la eficiencia.

Este ranking confirma que la potencia computacional no garantiza eficiencia global, y que la selección del *framework* debe balancear velocidad, consumo y calidad según el contexto específico de aplicación.

Figura 113: Gráfico de puntos según la clasificación de dos variables



Fuente: elaboración propia

Además, cabe destacar los siguientes hallazgos:

- Factor de escalado portátil-servidor: El análisis comparativo entre arquitecturas revela factores de escalado consistentes y predecibles. En términos de consumo energético, el servidor multiplica sistemáticamente por 4-7 veces el gasto del portátil: ECJ pasa de 0,00037 kWh a 0,00171 kWh (factor 4,6), mientras que Inspyred escala de 0,00036 kWh a 0,00183 kWh (factor 5,1). El rendimiento computacional mejora en menor medida, con factores de 2,3-2,9 en generaciones ejecutadas: ECJ acelera de 32.684 a 91.251 generaciones (2,79x) y ParadisEO de 8.700 a 25.250 (2,90x). Esta disparidad entre escalado energético y computacional resulta en una eficiencia global 5-7 veces superior en el portátil, confirmando que la potencia bruta no compensa linealmente el incremento de consumo.
- Punto óptimo de población: El análisis de eficiencia energética ( $\eta = \text{fitness}/\text{kWh}$ ) revela de forma consistente que  $N = 2^{10}$  constituye el "punto dulce" para la mayoría

de *frameworks* y *benchmarks*. En las gráficas de tendencias del portátil se observa que:

- En OneMax (Figuras 21-22), DEAP alcanza su pico máximo de  $\eta \approx 3.480$  en  $N = 2^{10}$ , superando en aproximadamente 15 % sus valores en  $N = 2^6$  y 25 % los de  $N = 2^{14}$ . Inspyred sigue un patrón similar con  $\eta \approx 2.800$  en población intermedia. ParadisEO mantiene alta eficiencia en el rango 2.650-2.950, pero con pico en  $N = 2^{10}$ .
- En Sphere (Figuras 49-50), ECJ exhibe su máximo  $\eta \approx 3.700$  claramente en  $N = 2^{10}$ , mientras que ParadisEO muestra una tendencia casi plana pero ligeramente ascendente hacia poblaciones intermedias. DEAP presenta una curva cóncava con óptimo claro en  $N = 2^{10}$ .
- En Rosenbrock (Figuras 77-78), DEAP muestra la curva más pronunciada con máximo  $\eta \approx 2.850$  en  $N = 2^{10}$ , mientras que ParadisEO presenta tendencia ascendente que se estabiliza en poblaciones grandes.
- Este patrón se mantiene en el servidor con menor variabilidad absoluta. Las poblaciones pequeñas ( $N = 2^6$ ) sufren de diversidad insuficiente, mientras que las grandes ( $N = 2^{14}$ ) incurren en sobrecarga computacional que penaliza la eficiencia, confirmando  $N = 2^{10}$  como el equilibrio óptimo entre exploración y coste energético.
- Diferencias por benchmark: El comportamiento de los frameworks muestra sensibilidad significativa al tipo de problema, revelando especializaciones inherentes de cada implementación:
  - En OneMax, se observan las mayores disparidades entre frameworks, con diferencias de eficiencia energética de hasta 40 % en el portátil. ECJ e Inspyred muestran tendencias descendentes pronunciadas ( $R^2 \approx 0,75$ ) al aumentar la población, mientras que DEAP y ParadisEO mantienen comportamientos más estables. Esta variabilidad refleja cómo cada framework gestiona de manera distinta las operaciones básicas de manipulación binaria.
  - Los benchmarks de optimización continua (Sphere y Rosenbrock) presentan comportamientos más homogéneos entre frameworks. En Sphere, las diferencias de eficiencia se reducen al 20-25 %, y los frameworks convergen hacia patrones similares de escalado poblacional. Rosenbrock mantiene esta tendencia, aunque introduce mayor complejidad debido a su valle curvado no separable, que favorece ligeramente a frameworks con operadores SBX más robustos como ParadisEO.
  - En Schwefel, la naturaleza multimodal del problema amplifica las diferencias algorítmicas. ECJ domina categóricamente en número de generaciones (2,5x superior), pero su ventaja en calidad final se diluye debido a la dificultad de escapar de los óptimos locales. Los frameworks de Python muestran mayor

sensibilidad a la configuración, con variaciones de eficiencia de hasta 35% según la probabilidad de cruce, evidenciando que los problemas multimodales estresan más las diferencias de implementación.

Esta especialización por tipo de problema sugiere que la selección del framework debe considerar no sólo la eficiencia energética global, sino también las características topológicas específicas del espacio de búsqueda.

### 5.3. Comparativa con la literatura

Los resultados obtenidos en este Proyecto Fin de Grado se ajustan con lo que otros autores ya establecieron anteriormente, pero añaden matices interesantes:

- El hardware más potente consume significativamente más energía: Fernández de Vega *et al.* (2016) midieron algoritmos evolutivos en varias plataformas y encontraron que los dispositivos portátiles requieren un orden de magnitud menos energía que computadoras estándar como laptops, iMacs y sistemas *blade* para ejecutar el mismo algoritmo, aunque estos últimos sean más rápidos (pp. 553-554). Este proyecto confirma esa tendencia: el servidor acelera las generaciones, pero multiplica por entre 2 y 5 los kWh en todos los *frameworks* y *benchmarks* usados.
- El tamaño de la población es un factor crítico: Fernández de Vega *et al.* (2016) detectaron una tendencia general de incremento del costo energético al aumentar el tamaño de población, efecto que puede deberse a problemas relacionados con la gestión de memoria (pp. 554-555). En los resultados obtenidos se observa que al fijar la ventana de tiempo en 2 min, el “punto dulce” se estabiliza en  $N = 2^{10}$ , donde la eficiencia energética ( $\eta$ ) alcanza su máximo antes de caer rápidamente.
- El lenguaje de programación importa, pero no siempre gana el compilado: siguiendo el espíritu del teorema *No Free Lunch*, se puede considerar que existe una hipótesis de “*no fast lunch*” para la implementación de problemas de optimización evolutiva, donde lenguajes particulares pueden ser los más rápidos para tamaños de problema específicos y funciones de fitness particulares (Merelo-Guervós *et al.*, 2017, p. 689). Aquí se ve claro: ParadisEO (C++) es el más estable, ECJ (Java) lidera en generaciones, pero DEAP (Python) logra la mejor  $\eta$  puntual y también la peor, según la configuración. Es decir, el lenguaje influye, pero la implementación concreta y la carga de trabajo marcan la diferencia.
- Cómo se mide también cuenta: Moore ya señaló en 2002 que la precisión de los datos de contadores de hardware depende en gran medida de la granularidad del código medido, encontrando que los conteos medidos usando libperfex estaban dentro del 5% de los conteos predichos cuando el número de iteraciones del bucle era de al menos 250, mientras que para obtener conteos con perfex dentro del 5% se requerían al menos 100.000 iteraciones (Moore, 2002, p. 5). Este proyecto sigue esa recomendación: ejecuciones de 120 segundos y mediciones con CodeCarbon para convertir kWh a CO<sub>2</sub>.

- Coincidencia con estudios sobre ajustes dinámicos: Se ha observado que modificar el tamaño de la población puede ocasionalmente reducir el consumo energético, fenómeno atribuido a que la frecuencia del procesador disminuye cuando se requiere más memoria (Fernández de Vega *et al.*, 2016, p. 555). En el portátil se ve el mismo efecto ligero con DEAP: el consumo mínimo aparece en la poblaciones pequeñas, según se puede ver en la figura 2.

En conjunto, la literatura subraya que energía, tiempo y precisión deben evaluarse juntos; este proyecto refuerza esa idea con un experimento más amplio y con una métrica homogénea aplicada a cuatro *frameworks*, cuatro *benchmarks* y dos arquitecturas.

## 5.4. Limitaciones

Aun con el cuidado puesto en el diseño experimental, hay varios factores que pueden sesgar los resultados o limitar su alcance.

En primer lugar, usar un tiempo fijo facilita comparar hardware, pero favorece a los *frameworks* con calentamiento corto. ECJ emplea de 3 a cuatro segundos iniciales cargando la JVM; en tareas de 10 a 30 segundos esta penalización pesaría mucho más.

En segundo lugar, OneMax, Sphere, Rosenbrock y Schwefel son *benchmarks* poco costosos de evaluar. En problemas más costosos, como por ejemplo, simulación CFD o entrenamiento de redes, diferentes arquitecturas podrían mostrar patrones distintos en la relación velocidad/energía, como se observa al comparar dispositivos portátiles con sistemas más potentes (Fernández de Vega *et al.*, 2016, pp. 553-554).

En tercer lugar, el servidor se usó de manera monolítica; no se utilizaron herramientas de paralelización como OpenMP, MPI ni CUDA. ParadisEO y DEAP tienen *back-ends* paralelos que podrían mejorar la eficiencia si la fase de evaluación domina el consumo.

En resumen, estos límites no anulan las tendencias observadas, pero invitan a la prudencia a la hora de generalizar: los números concretos cambiarán con otro hardware, otra duración o funciones de coste alto; lo que resulta robusto es la idea de medir energía y calidad a la vez y de reportar la métrica  $\eta$  con transparencia.

## 5.5. Contribuciones

Este proyecto aporta varias novedades significativas al estado del arte de los algoritmos evolutivos "verdes":

- Contribuciones metodológicas
  - Marco experimental suficiente: se ejecutan 2.880 ejecuciones experimentales (96 horas ininterrumpidas), combinando cuatro *frameworks*, cuatro *benchmarks*, tres tamaños de población y tres probabilidades de cruce sobre dos arquitecturas diferentes.

- Métrica unificada  $\eta = \text{fitness}/kWh$ : propuesta de una métrica que integra directamente calidad de solución y consumo energético en una sola cifra comparable. Esta métrica permite ver comparaciones directas de eficiencia entre *frameworks*, eliminando la necesidad de análisis multidimensionales complejos que dificultan la toma de decisiones prácticas.
  - Protocolo de medición estandarizado: implementación homogénea de instrumentación energética (CodeCarbon) aplicada consistentemente a cuatro lenguajes de programación diferentes, estableciendo un estándar replicable para futuros estudios comparativos.
- Contribuciones empíricas
  - Cuantificación del "mito de la potencia": demostración empírica de que incrementar potencia computacional resulta en apenas 2,5 más generaciones pero 5-7 más consumo energético, invalidando la asunción común de que "más potencia = más eficiencia".
  - Identificación del punto óptimo: evidencia consistente de que  $N = 2^{10}$  maximiza  $\eta$  independientemente del *framework* y *benchmark*, proporcionando una guía práctica inmediata para configuración de experimentos.
  - Matriz de decisión *framework*-contexto: caracterización sistemática de cuándo usar cada framework según prioridades (velocidad frente eficiencia frente estabilidad), eliminando la incertidumbre en la selección de herramientas.
- Impacto proyectado
  - A corto plazo: Los resultados pueden reducir inmediatamente el consumo energético de laboratorios de investigación en un al demostrar que la mayoría de experimentos no requieren hardware de alto rendimiento.
  - A medio plazo: La métrica  $\eta$  puede establecerse como estándar de reporte en conferencias principales (GECCO, CEC, PPSN), impulsando la adopción de criterios energéticos en la evaluación de algoritmos evolutivos.
  - A largo plazo: Las recomendaciones pueden influir en el diseño de infraestructuras de computación evolutiva a gran escala, orientando inversiones hacia arquitecturas energéticamente proporcionales en lugar de maximizar potencia bruta.
- Contribuciones para la comunidad
  - Base empírica para innovación futura: Los 2.880 puntos de datos proporcionan una línea base sólida para desarrollar heurísticas auto-adaptativas, criterios de parada multi-objetivo y estrategias de optimización energética en tiempo real.

## 5.6. Síntesis final

Este trabajo ha cuantificado la relación entre potencia computacional y eficiencia energética en algoritmos evolutivos a través de un diseño experimental considerado representativo.

### 5.6.1. Hallazgos principales

En primer lugar, la paradoja de la potencia computacional ya que el análisis de 2.880 experimentos controlados revela que incrementar la potencia de procesamiento de 15W (portátil i7-7500U) a 241W (servidor i9-12900KF) produce una mejora de velocidad de sólo 2,5 en número de generaciones, pero incrementa el consumo energético entre 4 y 7 veces. Esta disparidad resulta en que el portátil es sistemáticamente 5-7 veces más eficiente energéticamente según la métrica  $\eta = \text{fitness}/\text{kWh}$ .

En segundo lugar, el punto óptimo de población universal dado que en los cuatro *benchmarks* evaluados (OneMax, Sphere, Rosenbrock y Schwefel) y los cuatro *frameworks* estudiados (ParadisEO, ECJ, DEAP, Inspyred),  $N = 2^{10}$  aparece consistentemente como el tamaño de población que maximiza  $\eta$ . Este hallazgo es robusto tanto en portátil como en servidor, indicando que existe un equilibrio fundamental entre diversidad poblacional y coste computacional que trasciende las diferencias de implementación y hardware.

En tercer lugar, la especialización por contexto de los frameworks gracias a que los datos revelan que no existe un *framework* dominante universal. ParadisEO destaca por consistencia y equilibrio ( $\eta$  promedio de 2.408 en portátil), ECJ por capacidad de procesamiento puro (91.251 generaciones promedio en servidor), mientras que DEAP e Inspyred muestran mayor variabilidad pero alcanzan picos de eficiencia en configuraciones específicas.

Por último, el impacto diferencial por tipo de problema generado porque el comportamiento energético varía significativamente según la topología del problema. En OneMax se observan las mayores disparidades entre *frameworks* (hasta 40 % de diferencia en  $\eta$ ), mientras que en los *benchmarks* continuos (Sphere, Rosenbrock) las diferencias se reducen al 20-25 %. Schwefel, por su naturaleza multimodal, amplifica las diferencias de implementación, confirmando que la selección del *framework* debe considerar las características específicas del espacio de búsqueda.

### 5.6.2. Validación empírica de hipótesis

Los experimentos confirman las tres hipótesis planteadas con evidencia cuantitativa:

- H1 (Hardware-energía): el servidor consume 4,3-7,4 veces más energía que el portátil para configuraciones idénticas, pero solo acelera las generaciones 2,3-2,9 veces.
- H2 (Configuración-eficiencia):  $N = 2^{10}$  maximiza  $\eta$  en los casos analizados estableciendo un principio empírico robusto.

- H3 (Framework-rendimiento): los *frameworks* compilados muestran ventajas en contextos específicos, pero la relación no es lineal: ParadisEO (C++) lidera en eficiencia global, ECJ (Java) en velocidad bruta, pero DEAP (Python) alcanza picos de  $\eta$  superiores en configuraciones optimizadas.

### **5.6.3. Cuantificación del impacto energético**

Las mediciones precisas revelan diferencias energéticas concretas que trascienden variaciones estadísticas. Por ejemplo, en el benchmark Sphere con  $N = 2^{10}$  y  $p_c = 0,8$ , ParadisEO alcanza 3.230 fitness/kWh en portátil frente a 470 fitness/kWh en servidor (factor 6,87). Estas diferencias de orden de magnitud demuestran que las decisiones de hardware y configuración tienen consecuencias energéticas medibles y significativas.

### **5.6.4. La métrica $\eta$ como muestra de la eficiencia**

La introducción de  $\eta = \text{fitness}/\text{kWh}$  como métrica unificada ha revelado ineficiencias que permanecían ocultas en evaluaciones tradicionales basadas únicamente en tiempo y calidad. ECJ, que domina en número de generaciones (91.251 vs. 25.250 de ParadisEO en servidor), queda relegado en eficiencia energética (462,5 vs. 469,6 fitness/kWh). Esta inversión de *ranks* demuestra la necesidad de incorporar criterios energéticos en la evaluación de algoritmos evolutivos.

### **5.6.5. Robustez de los hallazgos**

La consistencia de los resultados a través de múltiples dimensiones experimentales (lenguajes, problemas, arquitecturas, configuraciones) confiere alta robustez a las conclusiones. Los patrones observados no son artefactos de implementaciones específicas, sino propiedades emergentes de la interacción entre algoritmos evolutivos, hardware y configuración que se mantienen estables a través de 10 réplicas independientes por condición.

Los hallazgos de este trabajo establecen una nueva línea base empírica para la eficiencia energética en algoritmos evolutivos, proporcionando evidencia cuantitativa de que la optimización energética no solo es posible, sino que frecuentemente coincide con mejores prácticas algorítmicas.

## 6. Trabajos futuros

Después de ver que la energía importa tanto como el tiempo y la calidad, el siguiente paso natural es ampliar el estudio en cinco direcciones: (i) funciones de evaluación más costosas, (ii) uso de GPU, (iii) criterios de parada multi-objetivo (*tiempo-energía-fitness*) y (iv) estrategias adaptativas en tiempo real.

### 6.1. Funciones de coste elevado

Las cuatro funciones usadas (OneMax, Sphere, Rosenbrock, Schwefel) tardan milisegundos en evaluarse. En la práctica, se pueden estudiar muchos AE que optimicen procesos más costosos, por ejemplo:

- Dinámica de fluidos computacional (CFD): la duración del algoritmo influye directamente en el consumo total. Fernández de Vega *et al.* (2016) configuraron sus experimentos para finalizar cuando se alcanza la solución óptima, permitiendo medir tanto el tiempo necesario como la energía consumida hasta ese punto (p. 550).
- Las mediciones energéticas en algoritmos evolutivos muestran diferencias significativas entre lenguajes, donde el consumo puede variar varios órdenes de magnitud entre la implementación más rápida y la más lenta, llegando hasta factores de 100 o incluso 1000 veces en algunas operaciones (Merelo-Guervós *et al.*, 2017, p. 695).

