# Resolución del problema de la mochila mediante la metaheurística PSO acelerada con JAX

Joel Ermantraut[1], Tomas Crisol[1], Ariel Díaz[1], Leandro Balmaceda[1], Adrián Rostagno[1], Santiago Aggio[1], Aníbal Blanco[2], Javier Iparraguirre [1]

[1] Universidad Tecnológica Nacional FRBB
[2] Planta Piloto de Ingeniería Química, Universidad Nacional del Sur-CONICET
8000, Bahía Blanca, Argentina
j.iparraguirre@computer.org

**Resumen.** Un problema muy desafiante de la matemática aplicada moderna es la optimización numérica de problemas NP duros. Muchos problemas de interés a las ciencias e ingenierías caen en esta categoría y resulta deseable contar con algoritmos de búsqueda e implementaciones