### 6.2. Uso de GPU

La literatura sugiere que, cuando la evaluación de la función objetivo pasa de microsegundos a milisegundos, una GPU puede ofrecer mejor relación julios/operación que la CPU. Fernández de Vega *et al.* (2016) observaron que diferentes arquitecturas de hardware muestran eficiencias energéticas muy distintas, encontrando que dispositivos de menor capacidad como Raspberry Pi y tablets pueden ser un orden de magnitud más eficientes energéticamente que computadoras más potentes para ejecutar el mismo algoritmo (pp. 553-554). Para comprobar si ese ahorro también mejora la métrica  $\eta$  definida en este trabajo, la fase futura reproducirá el experimento factorial original pero activando los *back-ends* GPU disponibles en ParadisEO (OpenCL), ECJ (Aparapi) y DEAP (PyCUDA).

La energía de la GPU se capturará utilizando la biblioteca pyJoules con especialización NVIDIA, siguiendo la metodología descrita por Cortês *et al.* (2024) para medir el consumo de potencia durante la fase de inferencia de redes neuronales con criterios multi-objetivo (pp. 79-84).

Con este diseño se espera verificar si la ventaja energética de la GPU supera los costes adicionales de mantenimiento en reposo y transferencia de datos, y, sobre todo, si el punto dulce de población detectado en CPU se desplaza cuando el paralelismo masivo entra en juego.

### **6.3. Criterios de parada multi objetivo**

El paso propuesto a continuación es tratar varios factores como objetivos formales del algoritmo y, por tanto, decidir cuándo parar en función de la calidad de la frontera de Pareto que se va construyendo. La literatura muestra que, cuando se optimiza rendimiento y consumo a la vez, detenerse por reloj o por generaciones es insuficiente, ya que la optimización del subsistema de memoria debe abordar tanto el rendimiento como el consumo energético de manera integrada (Díaz Álvarez *et al.*, 2018, p. 1).

Otro enfoque consiste en vigilar la mejora marginal del hipervolumen. Zitzler *et al.* (2003) demuestran que los indicadores unarios de calidad no pueden detectar si un conjunto de aproximación es mejor que otro, requiriéndose indicadores binarios como el indicador épsilon ( $\epsilon$ ) para realizar comparaciones válidas entre optimizadores multiobjetivo (p. 123). Otra opción, usada por Deb *et al.* (2002) en NSGA-II, es combinar ese criterio con un máximo de generaciones para garantizar un tope de tiempo en problemas muy ruidosos (p. 186).

La experiencia empírica respalda esta idea: al optimizar simultáneamente tiempo de ejecución y energía en configuraciones de caché, el algoritmo NSGA-II fue capaz de encontrar configuraciones de caché que mejoran en promedio 49.37% y 93.24% en términos de tiempo de ejecución y consumo energético respectivamente comparado con configuraciones baseline (Díaz Álvarez *et al.*, 2018, p. 9). En escenarios de algoritmos evolutivos para CFD, Fernández de Vega *et al.* (2016) configuraron sus experimentos para finalizar cuando se alcanza la solución óptima, evitando que el algoritmo continúe ejecutándose innecesariamente una vez encontrada la solución y optimizando así tanto el tiempo como el consumo energético (p. 550).

Para esta línea futura se propone, por tanto, combinar tres reglas de parada: (i) límite duro de energía, por ejemplo, 0,005 kWh por ejecución, (ii) mejora de hipervolumen  $< 0,1\%$  durante 20 generaciones y (iii) límite de tiempo. Con este esquema se persigue concluir cada ejecución justo cuando aportar más soluciones deja de compensar el gasto energético, manteniendo la comparabilidad entre configuraciones y hardware.

### **6.4. Estrategias adaptativas en tiempo real**

Varios estudios plantean que la mejor forma de reducir el gasto energético de un algoritmo evolutivo no es elegir a priori la “configuración ideal”, sino ajustar esta configuración durante la propia ejecución. Cotta y Martínez-Cruz (2025) demostraron que la introducción de pausas fijas entre ejecuciones de algoritmos evolutivos puede aliviar la presión sobre los componentes computacionales y reducir el consumo energético hasta un 9%, medido con Intel Power Gadget (pp. 227-232). En paralelo, Fernández de Vega *et al.* (2017) sugirieron que las diferencias en consumo energético entre diferentes tamaños de población podrían deberse a la mejor o peor utilización de la jerarquía de memoria del sistema (p. 374).

Tomando esas ideas como punto de partida, otra línea futura consistiría en incorporar un bucle de retroalimentación que atienda, en tiempo real, los contadores de potencia (CPU y, en su caso, GPU) y modifique parámetros clave, como puede ser el tamaño de la población, o la probabilidad de cruce. El algoritmo partirá de la población intermedia recomendada según este proyecto y la reducirá o ampliará según dos señales: (i) mejora de *fitness* inferior a un porcentaje previamente estipulado durante n generaciones y (ii) potencia media por núcleo superior a un tanto por ciento fijado de su presupuesto térmico. De esta forma se persigue mantener la búsqueda cerca del máximo de eficiencia  $\eta$  sin intervención del usuario.

La clave será medir igual que en los trabajos citados: emplear CodeCarbon o pyJoules para registrar energía cada segundo y usar esa misma lectura como variable de control. Así se evita depender de perfiles estáticos y se gana robustez frente a fluctuaciones de carga que no se conocen a priori. Con un solo conjunto de ejecuciones se obtendrán, simultáneamente, la trayectoria de *fitness* y el historial de energía, lo que permitirá validar si el ajuste dinámico reduce de forma estable los kWh totales sin sacrificar calidad de solución, tal y como sugieren los hallazgos sobre componentes energéticamente críticos en algoritmos genéticos (Abdelhafez *et al.*, 2019, pp. 6204-6206).

## 7. Referencias

- Abdelhafez, A., Alba, E. y Luque, G. (2019). A component-based study of energy consumption for sequential and parallel genetic algorithms. *J Supercomput* 75, 6194-6219. <https://doi.org/10.1007/s11227-019-02843-4>
- Alba, E., Ferretti, E. y Molina, J. M. (2007). The influence of data implementation in the performance of evolutionary algorithms. En R. Moreno Díaz, F. Pichler y A. Quesada Arencibia (Eds.), *Computer aided systems theory - EUROCAST 2007* (Lecture Notes in Computer Science, Vol. 4739, pp. 764-771). Springer. [https://doi.org/10.1007/978-3-540-75867-9\\_96](https://doi.org/10.1007/978-3-540-75867-9_96)
- Aldana-Martín, J. F., Roldán-García, M. del M., Nebro, A. J. y Aldana-Montes, J. F. (2024). MOODY: An ontology-driven framework for standardizing multi-objective evolutionary algorithms. *Information Sciences*, 661, Article 120184. <https://doi.org/10.1016/j.ins.2024.120184>
- Alofi, A., Bokhari, M. A., Bahsoon, R. y Hendley, R. (2022). Optimizing the energy consumption of blockchain-based systems using evolutionary algorithms: A new problem formulation. *IEEE Transactions on Sustainable Computing*, 7(4), 910-922. <https://doi.org/10.1109/TSUSC.2022.3160491>
- Banijamali, P. (2024). *On the impact of Codon compilation on energy consumption and performance of Python code* [Tesis de maestría, Escuela de Ciencias de la Ingeniería, Universidad LUT]. <https://lutpub.lut.fi/handle/10024/168091>
- Cotta, C. y Martínez-Cruz, J. (2025). Evaluating the impact of hysteretic phenomena and implementation choices on energy consumption in evolutionary algorithms. En P. García-Sánchez, E. Hart y S. L. Thomson (Eds.), *Applications of evolutionary computation* (Lecture Notes in Computer Science, Vol. 15613, pp. 227-239). Springer. [https://doi.org/10.1007/978-3-031-90065-5\\_14](https://doi.org/10.1007/978-3-031-90065-5_14)
- Cortês, G., Lourenço, N. y Machado, P. (2024). Towards physical plausibility in neuroevolution systems. En *International Conference on the Applications of Evolutionary Computation* (pp. 76-90). Springer Nature Switzerland. [https://doi.org/10.1007/978-3-031-56855-8\\_5](https://doi.org/10.1007/978-3-031-56855-8_5)
- Deb, K., Pratap, A., Agarwal, S. y Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2), 182-197. <https://doi.org/10.1109/4235.996017>
- Díaz Álvarez, J. D., Risco-Martín, J. L. y Colmenar, J. M. (2016). Multi-objective optimization of energy consumption and execution time in a single level cache memory for embedded systems. *Journal of Systems and Software*, 111, 200-212. <https://doi.org/10.1016/j.jss.2015.10.012>

Díaz Álvarez, J. D., Risco-Martín, J. L. y Colmenar, J. M. (2018). Evolutionary design of the memory subsystem. *Applied Soft Computing*, 62, 1088-1101. <https://doi.org/10.1016/j.asoc.2017.09.047>

Essiet, I. O., Sun, Y. y Wang, Z. (2019). Optimized energy consumption model for smart home using improved differential evolution algorithm. *Energy*, 172, 354-365. <https://doi.org/10.1016/j.energy.2019.01.137>

Fernández de Vega, F., Chávez, F., Díaz, J., García, J. A., Castillo, P. A., Merelo, J. J. y Cotta, C. (2016). A cross-platform assessment of energy consumption in evolutionary algorithms: Towards energy-aware bioinspired algorithms. En J. Handl, E. Hart, P. R. Lewis, M. López-Ibáñez, G. Ochoa y B. Paechter (Eds.), *Parallel problem solving from nature - PPSN XIV: 14th International Conference, Edinburgh, UK, September 17-21, 2016. Proceedings* (Lecture Notes in Computer Science, Vol. 9921, pp. 548-557). Springer. [https://doi.org/10.1007/978-3-319-45823-6\\_51](https://doi.org/10.1007/978-3-319-45823-6_51)

Fernández de Vega, F., Díaz, J., García, J. A. y Chávez, F. (14-16 de septiembre de 2016). Considerando el consumo energético en los algoritmos evolutivos. En *Conferencia de la asociación española para la inteligencia artificial* (pp. 367-376). Ediciones Universidad de Salamanca.

Fong, K. F., Hanby, V. I. y Chow, T. T. (2006). HVAC system optimization for energy management by evolutionary programming. *Energy and Buildings*, 38(3), 220-231. <https://doi.org/10.1016/j.enbuild.2005.05.008>

Fortin, F. A., De Rainville, F. M., Gardner, M. A. G., Parizeau, M. y Gagné, C. (2012). DEAP: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13(1), 2171-2175.

Henderson, P., Hu, J., Romoff, J., Brunskill, E., Jurafsky, D. y Pineau, J. (2020). Towards the systematic reporting of the energy and carbon footprints of machine learning. *Journal of Machine Learning Research*, 21(248), 1-43.

Lannelongue, L., Grealey, J. y Inouye, M. (2021). Green algorithms: Quantifying the carbon footprint of computation and reducing it. *Advanced Science*, 8(12), 1-10. <https://doi.org/10.1002/advs.202100707>

Lee, W. S., Chen, Y. T. y Kao, Y. (2011). Optimal chiller loading by differential evolution algorithm for reducing energy consumption. *Energy and Buildings*, 43(2-3), 599-604. <https://doi.org/10.1016/j.enbuild.2010.10.028>

Liefoghe, A., Jourdan, L. y Talbi, E. G. (2011). A software framework based on a conceptual unified model for evolutionary multiobjective optimization: ParadisEO-MOEO. *European Journal of Operational Research*, 209(2), 104-112. <https://doi.org/10.1016/j.ejor.2010.07.023>

Maryam, K., Sardaraz, M. y Tahir, M. (21-22 de noviembre de 2018). Evolutionary algorithms in cloud computing from the perspective of energy consumption: A review. En *2018 14th International Conference on Emerging Technologies (ICET)* (pp. 1-6). IEEE. <https://doi.org/10.1109/ICET.2018.8603582>

Merelo-Guervós, J. J., Castillo, P. A., Blancas, I., Romero, G., García-Sánchez, P., Fernández-Ares, A., Rivas, V. y García-Valdez, M. (2016). Benchmarking languages for evolutionary algorithms. En G. Squillero y P. Burelli (Eds.), *Applications of evolutionary computation* (Lecture Notes in Computer Science, Vol. 9598, pp. 27-41). Springer. [https://doi.org/10.1007/978-3-319-31153-1\\_3](https://doi.org/10.1007/978-3-319-31153-1_3)

Merelo-Guervós, J. J., Blancas-Álvarez, I., Castillo, P. A., Romero, G., Rivas, V. M., García-Valdez, M., Hernández-Águila, A. y Román, M. (24-29 de julio de 2016). A comparison of implementations of basic evolutionary algorithm operations in different languages. En *2016 IEEE Congress on Evolutionary Computation (CEC)* (pp. 1602-1609). IEEE. <https://doi.org/10.1109/CEC.2016.7743980>

Merelo-Guervós, J. J., Blancas-Álvarez, I., Castillo, P. A., Romero, G., García-Sánchez, P., Rivas, V. M., García-Valdez, M., Hernández-Águila, A. y Román, M. (2017). Ranking programming languages for evolutionary algorithm operations. En G. Squillero y K. Sim (Eds.), *Applications of evolutionary computation* (Lecture Notes in Computer Science, Vol. 10199, pp. 689-704). Springer. [https://doi.org/10.1007/978-3-319-55849-3\\_44](https://doi.org/10.1007/978-3-319-55849-3_44)

Merelo-Guervós, J. J., Romero López, G. y García-Valdez, M. (2025). Measuring energy consumption of BBOB fitness functions. En P. García-Sánchez, E. Hart y S. L. Thomson (Eds.), *Applications of evolutionary computation* (Lecture Notes in Computer Science, Vol. 15613, pp. 240-254). Springer. [https://doi.org/10.1007/978-3-031-90065-5\\_15](https://doi.org/10.1007/978-3-031-90065-5_15)

Moore, S. V. (2002). A comparison of counting and sampling modes of using performance monitoring hardware. En B. Monien y R. Feldmann (Eds.), *Euro-Par 2002 Parallel Processing: 8th International Euro-Par Conference, Paderborn, Germany, August 27-30, 2002* (Lecture Notes in Computer Science, Vol. 2400, pp. 904-912). Springer. [https://doi.org/10.1007/3-540-46080-2\\_95](https://doi.org/10.1007/3-540-46080-2_95)

Salto, C., Minetti, G., Alba, E. y Luque, G. (2018). Developing genetic algorithms using different MapReduce frameworks: MPI vs. Hadoop. En F. Herrera, S. Damas, R. Montes, S. Alonso, Ó. Cordón, A. González y A. Troncoso (Eds.), *CAEPIA 2018* (Lecture Notes in Artificial Intelligence, Vol. 11160, pp. 262-272). Springer. [https://doi.org/10.1007/978-3-030-00374-6\\_25](https://doi.org/10.1007/978-3-030-00374-6_25)

Scott, E. O. y Luke, S. (13-17 de julio de 2019). *ECJ at 20: Toward a general metaheuristics toolkit*. Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19) (pp. 1391-1398). Association for Computing Machinery. <https://doi.org/10.1145/3319619.3326865>

Stoico, V., Dragomir, A. C. y Lago, P. (2025). An empirical study on the performance and energy usage of compiled Python code. En *Proceedings of The 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025)* (pp. 1-11). ACM.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M. y Fonseca, V. G. da. (2003). Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2), 117-132.  
<https://doi.org/10.1109/TEVC.2003.810758>

## **8. Anexos**

### **8.1. Anexo I: Glosario de términos**

#### ***AE***

Metaheurística que optimiza mediante población, selección, cruce y mutación, imitando la evolución natural.

#### ***BENCHMARK***

Conjunto de problemas estándar de referencia que se utiliza para evaluar y comparar el rendimiento de algoritmos en términos de precisión, velocidad y consumo de recursos.

#### ***BIT-FLIP***

Operador de mutación en representaciones binarias que invierte el valor de uno o varios bits de un individuo para explorar el espacio de búsqueda.

#### ***BOBB***

Conjunto de funciones de prueba estándar diseñado para evaluar y comparar algoritmos de optimización sin conocimiento explícito del problema, midiendo rendimiento en términos de precisión y llamadas a la función.

#### ***BYTE-CODE***

Forma intermedia de código generada tras la compilación de un lenguaje que se ejecuta en una máquina virtual mediante interpretación o compilación *Just-In-Time*.

#### ***CARBON IMPACT STATEMENTS***

Informe que acompaña a un experimento computacional y detalla su energía consumida, intensidad de carbono y emisiones equivalentes de CO<sub>2</sub>, permitiendo evaluar y comparar la huella ambiental del trabajo.

#### ***CHILLERS***

Equipos de refrigeración que extraen calor de un fluido mediante un ciclo de compresión o absorción.

#### ***CPU***

Procesador general de propósito que ejecuta instrucciones secuenciales; coordina y controla las operaciones lógicas y aritméticas del sistema.

## ***CROSS-PLATFORM***

Característica de software o métodos que funcionan de manera equivalente en diferentes sistemas operativos, arquitecturas o entornos de ejecución sin modificar el código fuente.

## ***CUDA***

(*Compute Unified Device Architecture*) Plataforma y API de NVIDIA que permite la ejecución de cómputo paralelo en GPUs mediante extensiones de C/C++, facilitando el desarrollo de aplicaciones aceleradas por hardware gráfico.

## ***DIFFERENTIAL EVOLUTION***

Algoritmo evolutivo para optimización continua que genera nuevas soluciones sumando la diferencia de dos individuos a un tercero, seguido de cruzamiento y selección, facilitando la exploración y explotación eficaz del espacio de búsqueda.

## ***DISPOSITIVO EMBEBIDO***

Sistema computacional dedicado, integrado en un producto o equipo específico, con hardware y software optimizados para una única función o conjunto limitado de tareas.

## ***ENSAYOS CROSS-PLATAFORMA***

Experimentos diseñados para ejecutar el mismo algoritmo en distintos hardware, sistemas operativos o lenguajes, con el fin de comparar su rendimiento y consumo en condiciones equivalentes.

## ***EVOLUTIONARY PROGRAMMING***

Variante de algoritmo evolutivo centrada en la mutación de cada individuo para generar descendencia, sin emplear cruces.

## ***FITNESS***

Valor numérico que cuantifica la calidad de una solución; guía la selección en un AE.

## ***FLAGS***

Parámetros de línea de comandos que habilitan o deshabilitan funcionalidades específicas de un programa.

## ***FRAMEWORK***

Conjunto de bibliotecas y herramientas que provee una estructura base para desarrollar y ejecutar algoritmos.

## **GA**

Algoritmo evolutivo que emplea operadores de selección, cruce y mutación sobre una población de individuos codificados como cadenas inspirándose en la genética para explorar el espacio de soluciones y optimizar una función objetivo.

## **GAUSSIAN PROCESSES**

Modelos probabilísticos no paramétricos que definen una distribución sobre funciones, usando una media y una función de covarianza para predecir valores continuos con estimaciones de incertidumbre.

## **GiB**

Unidad de información que equivale a  $2^{30}$  bytes (1.073.741.824 bytes), utilizada para expresar capacidades de almacenamiento y memoria con base binaria.

## **GREEN ALGORITHMS**

Conjunto de métodos y principios que cuantifican el consumo energético y las emisiones de CO<sub>2</sub> de procesos computacionales, promoviendo el diseño y la ejecución de código con mínima huella ambiental.

## **HALL OF FAME**

Estructura que registra los mejores individuos encontrados durante la evolución.

## **HVAC**

(*Heating, Ventilation, and Air Conditioning*) Sistema integrado que controla la calefacción, ventilación y climatización de espacios cerrados, regulando temperatura, humedad y calidad del aire para mantener condiciones ambientales óptimas.

## **IBEA**

(*Indicator-Based Evolutionary Algorithm*) Algoritmo evolutivo basado en indicadores para asignar un valor de aptitud a cada solución, guiando la selección y supervivencia hacia mejorar indicadores específicos en la frontera de Pareto.

## **IN-PLACE**

Estrategia de ejecución en la que los datos se modifican directamente en la misma memoria original, sin utilizar estructuras auxiliares adicionales.

## **INLINING**

Técnica de optimización donde el compilador reemplaza una llamada a función con el cuerpo de la función, eliminando la sobrecarga de salto y permitiendo optimizaciones adicionales.

## **JVM**

Entorno de ejecución (Maquina Virtual de Java) que interpreta o compila *Just-In-Time byte-code* de Java, gestionando memoria, carga de clases y optimizaciones en tiempo de ejecución.

## **MACHINE LEARNING**

Área de la IA que diseña algoritmos capaces de inferir patrones a partir de datos y mejorar su desempeño en una tarea sin reglas programadas explícitamente.

## **MÉTODO LAGRANGIANO**

Técnica de optimización que introduce multiplicadores de Lagrange para transformar problemas con restricciones en un problema sin restricciones, agregando las restricciones ponderadas directamente a la función objetivo.

## **MINERÍA BLOCKCHAIN**

Proceso computacional que valida y registra transacciones en una cadena de bloques resolviendo complejos problemas criptográficos.

## **MONITORING**

Proceso continuo de recopilación y registro de métricas operativas durante la ejecución de un sistema o algoritmo para evaluar su comportamiento y detectar anomalías.

## **MOODY**

Estrategia de optimización multiobjetivo que adapta dinámicamente las preferencias o pesos de los objetivos para responder a cambios en el entorno o la función de aptitud, empleando operadores evolutivos y un mecanismo de detección de variaciones en el paisaje de búsqueda.

## **MPI**

(*Message Passing Interface*) Estándar API para programación paralela en sistemas de memoria distribuida, que define funciones para enviar y recibir mensajes entre procesos en distintos nodos.

## **NSGA-II**

Algoritmo genético de ordenamiento no dominado multiobjetivo que emplea clasificación por niveles de no dominancia, selección basada en distancia de enjambre y operador de crowded-comparison para mantener diversidad y convergencia en la población.

## ***NSGA-III***

Algoritmo genético de ordenamiento no dominado multiobjetivo Para problemas con más de dos objetivos.

## ***OFFSPRING***

Individuos descendientes generados a partir de la población actual mediante operadores genéticos.

## ***OPENMP***

Especificación de API basada en directivas para programación paralela en sistemas de memoria compartida.

## ***OVERHEAD***

Consumo adicional de recursos asociado a tareas auxiliares que no contribuyen directamente a la evaluación de la solución.

## ***PSO***

Algoritmo de optimización basado en población que simula el comportamiento social de bandadas o cardúmenes. Cada partícula ajusta su posición guiada por su mejor solución personal y la mejor global, convergiendo hacia óptimos.

## ***RAPL***

Interfaz de Intel que expone contadores hardware de energía y potencia en tiempo real para CPU, DRAM y otros dominios, útil para medir consumo de forma granular.

## ***SBX***

(*Simulated Binary Crossover*) Operador de cruce para codificaciones reales que genera hijos simulando el comportamiento de un cruce binario, generando dos descendientes a partir de dos padres con distribución controlada por un parámetro de dispersión.

## ***SCHEDULING***

Asignación planificada de tareas o trabajos a recursos computacionales definiendo el orden y la duración de ejecución.

## ***SERVIDORES BLADE***

Hardware modular en el que múltiples servidores compactos se instalan en un mismo chasis compartido, aprovechando recursos comunes para ahorrar espacio y facilitar la escalabilidad.

## **SPEA2**

Algoritmo evolutivo de Pareto de fuerza 2 Que asigna a cada individuo un “índice de fuerza” según cuántas soluciones domina y calcula densidad local mediante distancia a los vecinos, para seleccionar y archivar soluciones de Pareto.

## **SPEED-UP**

Medida de mejora en el rendimiento de un algoritmo al ejecutarse en paralelo o en hardware más potente, calculada como el cociente entre el tiempo de ejecución en la configuración base y el tiempo en la nueva configuración.

## **SURROGATE MODELS**

Modelos aproximados que emulan el comportamiento de funciones costosas de evaluar. Se construyen a partir de muestras previas y se usan para predecir resultados, reduciendo el número de evaluaciones directas durante la optimización.

## **TOOLBOXES**

Conjuntos de bibliotecas y utilidades especializadas diseñadas para un dominio concreto.

## **TOOLKITS**

Colección de paquetes, bibliotecas y utilidades que proporcionan funcionalidades específicas.

## **TRACKER**

Herramienta de software que registra y almacena métricas en tiempo real durante la ejecución de un proceso.

## **WARM-UP**

Periodo inicial de ejecución durante el cual el sistema o hardware alcanza un estado operativo estable.

## **WHISKERS**

Elementos de un diagrama de caja que representan la variabilidad de los datos fuera del rango intercuartílico, extendiéndose desde los cuartiles hasta el valor máximo y mínimo dentro de un límite específico.

## **WRAPPERS**

Método que evalúa subconjuntos de atributos usando el rendimiento de un modelo predictivo.

## 8.2. Anexo II: Implementaciones

En este anexo se presentan los fragmentos de código más relevantes para la configuración y ejecución de cada uno de los cuatro *benchmarks* utilizados (OneMax, Sphere, Rosenbrock y Schwefel) en los cuatro *frameworks* estudiados (DEAP, Inspyred, ECJ y ParadisEO). No se incluye el código completo de cada proyecto, sino únicamente las secciones necesarias para comprender cómo se ha parametrizado la función de evaluación, la generación de población, los operadores genéticos y el control de tiempo de ejecución (2 min) en cada entorno. El código desarrollado para este Proyecto Final de Grado se encuentra disponible en el siguiente repositorio: <https://github.com/fjluquehdez/Proyecto-Fin-Grado.git>

### 8.2.1. OneMax

En este apartado se muestra cómo se implementa el *benchmark* OneMax (contar el número de unos en un individuo binario) en cada *framework*. Para cada caso, se extraen y explican los fragmentos que definen la función de evaluación, la representación de la población, la configuración de los operadores de cruce y mutación, y la lógica que interrumpe la ejecución a los 120 segundos.

#### 8.2.1.1. DEAP

Python

```
# Definición de tipos y creación del individuo
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)

# Registro de atributos, individuo y población
toolbox = base.Toolbox()
toolbox.register("attr_bool", random.randint, 0, 1)
toolbox.register("individual", tools.initRepeat, creator.Individual,
toolbox.attr_bool, tamaño_individuo)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Registro de operadores y evaluación
toolbox.register("evaluate", lambda ind: (sum(ind) / len(ind) * 100,))
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=mutacion)
toolbox.register("select", tools.selTournament, tournsize=2)

# Creación de población inicial
population = toolbox.population(n=poblacion)
-----
# Bucle evolutivo con límite de 120 segundos
start_time = time.time()
for generation in range(1000000):
    # Generar descendientes clonando la población
    offspring = list(map(toolbox.clone, population))
```

```

# Aplicar cruce de dos puntos con probabilidad 'cruce'
for c1, c2 in zip(offspring[::2], offspring[1::2]):
    if random.random() < cruce:
        toolbox.mate(c1, c2)
        del c1.fitness.values
        del c2.fitness.values

# Aplicar mutación bit-flip con probabilidad 'mutacion'
for mutant in offspring:
    if random.random() < mutacion:
        toolbox.mutate(mutant)
        del mutant.fitness.values

# Evaluar solo a los individuos cuyo fitness ya no sea válido
invalid = [ind for ind in offspring if not ind.fitness.valid]
for ind, fit in zip(invalid, map(toolbox.evaluate, invalid)):
    ind.fitness.values = fit

# Seleccionar la siguiente generación (misma dimensión de población)
population[:] = toolbox.select(offspring, len(population))

# Comprobar parada por fitness completo o 120 segundos transcurridos
total_ones = sum(sum(ind) for ind in population)
if (total_ones / (poblacion * tamano_individuo) * 100) == 100 or
time.time() - start_time >= 120:
    break

```

En el fragmento de código de DEAP para OneMax, lo primero que se hace es definir cómo se representarán los individuos y su *fitness*. Con `creator.create("FitnessMax", base.Fitness, weights=(1.0,))`, se establece un tipo de *fitness* que se desea maximizar, y a partir de esa definición se crea el tipo `Individual` como una lista de bits que porta ese atributo de *fitness*. Así, cada individuo será una lista de ceros y unos con un campo `.fitness` al que se asignará su *fitness*.

A continuación, se crea un objeto `toolbox` que agrupa todas las funciones necesarias para generar la población y aplicar los operadores genéticos. Primero, `toolbox.register("attr_bool", random.randint, 0, 1)` indica que cada gen del individuo se genera con un valor aleatorio entre 0 y 1. Luego, con `toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_bool, tamano_individuo)`, se dice que un individuo se construye repitiendo la función `attr_bool` tantas veces como marque `tamano_individuo`, de modo que cada individuo sea una lista de longitud fija llena de valores 0 o 1. Finalmente, `toolbox.register("population", tools.initRepeat, list, toolbox.individual)` define que la población entera es simplemente una lista de individuos creados por

`toolbox.individual`, y la llamada `population = toolbox.population(n=poblacion)` genera la cantidad de individuos especificada (poblacion puede ser 64, 1.024 o 16.384 según la configuración).

En el mismo toolbox se registran los operadores genéticos y la función de evaluación. La línea `toolbox.register("evaluate", lambda ind: (sum(ind) / len(ind) * 100,))` implementa la función OneMax: calcula el porcentaje de bits “1” en la lista y lo devuelve como una tupla (DEAP requiere que el valor de *fitness* siempre esté en un iterable). Para el cruce se usa `tools.cxTwoPoint`, que intercambia dos segmentos entre dos individuos, y para la mutación `tools.mutFlipBit`, que invierte cada bit con probabilidad `mutacion`. La selección se realiza con `tools.selTournament` de tamaño 2, de modo que, en cada torneo, se eligen dos candidatos al azar y se escoge el mejor entre ellos.

Una vez configurada la representación y los operadores, se lanza la primera generación con `population = toolbox.population(n=poblacion)`. A partir de ahí, comienza el bucle evolutivo que se repetirá hasta 1.000.000 de generaciones o hasta que se cumpla el criterio de parada. Al iniciar el bucle se marca el instante inicial con `start_time = time.time()`. En cada iteración, los descendientes se crean clonando la población actual (`offspring = list(map(toolbox.clone, population))`). Luego, para cada par consecutivo de individuos en `offspring`, se aplica el cruce de dos puntos si un número aleatorio es menor que la probabilidad cruce; tras el cruce se elimina el valor de *fitness* para forzar la reevaluación posterior. A continuación, cada descendiente se somete a mutación bit-flip con probabilidad `mutacion` y, si muta, también se invalida su *fitness*.

Después de mutar a todos los descendientes, solo se evalúan aquellos cuyo *fitness* ya no sea válido; esto se consigue filtrando con `invalid = [ind for ind in offspring if not ind.fitness.valid]` y luego aplicando `toolbox.evaluate` únicamente a esos individuos. Con esto se minimiza el número de llamadas a la función de evaluación, pues los individuos que no han cambiado conservan su *fitness* anterior. La selección de la siguiente generación se realiza con `population[:] = toolbox.select(offspring, len(population))`, de modo que la población resultante mantenga el mismo tamaño pero esté compuesta por los descendientes elegidos.

Para decidir si detener la evolución, en cada iteración se calcula el *fitness* promedio de la población como el porcentaje total de bits “1” (`total_ones = sum(sum(ind) for ind in population)` dividido por el producto de población y tamaño de individuo). Si ese promedio es igual a 100, significa que todos los individuos tienen todos los bits en “1”. Alternativamente, si han transcurrido 120 segundos desde `start_time`, también se rompe el bucle. De esta manera, el algoritmo se asegura de no exceder el límite temporal establecido.

### 8.2.1.2. Inspyred

Python

```
# Función de fitness (OneMax normalizado 0-1)
def onemax_fitness(candidates, args):
    return [sum(candidate) / INDIVIDUAL_SIZE for candidate in candidates]

# Generador de individuos binarios
def generator(random, args):
    return [random.choice([0, 1]) for _ in range(args.get('individual_size',
INDIVIDUAL_SIZE))]

-----
# Clase para almacenar las estadísticas de evolución
class EvolutionStats:
    def __init__(self):
        self.initial_fitness = 0
        self.best_fitness = 0
        self.gen_best_fitness = 0
        self.current_generation = 0
        self.termination_cause = "timeout"

-----
# Configurar el objeto GA de Inspyred
global stats
stats = EvolutionStats()
rand = random.Random()
rand.seed(time.time())
ea = ec.GA(rand)
ea.selector = inspyred.ec.selectors.tournament_selection
ea.variator = [inspyred.ec.variators.uniform_crossover,
inspyred.ec.variators.bit_flip_mutation]
ea.replacer = inspyred.ec.replacers.generative_replacement
ea.terminator = time_generation_termination
ea.observer = stats_observer

-----
# Llamada a evolution
final_pop = ea.evolve(generator=generator,
                       evaluator=onemax_fitness,
                       pop_size=pop_size,
                       maximize=True,
                       bounder=inspyred.ec.Bounder(0, 1),
                       max_generations=MAX_GENERATIONS,
                       start_time=start_time,
                       max_time=MAX_TIME_SECONDS,
                       num_selected=pop_size,
                       individual_size=INDIVIDUAL_SIZE,
                       crossover_rate=cx_rate,
                       mutation_rate=MUTATION_RATE,
                       tournament_size=2
                      )
```

La implementación de OneMax en Inspyred comienza definiendo dos funciones clave. Primero, `onemax_fitness` recibe una lista de candidatos (cada uno es una lista de bits) y, para cada uno, calcula el porcentaje de unos normalizando la suma sobre el tamaño total `INDIVIDUAL_SIZE`. Este valor devuelto sirve como el *fitness* que Inspyred intentará maximizar. Segundo, la función `generator` construye cada individuo eligiendo aleatoriamente ceros o unos hasta completar la longitud deseada, que se pasa en el argumento `individual_size`.

A continuación se instancia el objeto GA con `ea = ec.GA(rand)`, usando un generador de números aleatorios `rand` con semilla variable para cada ejecución. El selector se fija como torneo con `ea.selector = tournament_selection`, y los variadores combinan cruce uniforme (`uniform_crossover`) con una mutación implementada en `style_mutation`. Esa mutación personalizada primero decide, con probabilidad `mutation_individual_rate`, si un individuo debe mutar; y en caso afirmativo aplica `bit_flip_mutation` con probabilidad `mutation_bit_rate` por bit. El reemplazo se configura con `generational_replacement`, lo que indica que toda la población se renueva cada generación. Para controlar cuándo detener, `ea.terminator = time_generation_termination` evalúa tanto el número de generaciones como el tiempo transcurrido, cortando la evolución si algún límite se alcanza. Finalmente, `ea.observer = stats_observer` recoge métricas en cada generación, como el mejor *fitness* actual y la generación en que ocurrió.

La llamada a `ea.evolve(...)` consolida toda la configuración: se le pasan el generador, el evaluador y parámetros de población como `pop_size`, `max_generations` y `max_time`. También recibe `individual_size` para saber cuántos bits produce el generador, y `crossover_rate`, `mutation_individual_rate` y `mutation_bit_rate` para ajustar la probabilidad de aplicar cada variador. El parámetro `maximize=True` indica que el algoritmo busca maximizar el valor devuelto por `onemax_fitness`. Además, `num_selected=pop_size` mantiene constante el tamaño de la población en cada iteración, y `bounder=Bounder(0,1)` garantiza que los bits se mantengan en {0,1} tras el cruce.

Inspyred internamente gestiona el bucle evolutivo. En cada generación, crea la población inicial llamando a `generator` repetidamente, evalúa todos los individuos con `onemax_fitness`, selecciona padres, genera descendientes con cruce y mutación, reevalúa solo los que cambiaron y aplica el reemplazador. El algoritmo finaliza automáticamente cuando se cumple `time_generation_termination`: el algoritmo comprueba los contadores de tiempo y generaciones y detiene la ejecución al superar 120 segundos o el número máximo de generaciones. Con esto, Inspyred simplifica mucho el código del usuario, relegando toda la lógica de bucle, evaluación y selección interna a su propio motor.

### 8.2.1.3. ECJ

```
Java
// RunOneMax.java
public class RunOneMax {
    public static void main(String[] args) {
        String[] ecjArgs = {"-file", "onemax.params"};
        Evolve.main(ecjArgs);
    }
}

-----
// OneMaxProblem.java (método evaluate)
BitVectorIndividual individual = (BitVectorIndividual) ind;
int sum = 0;
for (boolean gene : individual.genome) {
    if (gene) sum++;
}
((SimpleFitness) individual.fitness).setFitness(state, sum, sum ==
individual.genome.length);
individual.evaluated = true;
-----
// TimeConstrainedEvaluator.java (comprobación de tiempo)
long currentTime = System.currentTimeMillis();
if (currentTime - startTime > MAX_TIME) {
    timeLimitReached = true;
}
```

```
Unset
# onemax.params
pop.subpop.0.size = 1024
pop.subpop.0.species = ec.vector.BitVectorSpecies
pop.subpop.0.species.genome-size = 1024
select.tournament.size = 3

pop.subpop.0.species.pipe.source.0 = ec.vector.breed.VectorCrossoverPipeline
pop.subpop.0.species.pipe.source.0.prob = 0.2

pop.subpop.0.species.pipe.source.1 = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.1.prob = 0.1

eval.problem = OneMaxProblem
eval.max-time = 120
eval.max-generations = 1000000
```

El punto de entrada de ECJ se encuentra en `RunOneMax.java`, donde se invoca `Evolve.main(...)` pasándole el nombre del archivo de parámetros (`onemax.params`). Gracias a esta llamada, ECJ lee automáticamente toda la configuración (tamaño de población, operadores, criterios de parada, etc.) sin necesidad de código adicional en Java para cada experimento, lo que simplifica la replicación de distintas configuraciones.

La clase `OneMaxProblem.java` implementa el problema OneMax extendiendo `Problem` e implementando `SimpleProblemForm`. En su método `evaluate`, se asegura de que el individuo sea un `BitVectorIndividual`, recorre su vector contando la cantidad de valores true (bits “1”) y finalmente asigna ese conteo al *fitness* de tipo `SimpleFitness`. Si el tamaño del genoma completo está lleno de unos, se marca como solución óptima. Esta lógica es todo lo que ECJ necesita para valorar cada individuo en términos de cuántos bits en 1 contiene, y se ejecuta automáticamente para cada miembro de la población.

El manejo de la detención por tiempo corre a cargo de `TimeConstrainedEvaluator.java`, que extiende `SimpleEvaluator`. Durante la configuración, guarda el instante de inicio (`startTime`). Luego, en `evaluatePopulation`, tras evaluar toda la población, compara el tiempo transcurrido con la constante `MAX_TIME` (120.000 ms). Si se excede, marca `timeLimitReached = true`. ECJ llamará a esta evaluación cada generación, y cuando detecta que `timeLimitReached` es cierto, detiene la evolución. Así, no hace falta un bucle explícito en el usuario; ECJ interrumpe por sí mismo al cumplirse el límite de dos minutos.

Finalmente, el archivo `onemax.params` engloba la mayoría de parámetros importantes. Allí se define que la población tiene un único subgrupo (`pop.subpop.0.size`) cuyo tamaño es 64, 1.024 o 16.384, en función de la configuración a ejecutar. Se especifica que la especie es `BitVectorSpecies` con genoma de longitud 1024, lo que crea individuos binarios de esa longitud automáticamente. La línea `select.tournament.size = 2` indica que la selección será por torneo de dos competidores. Bajo “*Genetic Operators*” se registra `VectorCrossoverPipeline` con la probabilidad de cruce que corresponda y `VectorMutationPipeline` con probabilidad de mutación 0.1. El parámetro `eval.problem = OneMaxProblem` enlaza la evaluación a la clase Java que cuenta los unos, y `eval.max-time = 120` fija el límite de tiempo en 120 segundos. De esta forma, el archivo `.params` controla la totalidad de la evolución sin que el usuario tenga que escribir un solo bucle de generación: ECJ se encarga de gestionar poblaciones, operadores, evaluación y terminación basándose en estas líneas.

#### 8.2.1.4. ParadisEO

C/C++

```
// Definición del individuo: vector de bits con fitness a maximizar
class OneMax : public EO<eoMaximizingFitness> {
public:
    vector<bool> bits;
    OneMax(size_t n) : bits(n, false) {}
    void printOn(ostream &os) const override { for (bool b : bits) os << b;
}
    void readFrom(istream &is) override {
        for (size_t i = 0; i < bits.size(); ++i) {
            char c; is >> c; bits[i] = (c == '1');
        }
    }
};

-----  

// Evaluador: cuenta el número de 1s en el individuo
class OneMaxEval : public eoEvalFunc<OneMax> {
public:
    void operator()(OneMax &ind) override {
        int count = 0;
        for (bool b : ind.bits) if (b) count++;
        ind.fitness(count);
    }
};

-----  

// Inicializador: genera bits aleatorios con probabilidad 0.5
class OneMaxInit : public eoInit<OneMax> {
public:
    OneMaxInit(size_t n) : n(n) {}
    void operator()(OneMax &ind) override {
        ind.bits.resize(n);
        for (size_t i = 0; i < n; ++i)
            ind.bits[i] = (eo::rng.uniform() < 0.5);
    }
private:
    size_t n;
};

-----  

// Operador de cruce
class OnePointCrossover : public eoQuadOp<OneMax> {
public:
    bool operator()(OneMax &p1, OneMax &p2) override {
        size_t n = p1.bits.size();
        size_t point = eo::rng.random(n);
        for (size_t i = point; i < n; ++i)
            swap(p1.bits[i], p2.bits[i]);
    }
};
```

```

        return true;
    }
};

-----
// Operador de mutación bit-flip
class BitFlipMutation : public eoMonOp<OneMax> {
public:
    BitFlipMutation(double rate) : mutationRate(rate) {}
    bool operator()(OneMax &ind) override {
        bool changed = false;
        for (size_t i = 0; i < ind.bits.size(); ++i) {
            if (eo::rng.uniform() < mutationRate) {
                ind.bits[i] = !ind.bits[i];
                changed = true;
            }
        }
        return changed;
    }
private:
    double mutationRate;
};

-----
// Bucle de evolución con límite de tiempo (120 s)
auto start = chrono::steady_clock::now();
for (gen = 1; gen <= nGenerationsMax && keepRunning; gen++) {
    eoPop<OneMax> newPop;
    while (newPop.size() < popSize) {
        OneMax parent1 = select(pop);
        OneMax parent2 = select(pop);
        if (eo::rng.uniform() < pc) crossover(parent1, parent2);
        mutation(parent1); mutation(parent2);
        eval(parent1); eval(parent2);
        newPop.push_back(parent1);
        if (newPop.size() < popSize) newPop.push_back(parent2);
    }
    pop = newPop;
    auto now = chrono::steady_clock::now();
    if (bestFitness >= nbits) { keepRunning = false; stopReason = "Fitness alcanzado"; }
    else if (chrono::duration_cast<chrono::seconds>(now - start).count() >= 120) {
        keepRunning = false; stopReason = "Timeout de 2 minutos";
    }
}

```

En ParadisEO, el tipo de individuo para OneMax se define mediante la clase `OneMax`, que hereda de `EO<eoMaximizingFitness>`. Su atributo principal es `vector<bool> bits`, de

`longitud nbits = 1024`. La implementación de `printOn` y `readFrom` permite a ParadisEO escribir y leer cada individuo en un flujo de salida o entrada, interpretando cada carácter ‘1’ o ‘0’ como un valor booleano.

La clase `OneMaxEval` extiende `eoEvalFunc<OneMax>` y su método `operator()` recorre el vector `bits` contando cuántos valores `true` contiene el individuo. Ese conteo se asigna como *fitness* a maximizar mediante `ind.fitness(count)`. En cada llamada, ParadisEO usa esta función para evaluar el valor objetivo de cada individuo.

Para inicializar la población, `OneMaxInit` implementa `eoInit<OneMax>`. En su `operator()`, redimensiona `bits` a tamaño `n` y asigna cada bit a `true` con probabilidad 0.5 (usando `eo::rng.uniform() < 0.5`). Durante la creación de la población, el código principal invoca este inicializador en un bucle que crea `popSize` instancias de `OneMax`, garantizando que todos arranquen con bits aleatorios.

Los operadores genéticos se definen en dos clases: `OnePointCrossover`, que implementa `eoQuadOp<OneMax>`, y `BitFlipMutation`, que extiende `eoMonOp<OneMax>`. En el cruce de un punto, se elige un índice aleatorio `point` y se intercambian todos los bits desde `point` hasta el final entre los dos padres. En la mutación bit-flip, cada bit de un individuo se invierte con probabilidad `mutationRate`. ParadisEO aplica estos operadores directamente sobre copias de los padres para generar descendientes.

El bucle principal arranca midiendo el instante inicial con `auto start = chrono::steady_clock::now()`. Cada generación construye `newPop` seleccionando repetidamente dos padres mediante `select(pop)`, que en este caso es un selector por torneo de tamaño 2 (`eoDetTournamentSelect<OneMax>`). A cada par se le aplica el cruce con probabilidad `pc` y luego la mutación de cada individuo. Tras evaluar ambos descendientes con `eval(...)`, se agregan a `newPop` hasta completar `popSize`. Una vez construida la población de la siguiente generación, se reemplaza `pop` por `newPop`.

Después de cada generación, el código comprueba dos condiciones para detener la ejecución: si el mejor *fitness* (`bestFitness`) ha alcanzado `nbits` (es decir, todos los bits son 1) se marca `keepRunning = false` y `stopReason = "Fitness alcanzado"`. En caso contrario, calcula el tiempo transcurrido con `chrono::duration_cast<chrono::seconds>(now - start).count()`. Si ese valor es al menos 120 segundos, se detiene asignando `stopReason = "Timeout de 2 minutos"`. De este modo, ParadisEO interrumpe el bucle evolutivo de forma automática al cumplirse el tiempo máximo o al alcanzar el óptimo.

### 8.2.2. Sphere

En este apartado se muestra cómo se configura el *benchmark* Sphere (suma de cuadrados de un vector real) en cada *framework*. El código esencial incluye la definición de la función de evaluación, la construcción de individuos de tipo real, la selección de operadores genéticos adecuados (SBX para cruce y mutación polinómica acotada) y el control de tiempo de ejecución mediante un bucle que interrumpe tras 120 segundos.

#### 8.2.2.1. DEAP

```
Python
# Definición de la función Sphere: suma de cuadrados normalizada a
# porcentaje
def evaluate(ind):
    raw = sum(x * x for x in ind)
    return ((1 - raw / F_MAX))

# Configuración de tipos y registro
creator.create('FitnessMax', base.Fitness, weights=(1.0,))
creator.create('Individual', list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()
toolbox.register('attr_float', random.uniform, LOW, UP)
toolbox.register('individual', tools.initRepeat,
                 creator.Individual, toolbox.attr_float, tamaño_individuo)
toolbox.register('population', tools.initRepeat, list, toolbox.individual)

toolbox.register('mate', tools.cxSimulatedBinaryBounded,
                low=LOW, up=UP, eta=20.0)
toolbox.register('mutate', tools.mutPolynomialBounded,
                low=LOW, up=UP, eta=20.0, indpb=mutacion)
toolbox.register('select', tools.selTournament, tournsize=2)

# Inicialización y evaluación de la población
pop = toolbox.population(n=poblacion)
for ind in pop:
    if tiempo_excedido(t_start, max_time): break
    ind.fitness.values = toolbox.evaluate(ind)
-----
# Bucle evolutivo
start = time.time()
while not tiempo_excedido(start, max_time):
    # Cruce y mutación
    offspring = list(map(toolbox.clone, pop))
    for i in range(0, len(offspring), 2):
        if tiempo_excedido(start, max_time): break
        c1, c2 = offspring[i], offspring[i+1]
        if random.random() < cruce:
```

```

        toolbox.mate(c1, c2); del c1.fitness.values; del
c2.fitness.values
    for m in offspring:
        if tiempo_excedido(start, max_time): break
        if random.random() < mutacion:
            toolbox.mutate(m); del m.fitness.values

# Evaluación de descendientes y selección
for ind in offspring:
    if tiempo_excedido(start, max_time): break
    if not ind.fitness.valid:
        ind.fitness.values = toolbox.evaluate(ind)
    if tiempo_excedido(start, max_time): break

pop[:] = toolbox.select(offspring, len(pop))

# Verifica convergencia: fitness ≥ 100 %
valid = [ind for ind in pop if ind.fitness.valid]
if valid and tools.selBest(valid, 1)[0].fitness.values[0] >= 100.0:
    motivo = 'convergence'
    break

```

En este fragmento de código de DEAP para el *benchmark* Sphere, lo primero que aparece es la función de evaluación `evaluate`, que calcula la suma de los cuadrados de cada componente del individuo y la normaliza para convertirla en un porcentaje de *fitness*. Concretamente, la variable `raw` acumula  $x_i^2$  para cada valor real  $x$  en la lista `ind`, y luego ese resultado se divide por `F_MAX` (el valor máximo teórico que ocurre cuando todos los  $x_i$  están en el límite superior). Al restar `raw/F_MAX` de 1 y se obtiene un *fitness* en el que 1 equivale a la mejor solución posible, y valores cercanos a 0 indican candidatos muy alejados del óptimo.

A continuación, se definen los tipos que DEAP va a utilizar para representar individuos y su *fitness*. Con `creator.create('FitnessMax', base.Fitness, weights=(1.0,))` se especifica que se pretende maximizar ese valor porcentual de *fitness*, y después, con `creator.create('Individual', list, fitness=creator.FitnessMax)`, se declara que cada individuo será una lista de números reales que lleva incorporado un campo `.fitness`. Esto es fundamental porque DEAP crea internamente las estructuras necesarias para almacenar el *fitness* junto a la representación del individuo, lo que simplifica mucho el manejo posterior.

El objeto `toolbox` concentra toda la lógica para generar individuos y aplicar operadores. Primero, `toolbox.register('attr_float', random.uniform, LOW, UP)` indica que cada gen vendrá de una llamada a `random.uniform(-5.12, 5.12)`, con lo cual cada componente del vector real se sitúa de manera uniforme en ese intervalo. A partir de esa función, `toolbox.register('individual', tools.initRepeat, creator.Individual,`

`toolbox.attr_float,tamaño_individuo)` crea un individuo repitiendo `tamaño_individuo` veces la llamada a `attr_float`, de modo que cada lista de longitud fija (1.024) es construida con valores continuos aleatorios. Por último, `toolbox.register('population', tools.initRepeat, list, toolbox.individual)` configura la población completa como una lista de esos individuos, y, al invocar `pop = toolbox.population(n=poblacion)`, se obtiene un conjunto de `poblacion` vectores reales ya listos para ser evaluados.

Una vez generada la población inicial, el código recorre cada individuo y, siempre que no se supere el límite de tiempo, asigna su *fitness* con `ind.fitness.values = toolbox.evaluate(ind)`. De este modo, justo antes de iniciar el bucle evolutivo, cada elemento de `pop` tiene un valor asociado que indica qué tan buena es su combinación de coordenadas para minimizar la suma de cuadrados.

El bucle evolutivo se maneja con un `while` que se ejecuta mientras no se agote el tiempo de 120 segundos, evaluando continuamente mediante la función `tiempo_excedido`. Al entrar en cada iteración, se clona la población actual para producir los descendientes en `offspring`. Cuando dos individuos consecutivos se procesan, se aplica el operador SBX (*Simulated Binary Crossover*) con probabilidad cruce; esto mezcla dos vectores reales dentro de los límites [-5.12,5.12], garantizando que el resultado permanezca válido para el problema de Sphere. Después, cada descendiente se somete a mutación polinómica acotada con probabilidad `mutacion`, de modo que cada coordenada sufra una pequeña perturbación diseñada para explorar el entorno sin salirse del rango válido. Tras el cruce y la mutación, se elimina el valor de *fitness* anterior de los descendientes modificados para forzar su reevaluación.

En la siguiente fase, se recorren todos los descendientes y, si su campo `fitness.valid` es falso (lo cual ocurre en los que han sido alterados), se recalcula su *fitness* con `toolbox.evaluate(ind)`. De esta manera, solo se invoca la función de evaluación en aquellos vectores que han cambiado, mejorando la eficiencia al evitar cálculos innecesarios. Una vez que todos los descendientes válidos tienen su *fitness* actualizado, el código procede a seleccionar la siguiente generación reemplazando la población entera con los mejores individuos de `offspring` usando `pop[:] = toolbox.select(offspring, len(pop))`, donde el método de selección es un torneo de tamaño 2.

Finalmente, justo antes de volver a la siguiente iteración, se comprueba si alguno de los descendientes ya ha alcanzado un *fitness* mayor o igual a 100.0, lo que indica que la suma de cuadrados es cero y, por tanto, el vector es la solución óptima de Sphere. Si se detecta esta condición, se asigna el motivo de parada como “convergence” y se sale del bucle. Si no se alcanza la convergencia, el código vuelve a comprobar el tiempo transcurrido; en cuanto supera los 120 segundos, el bucle se interrumpe por “timeout”. Gracias a estas comprobaciones periódicas, el algoritmo no se extiende más allá del presupuesto temporal establecido, asegurando que todas las configuraciones se evalúan bajo las mismas restricciones de tiempo y manteniendo la comparabilidad entre mediciones de *fitness* y consumo energético.

### 8.2.2.2. Inspyred

Python

```
# Función evaluadora de Sphere (minimización convertida a maximización)
def evaluador_sphere(candidates, args):
    fitness_values = [sum(x**2 for x in candidate) for candidate in candidates]
    return [-fitness for fitness in fitness_values]

# Generador de individuos: vectores reales en [-5.12, 5.12]
def generator(random, args):
    size = args.get('individual_size', INDIVIDUAL_SIZE)
    return [random.uniform(LOW, UP) for _ in range(size)]
-----
# Operadores variadores: SBX y mutación polinómica
def sbx_crossover_variator(random, candidates, args):
    crossover_rate = args.setdefault('crossover_rate', 1.0)
    sbx_eta = args.setdefault('sbx_eta', 20.0)
    bounds = (LOW, UP)
    offspring = []
    for i in range(0, len(candidates) - 1, 2):
        mom, dad = candidates[i], candidates[i + 1]
        if random.random() >= crossover_rate:
            offspring.append(mom[:]); offspring.append(dad[:])
            continue
        child1, child2 = mom[:], dad[:]
        for j in range(len(mom)):
            if random.random() < 0.5 and abs(mom[j] - dad[j]) > 1e-10:
                y1, y2 = min(mom[j], dad[j]), max(mom[j], dad[j])
                u = random.random()
                beta = 1.0 + (2.0 * (y1 - bounds[0]) / (y2 - y1))
                alpha = 2.0 - beta ** (-(sbx_eta + 1.0))
                if u <= 1.0 / alpha:
                    beta_q = (u * alpha) ** (1.0 / (sbx_eta + 1.0))
                else:
                    beta_q = (1.0 / (2.0 - u * alpha)) ** (1.0 / (sbx_eta + 1.0))
                c1 = 0.5 * ((y1 + y2) - beta_q * (y2 - y1))
                c2 = 0.5 * ((y1 + y2) + beta_q * (y2 - y1))
                child1[j] = max(min(c1, bounds[1]), bounds[0])
                child2[j] = max(min(c2, bounds[1]), bounds[0])
            offspring.append(child1);
            offspring.append(child2)
        if len(candidates) % 2 != 0:
            offspring.append(candidates[-1][:])
    return offspring

def polynomial_mutation_variator(random, candidates, args):
    mut_rate = args.setdefault('mutation_rate', 0.1)
```

```

eta = args.setdefault('eta', 20.0)
low, up = LOW, UP
mutants = []
for candidate in candidates:
    mutant = candidate[:]
    for i in range(len(candidate)):
        if random.random() < mut_rate:
            x = mutant[i]
            delta_1 = (x - low) / (up - low)
            delta_2 = (up - x) / (up - low)
            rand = random.random()
            mut_pow = 1.0 / (eta + 1.0)
            if rand < 0.5:
                xy = 1.0 - delta_1
                val = 2.0 * rand + (1.0-2.0 * rand) * (xy ** (eta+1.0))
                delta_q = val ** mut_pow - 1.0
            else:
                xy = 1.0 - delta_2
                val = 2.0*(1.0-rand)+2.0 * (rand-0.5) * (xy **(eta+1.0))
                delta_q = 1.0 - val ** mut_pow
            x = x + delta_q * (up - low)
            mutant[i] = min(max(x, low), up)
    mutants.append(mutant)
return mutants
-----
# Terminadores: tiempo máximo y convergencia
def time_termination(population, num_generations, num_evaluations, args):
    start_time = args.get('start_time', None)
    max_time = args.get('max_time', MAX_TIME_SECONDS)
    if start_time is None:
        return False
    elapsed_time = time.time() - start_time
    if elapsed_time >= max_time:
        stats.termination_cause = "timeout"
        return True
    return False

def convergence_termination(population, num_generations, num_evaluations,
args):
    if not population:
        return False
    best_ind = max(population)
    raw_fitness = -best_ind.fitness
    normalized_fitness = normalizar_fitness(raw_fitness)
    if normalized_fitness >= 1.0:
        stats.termination_cause = "convergence"
        return True
    return False

```

```

def best_fitness_observer(population, num_generations, num_evaluations,
args):
    if not population:
        return
    best_ind = max(population)
    raw_fitness = -best_ind.fitness
    normalized_fitness = normalizar_fitness(raw_fitness)
    if num_generations == 0:
        stats.initial_fitness = normalized_fitness
        stats.best_fitness = normalized_fitness
        stats.best_raw_fitness = raw_fitness
    if normalized_fitness > stats.best_fitness:
        stats.best_fitness = normalized_fitness
        stats.best_raw_fitness = raw_fitness
        stats.gen_fitness_max = num_generations
-----
# Llamada a evolve
ea = ec.GA(rand)
ea.selector = inspyred.ec.selectors.tournament_selection
ea.variator = [sbx_crossover_variator, polynomial_mutation_variator]
ea.replacer = inspyred.ec.replacers.generation_replacement
ea.terminator = [
    inspyred.ec.terminators.generation_termination,
    time_termination,
    convergence_termination
]
ea.observer = best_fitness_observer

final_pop = ea.evolve(
    generator=generator,
    evaluator=evaluador_sphere,
    pop_size=pop_size,
    maximize=True,
    bounder=inspyred.ec.Bounder(LOW, UP),
    max_generations=MAX_GENERATIONS,
    num_selected=pop_size,
    individual_size=INDIVIDUAL_SIZE,
    crossover_rate=cx_rate,
    mutation_rate=MUTATION_RATE,
    sbx_eta=20.0,
    eta=20.0,
    start_time=start_time,
    max_time=MAX_TIME_SECONDS
)

```

En Inspyred, la función `evaluador_sphere` toma una lista de candidatos (cada uno es un vector real) y calcula para cada individuo la suma de los cuadrados de sus componentes.

Dado que Sphere es un problema de minimización, esta suma se devuelve con signo negativo para que Inspyred lo trate como maximización: así, el candidato que minimice la suma de cuadrados tendrá el valor más alto de *fitness*. La generación de cada individuo se delega a la función `generator`, que utiliza `random.uniform(LOW, UP)` para asignar cada coordenada dentro del rango [-5.12,5.12]. Al pasar `individual_size=INDIVIDUAL_SIZE` en la llamada a `evolve`, se indica cuántas veces debe repetirse esa llamada al generador, creando un vector de longitud 1.024.

Los operadores genéticos en Inspyred se definen como `variators` completos en lugar de operar individuo a individuo. El SBX (`sbx_crossover_variator`) recorre pares sucesivos de padres y, con probabilidad `crossover_rate`, genera dos hijos. Tras producir los valores intermedios `c1` y `c2`, cada componente se restringe para que no salga del intervalo `[LOW,UP]`. Si no se aplica el cruce en un par, simplemente se copian los padres a la siguiente generación.

La mutación polinómica (`polynomial_mutation_variator`) recorre cada vector y cada componente con probabilidad `mutation_rate = 0.1`. En cada coordenada afectada, se calcula `delta_1` y `delta_2` para medir la distancia al límite inferior y superior, respectivamente, y luego se aplica la “distribución polinómica” controlada por `eta = 20.0`. Según un número aleatorio `rand`, se determinan `delta_q` para desplazar el valor actual `x` a una nueva coordenada `x + delta_q*(up - low)`. Así, la mutación actúa sobre todo el vector y no solo sobre una posición.

Para detener la ejecución, Inspyred permite especificar múltiples terminadores. `time_termination` verifica cada generación si ha pasado más de `MAX_TIME_SECONDS` (120 s) desde `start_time` y, en ese caso, marca la causa como “`timeout`” y devuelve `True`. Por su parte, `convergence_termination` examina la población actual, selecciona el mejor individuo (`max(population)`) y convierte su *fitness* de vuelta al problema de minimización original. Si el *fitness* normalizado alcanza 1.0 (es decir, la suma de cuadrados es cero), marca “`convergence`” y retorna `True`.

Finalmente, la llamada a `ea.evolve(...)` inicia el ciclo evolutivo de Inspyred con todos los parámetros de configuración. Se proporcionan el generador, el evaluador, el tamaño de población (`pop_size`), el bounder que impide salirse de `[LOW,UP]`, y los parámetros internos como `crossover_rate`, `mutation_rate`, `sbx_eta` y `eta`. También se pasan `start_time` y `max_time` para que los terminadores tengan acceso a la hora de inicio y al tiempo máximo permitido. Inspyred se encarga internamente de crear la población inicial, aplicar selección por torneo, invocar los `variators` de SBX y mutación polinómica, volver a evaluar los descendientes y reemplazar la población mientras no se cumpla ningún terminador.

### 8.2.2.3. ECJ

```
Java

// RunSphere.java: arranque de ECJ con el archivo de parámetros
public class RunSphere {
    public static void main(String[] args) {
        String[] ecjArgs = { "-file", "sphere.params" };
        Evolve.main(ecjArgs);
    }
}

-----
```

```
// SphereProblem.java: evaluación de la función Sphere
public class SphereProblem extends Problem implements SimpleProblemForm {
    public static final double LOW    = -5.12;
    public static final double UP     =  5.12;
    public static final int   N      = 1024;
    public static final double F_MAX = N * UP * UP;

    @Override
    public void evaluate(final EvolutionState state,
                         final Individual ind,
                         final int subpopulation,
                         final int threadnum) {
        DoubleVectorIndividual iv = (DoubleVectorIndividual) ind;
        double raw = 0.0;
        for (double x : iv.genome) {
            raw += x * x;
        }
        // Convertir a fitness a maximizar en [0,100]
        double scaled = (1.0 - raw / F_MAX);
        ((SimpleFitness) iv.fitness).setFitness(state, scaled, raw == 0.0);
        iv.evaluated = true;
    }
}

-----
```

```
// TimeConstrainedEvaluator.java: terminación por límite de tiempo (120
segundos)
public class TimeConstrainedEvaluator extends SimpleEvaluator {
    public static final long MAX_TIME = 120000; // milisegundos
    private long startTime;
    public boolean timeLimitReached = false;

    @Override
    public void setup(final EvolutionState state, final Parameter base) {
        super.setup(state, base);
        startTime = System.currentTimeMillis();
    }

    @Override
```

```
public void evaluatePopulation(final EvolutionState state) {  
    if (timeLimitReached) return;  
    super.evaluatePopulation(state);  
    long elapsed = System.currentTimeMillis() - startTime;  
    if (elapsed > MAX_TIME) {  
        timeLimitReached = true;  
    }  
}
```

```
Unset

pop.subpop.0.size = 1024
pop.subpop.0.species = ec.vector.RealVectorSpecies
pop.subpop.0.species.genome-size = 1024
pop.subpop.0.species.min-gene = -5.12
pop.subpop.0.species.max-gene = 5.12

select.tournament.size = 2

pop.subpop.0.species.pipe.source.0 = ec.vector.breed.VectorCrossoverPipeline
pop.subpop.0.species.pipe.source.0.prob = 0.2

pop.subpop.0.species.pipe.source.1 = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.1.prob = 0.1

eval.problem = SphereProblem
eval = TimeConstrainedEvaluator
eval.max-generations = 100000
```

En ECJ, la ejecución de Sphere se inicia con la clase `RunSphere.java`, que simplemente invoca `Evolve.main(...)` indicando el archivo de parámetros `sphere.params`. De este modo, ECJ carga toda la configuración sin que sea necesario escribir bucles de evolución en Java.

El archivo `sphere.params` especifica que existe un único subgrupo de población (`pop.subpop.0`) cuyo tamaño varía entre  $2^6$ ,  $2^{10}$  o  $2^{14}$ . La naturaleza del individuo elegida es `ec.vector.RealVectorSpecies`, con un genoma de longitud 1.024 en el intervalo [-5.12,5.12]. Con estas líneas, ECJ sabe que cada individuo es una instancia de `DoubleVectorIndividual` cuya longitud de vector (número de genes) es 1.024 y que, al generarse o modificarse, cada componente debe mantenerse dentro de esos límites.

La selección de padres se realiza mediante un torneo de tamaño dos, indicado por `select.tournament.size = 2`. Esto significa que, para cada pareja de descendientes, ECJ elige dos individuos de la población actual al azar para generar la descendencia. Los operadores de variación se definen como parte del pipe de la especie: en primer lugar, `VectorCrossoverPipeline` con probabilidad 0.2 aplica un cruce SBX implícito para vectores reales en [-5.12,5.12]. Inmediatamente después, `VectorMutationPipeline` con probabilidad 0.1 introduce una mutación polinómica sobre cada coordenada, asegurando que las nuevas soluciones sigan dentro de los límites establecidos.

El archivo de parámetros también enlaza el problema al código Java mediante `eval.problem = SphereProblem`. La clase `SphereProblem.java` define el método `evaluate(...)` que se invoca para cada individuo antes o después de aplicar los operadores genéticos. En él, ECJ obtiene el `genome` del individuo (un array de dobles) y recorre cada componente para calcular la suma de los cuadrados. Esa suma “`raw`” se convierte en un `fitness` escalado entre 0 y 1 mediante `1 - raw/F_MAX` donde  $F_{MAX} = 1.024 \times 5,12^2$  es el peor caso posible. Al llamar a `((SimpleFitness) iv.fitness).setFitness(state, scaled, raw == 0.0)`, ECJ considera como óptimos aquellos vectores cuya suma de cuadrados sea cero (todos los componentes a 0) y asigna el valor máximo de `fitness` (1) para ellos. De esta forma, ECJ tratará de maximizar ese `fitness`, aunque el problema original de Sphere es de minimización.

Para controlar la duración del experimento, el parámetro `eval = TimeConstrainedEvaluator` indica que ECJ debe usar la clase `TimeConstrainedEvaluator.java` en lugar de su evaluador por defecto. En esa clase, `setup(...)` guarda el instante de inicio (`startTime`) y, en cada llamada a `evaluatePopulation(state)`, comprueba si han pasado más de 120.000 milisegundos (dos minutos). Si el tiempo excede este límite, establece `timeLimitReached = true` y evita nuevas evaluaciones en generaciones sucesivas. De este modo, ECJ detiene automáticamente la evolución al cumplirse el criterio de tiempo.

Además de esta terminación por tiempo, el parámetro `eval.max-generations = 100000` establece un tope nominal de generaciones, que ECJ solo alcanza si no se cumple antes el límite de tiempo o la convergencia. El control de convergencia y la recolección de métricas se delegan en la clase `SphereStatistics.java`, que se activa tras cada evaluación de población. En su método `postEvaluationStatistics`, ECJ obtiene el `fitness` máximo de la generación actual (`getMaxFitness(state)`) y lo compara con el `fitness` registrado hasta el momento. Si detecta que el mejor `fitness` alcanza 1.0 (equivalente a suma de cuadrados cero), marca `stopReason = "Convergence"` y llama `state.finish(...)` para interrumpir la ejecución. Si en cambio el evaluador ha señalado `timeLimitReached = true`, registra `stopReason = "Timeout"` y también finaliza la evolución.

#### 8.2.2.4. ParadisEO

```
C/C++
// Definición del individuo (solo la parte esencial)
struct Sphere : public EO<eoMaximizingFitness> {
    vector<double> x;
    Sphere(size_t n) : x(n, 0.0) {}
    void printOn(ostream &os) const override {
        for (double v : x) os << v << " ";
    }
    void readFrom(istream &is) override {
        for (auto &v : x) is >> v;
    }
};

// Evaluador mínimo: suma de cuadrados escalada a [0,100]
struct SphereFunction : public eoEvalFunc<Sphere> {
    static constexpr double LOW = -5.12, UP = 5.12;
    static constexpr size_t N = 1024;
    static constexpr double FMAX = N * UP * UP;
    void operator()(Sphere &ind) override {
        double raw = 0;
        for (double v : ind.x) raw += v * v;
        ind.fitness((1.0 - raw / FMAX));
    }
};

// SBX de un punto (simplificado al núcleo del cálculo)
struct SBXCrossover : public eoQuadOp<Sphere> {
    double eta;
    SBXCrossover(double _eta) : eta(_eta) {}
    bool operator()(Sphere &a, Sphere &b) override {
        for (size_t i = 0; i < a.x.size(); ++i) {
            double u = rng.uniform();
            double beta = (u <= 0.5)
                ? pow(2.0 * u, 1.0 / (eta + 1))
                : pow(1.0 / (2.0 * (1.0 - u)), 1.0 / (eta + 1));
            double c1 = 0.5 * ((1 + beta) * a.x[i] + (1 - beta) * b.x[i]);
            double c2 = 0.5 * ((1 - beta) * a.x[i] + (1 + beta) * b.x[i]);
            a.x[i] = clamp(c1, LOW, UP);
            b.x[i] = clamp(c2, LOW, UP);
        }
        return true;
    }
};

// Mutación polinómica esencial (único paso por gen si rnd < pm)
struct PolyMutation : public eoMonOp<Sphere> {
```

```

    double pm, eta;
PolyMutation(double _pm, double _eta) : pm(_pm), eta(_eta) {}
bool operator()(Sphere &ind) override {
    bool mutated = false;
    for (double &v : ind.x) {
        if (rng.uniform() < pm) {
            double u = rng.uniform();
            double delta = (u < 0.5)
                ? pow(2.0 * u, 1.0/(eta+1)) - 1
                : 1 - pow(2.0 * (1-u), 1.0/(eta+1));
            v = clamp(v + delta * (UP - LOW), LOW, UP);
            mutated = true;
        }
    }
    return mutated;
}
};

// Bucle principal (resumido)
int main(int argc, char **argv) {
    size_t popSize = 1024; // ajustable con -p
    double pc = 0.8, pm = 0.1; // ajustables con -c y -m
    SphereInit init; SphereFunction eval; SBXCrossover xover(20.0);
    PolyMutation mutate(pm, 20.0);
    eoDetTournamentSelect<Sphere> select(2);
    eoPop<Sphere> pop;
    for (size_t i = 0; i < popSize; ++i) {
        Sphere ind(SphereFunction::N);
        init(ind); eval(ind);
        pop.push_back(ind);
    }
    double bestFit = pop[0].fitness();

    auto t0 = chrono::steady_clock::now();
    const int maxTime = 120;
    size_t gen = 0;
    string stop = "timeout";

    while (true) {
        if (chrono::duration_cast<chrono::seconds>(
            chrono::steady_clock::now() - t0).count() >= maxTime) {
            break;
        }
        ++gen;
        eoPop<Sphere> offspring;
        while (offspring.size() < popSize) {
            Sphere a = select(pop), b = select(pop);
            if (rng.uniform() < pc) xover(a, b);
        }
    }
}

```

```

        mutate(a); mutate(b);
        eval(a); eval(b);
        offspring.push_back(a);
        if (offspring.size() < popSize) offspring.push_back(b);
    }
    pop = offspring;
    for (auto &ind : pop) {
        if (ind.fitness() > bestFit) {
            bestFit = ind.fitness();
            if (bestFit >= 100.0) { stop = "convergence"; }
        }
    }
    if (bestFit >= 100.0) break;
}
return 0;
}

```

En ParadisEO, cada individuo se representa con la estructura `Sphere`, que contiene un vector de 1.024 valores reales. Desde el momento en que se crea un `Sphere`, su vector está listo para recibir valores aleatorios, y las funciones `printOn` y `readFrom` permiten volcar el contenido o reconstruirlo de manera aleatoria.

La evaluación de cada individuo la realiza la clase `SphereFunction`. En su método `operator()`, recorre las 1.024 coordenadas de `ind.x`, suma  $v * v$  para cada componente y obtiene un valor “`raw`”. A continuación, convierte este valor en un *fitness* en el rango [0, 1] mediante  $1 - raw/F\_MAX$  donde  $F\_MAX = 1.024 \times 5,12^2$  es el peor caso posible. Con ello, un individuo que minimiza la suma de cuadrados (todas las coordenadas a cero) alcanza un *fitness* de 1, mientras que otros se sitúan en rangos inferiores.

Para generar la población inicial, `SphereInit` asigna a cada componente de `x` un valor uniformemente distribuido en [-5.12, 5.12]. En `main`, se crea un `eoPop<Sphere>` `pop` de tamaño `popSize` y, para cada elemento, se invoca `init(ind)` seguido de `eval(ind)` para obtener su *fitness* inicial. De esa evaluación se extrae el valor máximo (`initMax`), que sirve como referencia para calcular la variación de *fitness* en las generaciones sucesivas.

El cruce se implementa en `SBXCrossover`, que hereda de `eoQuadOp<Sphere>`. Para cada par de padres, calcula un factor `beta` en cada coordenada mediante un número aleatorio y el parámetro `eta = 20`. A partir de `beta`, genera dos nuevos valores `c1` y `c2` para cada índice, los cuales restringe al rango [-5.12, 5.12]. La mutación polinómica, definida en `PolyMutation`, recorre cada coordenada y, con probabilidad `pm`, aplica una mutación basada en una distribución polinómica controlada por `eta`, desplazando `v` en un tramo proporcional a  $(5.12 - (-5.12))$ .

El bucle evolutivo en `main` comienza midiendo el instante inicial y en cada generación primero comprueba si han transcurrido 120 segundos, en cuyo caso se detiene por

“timeout”. Si todavía queda tiempo, construye `eoPop<Sphere> offspring`, donde cada nuevo individuo nace seleccionando dos padres por torneo de tamaño 2 y aplicando SBX con probabilidad `pc`, seguido de mutación con probabilidad `pm`. Ambos hijos se evalúan inmediatamente con `eval` y se añaden a la nueva población hasta llenar `popSize`. Tras reemplazar `pop`, se recorre la población para actualizar `bestFit`; si llega a 1 se detiene por “convergence”.

### 8.2.3. Rosenbrock

En este apartado se muestra cómo se configura el *benchmark* Rosenbrock en cada *framework*. El código esencial incluye la definición de la función de evaluación, la construcción de individuos de tipo real, la selección de operadores genéticos adecuados (SBX para cruce y mutación polinómica acotada) y el control de tiempo de ejecución mediante un bucle que interrumpe tras 120 segundos.

#### 8.2.3.1. DEAP

Python

```
# Función de fitness: calcula Rosenbrock y normaliza
def rosenbrock_function(individual):
    raw = 0
    for i in range(len(individual) - 1):
        raw += 100 * (individual[i+1] - individual[i]**2)**2 +
(individual[i] - 1)**2
    raw = min(raw, WORST_CASE_VALUE)
    if raw > stats.worst_raw_value:
        stats.worst_raw_value = raw
    if raw < stats.best_raw_value:
        stats.best_raw_value = raw
    norm = 1.0 - (raw / (stats.worst_raw_value + 1e-10))
    return (max(0.0, min(1.0, norm)),)

# Crossover SBX seguro contra desbordamientos
def safe_sbx_crossover(ind1, ind2, eta, prob):
    if random.random() > prob:
        return ind1, ind2
    for i in range(len(ind1)):
        if abs(ind1[i] - ind2[i]) > 1e-10:
            y1, y2 = sorted((ind1[i], ind2[i]))
            try:
                beta = 1.0 + (2.0 * (y1 - LOWER_BOUND) / (y2 - y1))
                alpha = 2.0 - min(100.0, beta**(eta + 1.0))
                u = random.random()
                if u <= 1.0 / alpha:
                    beta_q = (u * alpha) ** (1.0 / (eta + 1.0))
                else:
                    beta_q = (1.0 / (2.0 - u * alpha)) ** (1.0 / (eta +
1.0))
            except:
                pass
            ind1[i] = beta_q * (ind2[i] - y1) + y1
            ind2[i] = beta_q * (ind1[i] - y1) + y1
    return ind1, ind2
```

```

        c1 = 0.5 * ((y1 + y2) - beta_q * (y2 - y1))
        c2 = 0.5 * ((y1 + y2) + beta_q * (y2 - y1))
        c1 = max(LOWER_BOUND, min(UPPER_BOUND, c1))
        c2 = max(LOWER_BOUND, min(UPPER_BOUND, c2))
        ind1[i], ind2[i] = (c1, c2) if ind1[i] < ind2[i] else (c2,
c1)
    except:
        blend = random.random()
        tmp1 = blend * ind1[i] + (1 - blend) * ind2[i]
        tmp2 = (1 - blend) * ind1[i] + blend * ind2[i]
        ind1[i], ind2[i] = tmp1, tmp2
    return ind1, ind2

# Mutación gaussiana por gen, limitada a los bounds
def custom_mutation(individual, indpb, bitpb, sigma):
    if random.random() < indpb:
        for i in range(len(individual)):
            if random.random() < bitpb:
                individual[i] += random.gauss(0, sigma)
                individual[i] = max(min(individual[i], UPPER_BOUND),
LOWER_BOUND)
    return (individual,)

# Registro de creators y toolbox
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)
toolbox = base.Toolbox()
def init_individual():
    mean = (LOWER_BOUND + UPPER_BOUND) / 2.0
    spread = (UPPER_BOUND - LOWER_BOUND) / 4.0
    return [random.uniform(mean - spread, mean + spread) for _ in
range(INDIVIDUAL_SIZE)]
toolbox.register("individual", tools.initIterate, creator.Individual,
init_individual)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("evaluate", rosenbrock_function)
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", safe_sbx_crossover, eta=2.0, prob=cx_rate)
toolbox.register("mutate", custom_mutation,
indpb=MUTATION_RATE_INDIVIDUAL,
bitpb=MUTATION_RATE_BIT,
sigma=(UPPER_BOUND - LOWER_BOUND) * 0.1)

# Evolución: selección, cruce, mutación y terminación por tiempo
population = toolbox.population(n=pop_size)
hof = tools.HallOfFame(1)
# Evaluar población inicial
for ind, fit in zip(population, map(toolbox.evaluate, population)):
```

```

ind.fitness.values = fit

gen = 0
stop_evolution = False
while gen < MAX_GENERATIONS and not stop_evolution:
    offspring = toolbox.select(population, len(population))
    offspring = list(map(toolbox.clone, offspring))
    for i in range(1, len(offspring), 2):
        toolbox.mate(offspring[i-1], offspring[i])
        del offspring[i-1].fitness.values, offspring[i].fitness.values
    for ind in offspring:
        toolbox.mutate(ind)
        if not ind.fitness.valid:
            del ind.fitness.values
    for ind, fit in zip([ind for ind in offspring if not ind.fitness.valid],
                       map(toolbox.evaluate, [ind for ind in offspring if
not ind.fitness.valid])):
        ind.fitness.values = fit
    population[:] = offspring
    hof.update(population)
    stop_evolution = stats_callback(gen, population, hof)
    gen += 1

```

En el desarrollo de DEAP para Rosenbrock, se define primero la función `rosenbrock_function` que, por cada individuo (lista de 1.024 reales), calcula el valor bruto de Rosenbrock y luego lo normaliza dinámicamente entre 0 y 1 usando los valores extremos. De esta forma, el algoritmo siempre maximiza un *fitness* creciente en cuanto el vector se acerca al óptimo del problema.

Para el cruce, `safe_sbx_crossover` implementa Simulated Binary Crossover (SBX) sobre cada par de valores progenitores. Se ordenan las dos coordenadas, se calcula el factor `beta` en función de un número aleatorio `u` y el parámetro `eta`, y se obtienen dos hijos `c1` y `c2` que luego se ajustan al intervalo `[LOWER_BOUND, UPPER_BOUND]`. Si en algún momento surge un error numérico (por ejemplo, división por cero o desbordamiento), se recurre a un cruce aritmético simple para asegurar la estabilidad del problema.

La mutación gaussiana queda en `custom_mutation`: primero decide, con probabilidad `indpb`, si un individuo debe mutar. Si es así, recorre cada una de las 1.024 coordenadas `y`, con probabilidad `bitpb`, añade un valor tomado de una distribución normal  $\text{gauss}(0, \sigma)$ , volviendo a ser ajustado dentro del rango correcto. `Sigma` se fija como el 10 % del tamaño total del intervalo `[LOWER_BOUND, UPPER_BOUND]`, provocando modificaciones moderadas en cada gen.

En el flujo principal, se registra el tipo `FitnessMax` y el tipo `Individual` como lista de valores reales, usando `init_individual` para inicializar cada individuo alrededor del

centro del rango permitido. Después de crear la población, se evalúa a cada individuo con `rosenbrock_function`. En cada generación, se seleccionan progenitores por torneo de tamaño 2, se clonian, se aplica SBX por pares y luego se mutan todos los descendientes. Solo aquellos cuyo atributo `.fitness` quedó invalidado se vuelven a evaluar, optimizando el número de llamadas a la función de Rosenbrock. Tras actualizarse, se llama a `stats_callback`, que registra el mejor *fitness*, detecta convergencia o verifica si han pasado 120 segundos, y en ese caso detiene la evolución.

### 8.2.3.2. Inspyred

Python

```
# Registro de tipos y toolbox
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)
toolbox = base.Toolbox()
toolbox.register("individual", tools.initRepeat, creator.Individual,
                 lambda: random.uniform(LOWER_BOUND, UPPER_BOUND),
                 INDIVIDUAL_SIZE)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Evaluador de Rosenbrock normalizado
def rosen_fitness(candidates, args):
    results = []
    for x in candidates:
        raw = sum(100*(x[i+1]-x[i]**2)**2 + (x[i]-1)**2 for i in
range(len(x)-1))
        raw = min(raw, WORST_CASE_VALUE)
        stats.worst_raw_value = max(stats.worst_raw_value, raw)
        stats.best_raw_value = min(stats.best_raw_value, raw)
        norm = 1 - raw/(stats.worst_raw_value + 1e-10)
        results.append((max(0, min(1, norm)),))
    return results
toolbox.register("evaluate", rosen_fitness)

# Operadores variadores para toda la población
def sbx_variator(random, candidates, args):
    eta, prob = args["eta"], args["crossover_rate"]
    low, up = LOWER_BOUND, UPPER_BOUND
    offspring = []
    for p1, p2 in zip(candidates[::2], candidates[1::2]):
        if random.random() > prob:
            offspring.extend([p1[:], p2[:]])
            continue
        c1, c2 = p1[:], p2[:]
        for i in range(len(p1)):
            if abs(p1[i]-p2[i]) > 1e-10:
                y1, y2 = sorted((p1[i], p2[i]))
                u = random.random()
                if u < prob:
                    if u < 0.5:
                        c1[i] = y1 + (y2 - y1) * u
                    else:
                        c2[i] = y1 + (y2 - y1) * u
                else:
                    if u < 0.5:
                        c1[i] = y1 + (y2 - y1) * (1 - u)
                    else:
                        c2[i] = y1 + (y2 - y1) * (1 - u)
        offspring.append(c1)
        offspring.append(c2)
```

```

        beta = (2*u)**(1/(eta+1)) if u <= 0.5 else
(1/(2*(1-u)))**(1/(eta+1))
        c1[i] = 0.5*((1+beta)*y1 + (1-beta)*y2)
        c2[i] = 0.5*((1-beta)*y1 + (1+beta)*y2)
        c1[i] = max(low, min(up, c1[i]))
        c2[i] = max(low, min(up, c2[i]))
    offspring.extend([c1, c2])
if len(candidates)%2: offspring.append(candidates[-1][:])
return offspring

def gauss_variator(random, candidates, args):
    mutp, sigma = args["mutation_rate"], args["sigma"]
    mutants = []
    for cand in candidates:
        m = cand[:]
        if random.random() < mutp:
            for i in range(len(m)):
                if random.random() < mutp:
                    m[i] += random.gauss(0, sigma)
                    m[i] = max(LOWER_BOUND, min(UPPER_BOUND, m[i]))
        mutants.append(m)
    return mutants

# Terminadores y observador
def time_term(pop, gen, evals, args):
    if time.time() - args["start_time"] >= MAX_TIME_SECONDS:
        stats.termination_cause = "timeout"
        return True
    return False

def conv_term(pop, gen, evals, args):
    if not pop: return False
    if max(ind.fitness for ind in pop) >= 1.0:
        stats.termination_cause = "convergence"
        return True
    return False

def best_obs(pop, gen, evals, args):
    best_norm = max(ind.fitness for ind in pop)
    if gen == 0:
        stats.initial_fitness = best_norm
    if best_norm > stats.best_fitness:
        stats.best_fitness = best_norm
        stats.gen_best_fitness = gen
    stats.current_generation = gen

# Configuración de Inspyred y llamada a evolve
ea = ec.GA(random.Random())

```

```

ea.selector      = inspyred.ec.selectors.tournament_selection
ea.variator     = [sbx_variator, gauss_variator]
ea.replacer     = inspyred.ec.replacers.generational_replacement
ea.terminator   = [inspyred.ec.terminators.generation_termination,
                  time_term, conv_term]
ea.observer     = best_obs

final_pop = ea.evolve(
    generator      = generator,
    evaluator      = rosen_fitness,
    pop_size       = pop_size,
    maximize       = True,
    bounder        = inspyred.ec.Bounder(LOWER_BOUND, UPPER_BOUND),
    max_generations = MAX_GENERATIONS,
    num_selected   = pop_size,
    individual_size = INDIVIDUAL_SIZE,
    crossover_rate = cx_rate,
    mutation_rate  = MUTATION_RATE_INDIVIDUAL,
    eta            = 2.0,
    sigma           = (UPPER_BOUND-LOWER_BOUND)*0.1,
    start_time     = start_time,
    max_time       = MAX_TIME_SECONDS
)

```

En este fragmento se definen los pasos básicos para configurar Rosenbrock en Inspyred. Primero, se registra un tipo de *fitness* `FitnessMax` y un tipo de individuo como lista de valores reales de longitud 1.024. La función `rosen_fitness` calcula el valor bruto de Rosenbrock para cada candidato, actualiza en `stats` el peor y el mejor valor observado y, a partir de ello, normaliza dinámicamente el *fitness* en [0,1]. A continuación, los variadores `sbx_variator` y `gauss_variator` operan sobre toda la población. El primero implementa SBX component-wise, evitando divisiones por cero y ajustando los valores resultantes al intervalo [-5,10]; el segundo realiza mutaciones gaussianas en cada gen con probabilidad `mutation_rate`, asegurándose que no se obtengan valores fuera del dominio.

Para controlar cuándo parar, se definen dos terminadores: `time_term`, que detiene la evolución si han pasado 120 segundos, y `conv_term`, que la detiene en cuanto algún individuo alcanza un *fitness* normalizado de 1.0 (es decir, valor de Rosenbrock igual a cero). El observador `best_obs` se encarga de almacenar el *fitness* inicial, actualizar el mejor *fitness* por generación y anotar la generación en que ocurre.

Finalmente, la llamada `ea.evolve(...)` reúne todo: el generador de vectores en [-5,10], el evaluador normalizado, el selector por torneo, los variadores SBX y gaussiano, el reemplazador generacional y los terminadores. Con esto, Inspyred gestiona automáticamente el bucle evolutivo, aplicando cruce y mutación por generación, evaluando sólo los individuos cuyo *fitness* quedó inválido y deteniéndose al cumplirse el criterio de tiempo o convergencia.

### 8.2.3.3. ECJ

```
Java
// RunRosenbrock.java
public class RunRosenbrock {
    public static void main(String[] args) {
        Evolve.main(new String[] { "-file", "rosenbrock.params" });
    }
}

-----
// RosenbrockProblem.java
public class RosenbrockProblem extends Problem implements SimpleProblemForm {
    public static final double LOW = -5.12, UP = 5.12;
    public static final int N = 1024;
    public static final double F_MAX = N * 40000; // cota superior

    @Override
    public void evaluate(final EvolutionState state, final Individual ind,
                        final int subpop, final int threadnum) {
        DoubleVectorIndividual iv = (DoubleVectorIndividual)ind;
        double raw = 0.0;
        for (int i = 0; i < iv.genome.length - 1; i++) {
            double d = iv.genome[i];
            double e = iv.genome[i + 1];
            raw += 100 * Math.pow(e - d * d, 2) + Math.pow(d - 1, 2);
        }
        if (Double.isNaN(raw) || raw > F_MAX) raw = F_MAX;
        double scaled = (1.0 - raw / F_MAX) * 100.0;
        scaled = Math.max(0.0, Math.min(100.0, scaled));
        ((SimpleFitness)iv.fitness).setFitness(state, scaled, raw == 0.0);
        iv.evaluated = true;
    }
}

-----
// TimeConstrainedEvaluator.java
public class TimeConstrainedEvaluator extends SimpleEvaluator {
    private long startTime;

    @Override
    public void setup(final EvolutionState state, final Parameter base) {
        super.setup(state, base);
        startTime = System.currentTimeMillis();
    }

    @Override
    public void evaluatePopulation(final EvolutionState state) {
        super.evaluatePopulation(state);
        if (System.currentTimeMillis() - startTime > 120000) {
```

```
        state.output.message("Time limit reached");
        state.finish(EvolutionState.R_SUCCESS);
    }
}
```

```
Unset

# rosenbrock.params
pop.subpop.0.size = 1024
pop.subpop.0.species = ec.vector.RealVectorSpecies
pop.subpop.0.species.genome-size = 1024
pop.subpop.0.species.min-gene = -5.12
pop.subpop.0.species.max-gene = 5.12

select.tournament.size = 2

pop.subpop.0.species.pipe.source.0 = ec.vector.breed.VectorCrossoverPipeline
pop.subpop.0.species.pipe.source.0.prob = 0.2

pop.subpop.0.species.pipe.source.1 = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.1.prob = 0.1

eval.problem = RosenbrockProblem
eval = TimeConstrainedEvaluator
eval.max-generations = 10000000
```

En la codificación de Rosenbrock en ECJ, se puede observar que `RunRosenbrock.java` invoca a ECJ usando el archivo `rosenbrock.params`, de modo que la gestión de la población y operadores queda controlada por dicho archivo. En `rosenbrock.params`, se define una única subpoblación de  $n$  vectores (se ajusta a  $2^6, 2^{10}, 2^{14}$ ) de longitud 1.024, donde cada gen toma valores dentro del intervalo  $[-5.12, 5.12]$ .

La selección se lleva a cabo mediante torneos de tamaño 2 y el conjunto de operadores aplica primero un cruce de tipo SBX (propio en ECJ en `VectorCrossoverPipeline`) con probabilidad ajustada a 0.01, 0.2 o 0.8, seguido de una mutación (`VectorMutationPipeline`) con probabilidad 0.1.

Cuando ECJ evalúa un individuo, llama a `RosenbrockProblem.evaluate(...)`, donde se calcula la suma de  $100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2$  para  $i = 0 \dots (N - 2)$ . El valor `raw` se limita a una cota `F_MAX` para evitar desbordamientos, y luego se convierte en un *fitness* de  $[0,1]$  con  $1 - \text{raw}/\text{F\_MAX}$ . Un vector cuyos componentes sean 1 en su totalidad tendrá un `raw = 0` y, por tanto, recibirá un *fitness* de 1.

Para asegurarse de que ninguna ejecución supere los dos minutos, `TimeConstrainedEvaluator.java` hereda de `SimpleEvaluator` y, tras cada llamada a `super.evaluatePopulation(state)`, mide el tiempo transcurrido desde `startTime`. Si han transcurrido más de 120.000 ms, imprime un mensaje y llama a `state.finish(...)` para detener ECJ inmediatamente. Con este evaluador se evita implementar bucles manuales de control de tiempo: ECJ interrumpe la evolución tan pronto se cumpla el límite de dos minutos.

#### 8.2.3.4. ParadisEO

```
C/C++
// Definición mínima del individuo y su evaluación
struct Rosenbrock : public EO<eoMaximizingFitness> {
    vector<double> x;
    double raw_value;
    Rosenbrock(size_t n) : x(n), raw_value(DBL_MAX) {}
    void printOn(ostream &os) const override { /* ... */ }
    void readFrom(istream &is) override { /* ... */ }
};

struct RosenbrockFunction : public eoEvalFunc<Rosenbrock> {
    void operator()(Rosenbrock &ind) override {
        double raw = 0.0;
        for (size_t i = 0; i + 1 < ind.x.size(); ++i)
            raw += 100.0 * pow(ind.x[i+1] - ind.x[i]*ind.x[i], 2)
                + pow(ind.x[i] - 1.0, 2);
        if (raw > WORST_CASE_VALUE) raw = WORST_CASE_VALUE;
        ind.raw_value = raw;
        stats.worst_raw_value = max(stats.worst_raw_value, raw);
        stats.best_raw_value = min(stats.best_raw_value, raw);
        double scaled = (1.0 - raw / F_MAX) * 100.0;
        scaled = clamp(scaled, 0.0, 100.0);
        ind.fitness(scaled);
    }
};

// Inicializador y operadores genéticos esenciales
struct RosenbrockInit : public eoInit<Rosenbrock> {
    void operator()(Rosenbrock &ind) override {
        ind.x.resize(INDIVIDUAL_SIZE);
        for (double &v : ind.x)
            v = rng.uniform(LOWER_BOUND, UPPER_BOUND);
    }
};

struct SafeSBXCrossover : public eoQuadOp<Rosenbrock> {
    double eta;
    SafeSBXCrossover(double _eta) : eta(_eta) {}
    bool operator()(Rosenbrock &a, Rosenbrock &b) override {
        for (size_t i = 0; i < a.x.size(); ++i) {
```

```

        if (abs(a.x[i] - b.x[i]) < 1e-10) continue;
        double y1 = min(a.x[i], b.x[i]);
        double y2 = max(a.x[i], b.x[i]);
        double beta=min(1.0 + 2.0*(y1-LOWER_BOUND)/(y2-y1), 100.0);
        double alpha = 2.0 - pow(beta, eta + 1.0);
        double u = rng.uniform();
        double beta_q = (u <= 1.0/alpha)
                      ? pow(u*alpha, 1.0/(eta+1.0))
                      : pow(1.0/(2.0 - u*alpha), 1.0/(eta+1.0));
        double c1 = 0.5*((1+beta_q)*y1 + (1-beta_q)*y2);
        double c2 = 0.5*((1-beta_q)*y1 + (1+beta_q)*y2);
        a.x[i]=clamp(a.x[i] < b.x[i] ? c1 : c2, LOWER_BOUND,
UPPER_BOUND);
        b.x[i]=clamp(a.x[i] < b.x[i] ? c2 : c1, LOWER_BOUND,
UPPER_BOUND);
    }
    return true;
}
};

struct RealMutation : public eoMonOp<Rosenbrock> {
    double p_ind, p_bit, sigma;
    RealMutation(double _p_ind, double _p_bit)
        : p_ind(_p_ind), p_bit(_p_bit) {
        sigma = (UPPER_BOUND - LOWER_BOUND) * 0.1;
    }
    bool operator()(Rosenbrock &ind) override {
        if (rng.uniform() < p_ind) {
            for (double &v : ind.x) {
                if (rng.uniform() < p_bit) {
                    v += rng.normal() * sigma;
                    v = clamp(v, LOWER_BOUND, UPPER_BOUND);
                }
            }
            return true;
        }
        return false;
    }
};

// Esqueleto del bucle evolutivo
int main(int argc, char **argv) {
    size_t popSize = 1024;
    double pc = 0.8, p_ind = 0.1, p_bit = 0.1;

    RosenbrockInit init;
    RosenbrockFunction eval;
    SafeSBXCrossover xover(2.0);
    RealMutation mutate(p_ind, p_bit);

```

```

eoDetTournamentSelect<Rosenbrock> select(2);
// Construir población inicial
eoPop<Rosenbrock> pop;
pop.reserve(popSize);
for (size_t i = 0; i < popSize; ++i) {
    Rosenbrock ind(INDIVIDUAL_SIZE);
    init(ind); eval(ind);
    pop.push_back(ind);
}
stats.initial_fitness = pop[0].fitness();
stats.best_fitness = stats.initial_fitness;
auto t0 = chrono::steady_clock::now();
size_t gen = 0;

while (true) {
    double elapsed = chrono::duration<double>(
        chrono::steady_clock::now() - t0).count();
    if (elapsed >= MAX_TIME_SECONDS || gen >= MAX_GENERATIONS)
        break;
    ++gen;
    eoPop<Rosenbrock> offspring;
    offspring.reserve(popSize);
    // Selección, cruce y mutación
    while (offspring.size() < popSize) {
        Rosenbrock p1 = select(pop), p2 = select(pop);
        if (rng.uniform() < pc) xover(p1, p2);
        mutate(p1); mutate(p2);
        eval(p1); eval(p2);
        offspring.push_back(p1);
        if (offspring.size() < popSize) offspring.push_back(p2);
    }
    pop = offspring;
    // Actualizar mejor fitness
    for (auto &ind : pop) {
        double f = ind.fitness();
        if (f > stats.best_fitness) {
            stats.best_fitness = f;
            stats.gen_best_fitness = gen;
        }
    }
}
double timeSec = chrono::duration<double>(
    chrono::steady_clock::now() - t0).count();
double variation = stats.best_fitness - stats.initial_fitness;
//...
return 0;
}

```

En ParadisEO, cada individuo Rosenbrock almacena un vector de 1.024 reales  $\mathbf{x}$  y un `raw_value` para guardar el coste de Rosenbrock sin normalizar. La clase `RosenbrockFunction` recorre esos 1.024 valores, calcula la suma de  $100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2$  y la limita a `WORST_CASE_VALUE` en caso de desbordamiento. Luego, convierte ese valor bruto en un *fitness* en  $[0,1]$  con  $1 - \text{raw}/F_{MAX}$  donde  $F_{MAX} = 1.024 \times 40.000$ . A su vez, actualiza en `stats` los valores mejor y peor vistos para poder normalizar dinámicamente.

Para poblar la población, `RosenbrockInit` asigna cada coordenada de  $\mathbf{x}$  un valor uniforme dentro del dominio  $[-5.12, 5.12]$ . En el `main`, se construye un `eoPop<Rosenbrock>` `pop` de tamaño `popSize`, y cada individuo se inicializa y evalúa inmediatamente para obtener su *fitness* inicial.

El operador SBX seguro (`SafeSBXCrossover`) recorre cada índice  $i$  de los vectores padres `a.x` y `b.x`. Si la diferencia absoluta es menor que  $1 \times 10^{-10}$ , omite ese gen. En caso contrario, ordena los dos valores  $(y_1, y_2) = \min, \max$  y calcula el factor beta con protección mediante `min(..., 100.0)` para evitar exponentes muy grandes. A partir de `beta` y un número aleatorio  $u$ , obtiene `beta_q` y genera dos hijos `c1` y `c2`, que luego limita a  $[-5.12, 5.12]$  y reasigna en el orden original. Si se lanza cualquier excepción (por ejemplo, en potencias grandes), cae en un cruce aritmético simple para garantizar estabilidad.

El bucle evolutivo en `main` controla las generaciones mediante un reloj: se mide `t0` al inicio y en cada iteración se comprueba si han transcurrido más de `MAX_TIME_SECONDS` o si se alcanzó `MAX_GENERATIONS`. Para crear la siguiente generación, se repite: seleccionar dos padres por torneo de tamaño 2, aplicar SBX con probabilidad  $p_c$ , evaluar ambos descendientes y añadirlos a `offspring` hasta llenar la población. Al terminar la evaluación de la nueva población, se escanea para actualizar el mejor *fitness* observado (`stats.best_fitness`) y su generación (`gen_best_fitness`).

#### 8.2.4. Schwefel

En este apartado se muestra cómo se configura el *benchmark* Schwefel en cada *framework*. El código esencial incluye la definición de la función de evaluación, la construcción de individuos de tipo real, la selección de operadores genéticos adecuados (SBX para cruce y mutación polinómica acotada) y el control de tiempo de ejecución mediante un bucle que interrumpe tras 120 segundos.

##### 8.2.4.1. DEAP

Python

```
# Función de fitness: evalúa Schwefel y normaliza
def schwefel_function(individual):
    raw = sum(-x * np.sin(np.sqrt(abs(x))) for x in individual)
    raw = 418.9829 * len(individual) - raw
```

```

raw = min(raw, WORST_CASE_VALUE)
stats.worst_raw_value = max(stats.worst_raw_value, raw)
stats.best_raw_value = min(stats.best_raw_value, raw)
theoretical_worst = 418.9829 * len(individual) + len(individual) *
UPPER_BOUND
norm_factor = max(stats.worst_raw_value, theoretical_worst)
norm = 1.0 - (raw / (norm_factor + 1e-10))
return (max(0.0, min(1.0, norm)),)

# SBX
def safe_sbx_crossover(ind1, ind2, eta, prob):
    if random.random() > prob:
        return ind1, ind2
    for i in range(len(ind1)):
        if abs(ind1[i] - ind2[i]) > 1e-10:
            y1, y2 = sorted((ind1[i], ind2[i]))
            beta = min(1.0 + 2.0*(y1 - LOWER_BOUND)/(y2 - y1), 100.0)
            alpha = 2.0 - min(1.0, beta**(eta + 1.0))
            u = random.random()
            if u <= 1.0/alpha:
                beta_q = (u * alpha) ** (1.0/(eta + 1.0))
            else:
                beta_q = (1.0/(2.0 - u * alpha)) ** (1.0/(eta + 1.0))
            c1 = 0.5*((1 + beta_q)*y1 + (1 - beta_q)*y2)
            c2 = 0.5*((1 - beta_q)*y1 + (1 + beta_q)*y2)
            c1 = max(LOWER_BOUND, min(UPPER_BOUND, c1))
            c2 = max(LOWER_BOUND, min(UPPER_BOUND, c2))
            if ind1[i] < ind2[i]:
                ind1[i], ind2[i] = c1, c2
            else:
                ind1[i], ind2[i] = c2, c1
    return ind1, ind2

# Mutación gaussiana
def custom_mutation(individual, indpb, bitpb, sigma):
    if random.random() < indpb:
        for i in range(len(individual)):
            if random.random() < bitpb:
                individual[i] += random.gauss(0, sigma)
                individual[i] = max(min(individual[i], UPPER_BOUND),
LOWER_BOUND)
    return individual,

# Callback para terminar al exceder 120 s o llegar a MAX_GENERATIONS
def stats_callback(gen, population, halloffame, stats_obj=None):
    current_best = max(ind.fitness.values[0] for ind in population)
    if gen == 0:
        stats.initial_fitness = current_best

```

```

        if current_best > stats.best_fitness:
            stats.best_fitness = current_best
            stats.gen_best_fitness = gen
        stats.current_generation = gen
        elapsed = time.time() - stats.start_time
        if elapsed >= MAX_TIME_SECONDS or gen >= MAX_GENERATIONS:
            stats.termination_cause = "timeout" if elapsed >= MAX_TIME_SECONDS
        else "max_generations"
    return True
return False

# Registro de creator y toolbox
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)
toolbox = base.Toolbox()
toolbox.register("individual", tools.initRepeat, creator.Individual,
                 lambda: random.uniform(LOWER_BOUND, UPPER_BOUND),
INDIVIDUAL_SIZE)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("evaluate", schwefel_function)
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", safe_sbx_crossover, eta=2.0, prob=cx_rate)
toolbox.register("mutate", custom_mutation,
                 indpb=MUTATION_RATE_INDIVIDUAL,
                 bitpb=MUTATION_RATE_BIT,
                 sigma=(UPPER_BOUND - LOWER_BOUND) * 0.1)

# Evolución: generar, evaluar, seleccionar, cruzar, mutar y terminar
population = toolbox.population(n=pop_size)
hof = tools.HallOfFame(1)
# Evaluar población inicial
for ind, fit in zip(population, map(toolbox.evaluate, population)):
    ind.fitness.values = fit
gen = 0
stop_evolution = False
stats.start_time = time.time()
while gen < MAX_GENERATIONS and not stop_evolution:
    offspring = toolbox.select(population, len(population))
    offspring = list(map(toolbox.clone, offspring))
    for i in range(1, len(offspring), 2):
        toolbox.mate(offspring[i-1], offspring[i])
        del offspring[i-1].fitness.values, offspring[i].fitness.values
    for ind in offspring:
        toolbox.mutate(ind)
        if not ind.fitness.valid:
            del ind.fitness.values
    for ind, fit in zip([ind for ind in offspring if not ind.fitness.valid],

```

```

        map(toolbox.evaluate, [ind for ind in offspring if
not ind.fitness.valid])):
    ind.fitness.values = fit
population[:] = offspring
hof.update(population)
stop_evolution = stats_callback(gen, population, hof)
gen += 1

```

En la codificación de DEAP para Schwefel, el punto de partida es la función `schwefel_function`. Para cada individuo (lista de 1.024 valores reales dentro del intervalo [-500,500]), se calcula el valor `raw` según la función descrita en el apartado 3.1. Ese valor se limita a `WORST_CASE_VALUE` para evitar desbordamientos y luego se normaliza dinámicamente en [0,1] mediante `normalized_fitness = 1 - (raw/max(worts_seen, 418,9829*n + n*500))`, de modo que un individuo cuya suma alcance su mínimo global (todas las coordenadas cercanas a 420.9687) obtiene un *fitness* cercano a 1.

Para realizar el cruce, `safe_sbx_crossover` implementa SBX coordinada a coordenada: si dos padres difieren en la coordenada  $i$  por más de  $10^{-10}$ , calcula el factor  $\beta$  con  $\eta = 2$  y genera dos hijos  $c_1$  y  $c_2$ , los cuales se limitan a [-500,500]. Si en algún paso numérico surgen excepciones (desbordamientos, división por cero), ejecuta un cruce aritmético simple para mantener la robustez. Este operador se registra en el `toolbox` con probabilidad de cruce `cx_rate`.

En el bucle evolutivo principal, se registra `FitnessMax` e `Individual` de DEAP de modo que cada individuo es una lista de reales. Con `toolbox`, se vinculan la generación de individuos uniformes en [-500,500], la evaluación `schwefel_function`, la selección por torneo de tamaño 2, el cruce SBX y la mutación gaussiana. Tras crear la población inicial, se evalúan todos los individuos. En cada generación, se seleccionan padres, se clonian, se aplica cruce en pares y se muta cada descendiente. Solo los descendientes sin *fitness* válido se reevalúan, optimizando el costo. Después de reemplazar la población entera con `offspring`, se actualiza el Hall of Fame y se invoca `stats_callback`.

#### 8.2.4.2. Inspyred

Python

```

def generator(random, args):
    size = args.get('individual_size', 1024)
    lower_bound = args.get('lower_bound', -500)
    upper_bound = args.get('upper_bound', 500)
    return [random.uniform(lower_bound, upper_bound) for _ in range(size)]
-----
def schwefel_fitness(candidates, args):
    fitness_values = []

```

```

for candidate in candidates:
    raw_value = 0
    D = len(candidate)
    for i in range(D):
        raw_value += -candidate[i] * np.sin(np.sqrt(abs(candidate[i])))
    raw_value = 418.9829 * D - raw_value

    if raw_value > args['worst_raw_value']:
        args['worst_raw_value'] = raw_value
    if raw_value < args['best_raw_value']:
        args['best_raw_value'] = raw_value

    theoretical_worst = 418.9829 * D + D * 500
    normalization_factor = max(args['worst_raw_value'],
theoretical_worst)
    normalized_fitness = 1.0 - (raw_value / normalization_factor)
    normalized_fitness = max(0.0, min(1.0, normalized_fitness))
    fitness_values.append(normalized_fitness)
return fitness_values

-----
def real_mutation(random, candidates, args):
    p_ind = args.get('mutation_individual_rate', 0.1)
    p_bit = args.get('mutation_bit_rate', 0.1)
    lower_bound = args.get('lower_bound', -500)
    upper_bound = args.get('upper_bound', 500)
    sigma = (upper_bound - lower_bound) * 0.1
    mutated = []
    for cand in candidates:
        if random.random() < p_ind:
            new_cand = cand[:]
            for i in range(len(new_cand)):
                if random.random() < p_bit:
                    delta = random.gauss(0, sigma)
                    new_cand[i] += delta
                    new_cand[i] = max(min(new_cand[i], upper_bound),
lower_bound)
            mutated.append(new_cand)
        else:
            mutated.append(cand)
    return mutated

-----
def safe_sbx_crossover(random, candidates, args):
    crossover_rate = args.setdefault('crossover_rate', 1.0)
    di = args.setdefault('sbx_distribution_index', 2.0)
    children = []
    for i, (p1, p2) in enumerate(zip(candidates[::2], candidates[1::2])):
        if random.random() < crossover_rate:
            c1 = p1[:]

```

```

c2 = p2[:]
for j in range(len(c1)):
    if abs(p1[j] - p2[j]) > 1e-10:
        if p1[j] < p2[j]:
            y1, y2 = p1[j], p2[j]
        else:
            y1, y2 = p2[j], p1[j]
    try:
        beta = 1.0 + (2.0 * (y1 + 500) / (y2 - y1))
        alpha = 2.0 - min(1.0, beta**(di + 1.0))
        u = random.random()
        if u <= 1.0 / alpha:
            beta_q = (u * alpha) ** (1.0 / (di + 1.0))
        else:
            beta_q = (1.0 / (2.0 - u * alpha)) ** (1.0 / (di
+ 1.0))
        c1[j] = 0.5 * ((y1 + y2) - beta_q * (y2 - y1))
        c2[j] = 0.5 * ((y1 + y2) + beta_q * (y2 - y1))
        c1[j] = max(-500, min(500, c1[j]))
        c2[j] = max(-500, min(500, c2[j]))
        if p1[j] > p2[j]:
            c1[j], c2[j] = c2[j], c1[j]
    except (OverflowError, ValueError, ZeroDivisionError):
        blend = random.random()
        c1[j] = blend * p1[j] + (1 - blend) * p2[j]
        c2[j] = (1 - blend) * p1[j] + blend * p2[j]
    children.append(c1)
    children.append(c2)
else:
    children.append(p1)
    children.append(p2)
return children
-----
ea = ec.GA(rand)
ea.selector = inspyred.ec.selectors.tournament_selection
ea.variator = [safe_sbx_crossover, real_mutation]
ea.replacer = inspyred.ec.replacers.generative_replacement
ea.terminator = time_generation_termination
ea.observer = stats_observer

final_pop = ea.evolve(
    generator=generator,
    evaluator=schwefel_fitness,
    pop_size=pop_size,
    maximize=True,
    bounder=inspyred.ec.Bounder(-500, 500),
    max_generations=1000000,
    start_time=start_time,
)

```

```

        max_time=120,
        num_selected=pop_size,
        individual_size=1024,
        crossover_rate=cx_rate,
        mutation_individual_rate=0.1,
        mutation_bit_rate=0.1,
        tournament_size=2,
        lower_bound=-500,
        upper_bound=500,
        sbx_distribution_index=2.0
    )

```

En esta configuración de Schwefel, se establece un algoritmo genético sobre una población de individuos representados como vectores de números reales de dimensión 1.024, cuyos valores están comprendidos en el intervalo [-500,500].

La generación inicial de la población se realiza mediante una función personalizada que utiliza una distribución uniforme dentro de dicho intervalo. La evaluación se lleva a cabo a través de la función de Schwefel, adaptada para maximizar una versión normalizada del valor calculado. Este valor normalizado se obtiene comparando el *fitness* del individuo con el peor valor teórico posible o el peor observado hasta el momento, lo cual permite expresar el progreso evolutivo en términos acotados entre 0 y 1.

El esquema de variación se compone de un operador de cruce SBX (Simulated Binary Crossover) con protección frente a errores numéricos y una mutación gaussiana. El SBX implementado actúa por pares y ajusta los hijos a través de una combinación basada en una distribución controlada, protegiendo los resultados mediante límites y estructuras condicionales ante desbordamientos o divisiones inestables. La mutación, por su parte, modifica genes con una cierta probabilidad si el individuo ha sido previamente seleccionado para mutar, respetando siempre los límites del dominio. Ambos operadores garantizan la viabilidad de los individuos resultantes.

La llamada al método `evolve` constituye el núcleo de la evolución, en la que se configuran todos los parámetros clave: tamaño de población, número máximo de generaciones, duración máxima en segundos, probabilidades de cruce y mutación, tamaño del torneo de selección y límites del espacio de búsqueda. Asimismo, se especifican los operadores personalizados y el mecanismo de selección y reemplazo generacional.

#### 8.2.4.3. ECJ

Java

```

public class RunSchwefel {
    public static void main(String[] args) {

```

```

        String[] ecjArgs = { "-file", "schwefel.params" };
        Evolve.main(ecjArgs);
    }
}

-----
public class SchwefelProblem extends Problem implements SimpleProblemForm {
    public static final double LOW = -500.0;
    public static final double UP = 500.0;
    public static final int N = 1024;
    public static final double F_MAX = 418.9829 * N + N * UP;

    @Override
    public void evaluate(final EvolutionState state, final Individual ind,
                        final int subpopulation, final int threadnum) {
        DoubleVectorIndividual iv = (DoubleVectorIndividual) ind;
        double raw = 418.9829 * iv.genome.length;
        for (int i = 0; i < iv.genome.length; i++) {
            raw -= iv.genome[i] *
            Math.sin(Math.sqrt(Math.abs(iv.genome[i])));
        }
        if (Double.isNaN(raw) || Double.isInfinite(raw)) raw = F_MAX;
        raw = Math.min(raw, F_MAX);
        double scaled = (1.0 - raw / F_MAX);
        scaled = Math.max(0.0, Math.min(1.0, scaled));
        ((SimpleFitness) iv.fitness).setFitness(state, scaled, raw == 0.0);
        iv.evaluated = true;
    }
}

-----
public class TimeConstrainedEvaluator extends SimpleEvaluator {
    public static final long MAX_TIME = 120000;
    private long startTime;
    public boolean timeLimitReached = false;

    @Override
    public void setup(final EvolutionState state, final Parameter base) {
        super.setup(state, base);
        startTime = System.currentTimeMillis();
    }

    @Override
    public void evaluatePopulation(final EvolutionState state) {
        if (timeLimitReached) return;
        super.evaluatePopulation(state);
        if (System.currentTimeMillis() - startTime > MAX_TIME)
            timeLimitReached = true;
    }
}

```

```

-----
@Override
public void setup(final EvolutionState state, final Parameter base) {
    crossoverProb = state.parameters.getDouble(
        new Parameter("pop.subpop.0.species.pipe.source.0.prob"), null,
        0.0);
    mutationProb = state.parameters.getDouble(
        new Parameter("pop.subpop.0.species.mutation-prob"), null, 0.0);
}

pop.subpop.0.size = 64
pop.subpop.0.species.genome-size = 1024
select.tournament.size = 2
pop.subpop.0.species.pipe.source.0.prob = 0.8
pop.subpop.0.species.mutation-prob = 0.1
eval = TimeConstrainedEvaluator
stat = SchwefelStatistics
-----
```

La configuración de Schwefel en ECJ comienza con una clase principal `RunSchwefel`, que invoca la ejecución del sistema de evolución mediante el archivo de parámetros `schwefel.params`. Esta llamada activa la carga del problema, los evaluadores y las configuraciones necesarias para ejecutar el algoritmo evolutivo.

El problema a evaluar está definido en la clase `SchwefelProblem`, que implementa la interfaz `SimpleProblemForm`. Allí se evalúan individuos representados como vectores de números reales (`DoubleVectorIndividual`) de tamaño 1.024, con valores comprendidos en [-500,500]. La función de evaluación calcula el valor clásico de Schwefel, definido según la fórmula expuesta en el apartado 3.1. y lo transforma en un *fitness* escalado entre 0 y 1, donde el valor óptimo 0 se corresponde con un *fitness* del 100 %.

El *fitness* se asigna a cada individuo a través de la clase `SimpleFitness`, con la marca de "ideal" establecida si el valor Schwefel es exactamente 0. Para mantener la estabilidad numérica, se incluyen controles frente a NaN e infinitos, y se impone un límite superior basado en el peor valor teórico esperado. El evaluador personalizado `TimeConstrainedEvaluator` impone un límite de tiempo de 2 minutos para la ejecución del proceso evolutivo. Este evaluador hereda de `SimpleEvaluator` y sobreescribe el método `evaluatePopulation`, verificando si se ha alcanzado el tiempo máximo desde el inicio de la evaluación.

La clase `SchwefelStatistics` se encarga de registrar estadísticas relevantes como el *fitness* inicial, el mejor *fitness* alcanzado, la generación en la que se logró, el tiempo de ejecución y la razón de parada (tiempo, convergencia o límite de generaciones).

Además, se calcula la variación del *fitness* y se registra todo en un archivo CSV junto con los parámetros usados, como la probabilidad de cruce y mutación, que se recuperan

directamente desde el archivo `.params`. En dicho archivo, se establece el tamaño de población en 64, la dimensión de los individuos en 1.024, una probabilidad de cruce de 0.8 y una de mutación de 0.1, utilizando selección por torneo con tamaño 2.

Es menester recordar que, tanto el tamaño de la población como la probabilidad de cruce, se modifican dinámicamente en las ejecuciones que surgen de las configuraciones ya expuestas con anterioridad en esta memoria.

#### 8.2.4.4. ParadisEO

```
C/C++
struct Schwefel : public EO<eoMaximizingFitness> {
    vector<double> x;
    double raw_value;
    Schwefel(size_t n = 1024) : x(n, 0.0), raw_value(DBL_MAX) {}
    void printOn(ostream &os) const override {
        for (double v : x) os << v << " ";
        os << " raw=" << raw_value;
    }
    void readFrom(istream &is) override {
        for (auto &v : x) is >> v;
    }
};

struct SchwefelFunction : public eoEvalFunc<Schwefel> {
    void operator()(Schwefel &ind) override {
        double raw = 418.9829 * ind.x.size();
        for (double xi : ind.x)
            raw -= xi * sin(sqrt(abs(xi)));
        raw = min(raw, 1e10);
        ind.raw_value = raw;
        if (raw > stats.worst_raw_value) stats.worst_raw_value = raw;
        if (raw < stats.best_raw_value) stats.best_raw_value = raw;
        double scaled = (1.0 - (raw / F_MAX));
        ind.fitness(max(0.0, min(1.0, scaled)));
    }
};

struct SchwefelInit : public eoInit<Schwefel> {
    void operator()(Schwefel &ind) override {
        for (double &v : ind.x)
            v = rng.uniform(-500.0, 500.0);
    }
};

struct SafeSBXCrossover : public eoQuad0p<Schwefel> {
    double eta;
    SafeSBXCrossover(double _eta) : eta(_eta) {}
```

```

bool operator()(Schwefel &a, Schwefel &b) override {
    for (size_t i = 0; i < a.x.size(); ++i) {
        if (abs(a.x[i] - b.x[i]) < 1e-10) continue;
        double y1 = min(a.x[i], b.x[i]);
        double y2 = max(a.x[i], b.x[i]);
        try {
            double beta = min(100.0, 1.0 + 2.0 * (y1 + 500) / (y2 -
y1));
            double alpha = 2.0 - min(100.0, pow(beta, eta + 1.0));
            double u = rng.uniform();
            double beta_q = (u <= 1.0 / alpha)
                ? pow(u * alpha, 1.0 / (eta + 1.0))
                : pow(1.0 / (2.0 - u * alpha), 1.0 / (eta + 1.0));
            double c1 = clamp(0.5 * ((y1 + y2) - beta_q * (y2 - y1)),
-500.0, 500.0);
            double c2 = clamp(0.5 * ((y1 + y2) + beta_q * (y2 - y1)),
-500.0, 500.0);
            if (a.x[i] > b.x[i]) { a.x[i] = c2; b.x[i] = c1; } else {
a.x[i] = c1; b.x[i] = c2; }
            } catch (...) {
                double blend = rng.uniform();
                a.x[i] = clamp(blend * a.x[i] + (1 - blend) * b.x[i],
-500.0, 500.0);
                b.x[i] = clamp((1 - blend) * a.x[i] + blend * b.x[i],
-500.0, 500.0);
            }
        }
        return true;
    }
};

struct RealMutation : public eoMonOp<Schwefel> {
    double p_ind, p_bit, sigma;
    RealMutation(double _p_ind, double _p_bit)
        : p_ind(_p_ind), p_bit(_p_bit), sigma((500.0 - (-500.0)) * 0.1) {}
    bool operator()(Schwefel &ind) override {
        if (rng.uniform() >= p_ind) return false;
        bool mutated = false;
        for (double &xi : ind.x) {
            if (rng.uniform() < p_bit) {
                xi += rng.normal() * sigma;
                xi = clamp(xi, -500.0, 500.0);
                mutated = true;
            }
        }
        return mutated;
    }
};

```

```

SchwefelInit init;
SchwefelFunction eval;
SafeSBXCrossover xover(2.0);
RealMutation mutate(0.1, 0.1);
eoDetTournamentSelect<Schwefel> select(2);
eoPop<Schwefel> pop;
for (size_t i = 0; i < popSize; ++i) {
    Schwefel ind(INDIVIDUAL_SIZE);
    init(ind); eval(ind); pop.push_back(ind);
}
size_t gen = 0;
auto t0 = chrono::steady_clock::now();
while (true) {
    auto dt = chrono::duration_cast<chrono::seconds>(
        chrono::steady_clock::now() - t0).count();
    if (dt >= MAX_TIME_SECONDS || gen >= MAX_GENERATIONS) break;
    ++gen;
    sort(pop.begin(), pop.end(), [](const Schwefel& a, const Schwefel& b) {
        return a.fitness() > b.fitness();
    });
    eoPop<Schwefel> offspring(pop.begin());
    while (offspring.size() < popSize) {
        Schwefel p1 = select(pop), p2 = select(pop);
        if (rng.uniform() < 0.8) xover(p1, p2);
        mutate(p1); mutate(p2);
        eval(p1); eval(p2);
        offspring.push_back(p1);
        if (offspring.size() < popSize) offspring.push_back(p2);
    }
    pop = offspring;
}

```

En la implementación de Schwefel con el framework ParadisEO se define un individuo (`Schwefel`) como una estructura que hereda de `EO<eoMaximizingFitness>`, lo que permite trabajar con valores de *fitness* que deben maximizarse. Cada individuo está formado por un vector de números reales (`vector<double> x`) con dimensión fija igual a 1.024. Además del vector de decisión, se almacena el valor bruto de Schwefel calculado durante la evaluación (`raw_value`), útil para monitorizar el progreso del algoritmo.

La función de evaluación está implementada en la clase `SchwefelFunction`, y sigue la formulación estándar del *benchmark* según la fórmula expuesta en el apartado 3.1. con una normalización posterior para convertir el valor a una escala de *fitness* entre 0 y 1. Esta

normalización permite que el algoritmo evolutivo opere en un contexto de maximización, donde 1 representa el valor óptimo.

La población inicial se genera mediante un inicializador personalizado (**SchwefelInit**) que asigna a cada componente del vector un valor aleatorio dentro del dominio permitido, que en el caso de Schwefel es [-500,500].

Para la evolución se utilizan operadores genéticos personalizados. El cruce se realiza mediante un operador SBX robusto (**SafeSBXCrossover**) que implementa la recombinación basada en distribución simulada binaria, con protecciones contra errores numéricos y control de los valores generados para mantenerlos dentro de los límites definidos.

La mutación se aplica mediante un operador (**RealMutation**) que introduce perturbaciones gaussianas sobre los genes con una probabilidad de activación doble: una asociada al individuo completo (**p\_ind**) y otra a cada componente del vector (**p\_bit**). Esta mutación controlada permite ajustar el grado de exploración local sin desestabilizar el proceso evolutivo. La selección de individuos se realiza mediante torneo determinista (**eoDetTournamentSelect**) con un tamaño de torneo igual a 2, lo que ofrece un buen equilibrio entre presión selectiva y diversidad.

El ciclo evolutivo se ejecuta en un bucle que se interrumpe al alcanzar un tiempo máximo de ejecución (120 segundos) o un número máximo de generaciones. En cada generación se evalúan los nuevos individuos, se actualizan las estadísticas y se verifica si se ha alcanzado la convergencia (cuando el valor bruto de Schwefel es cercano a cero).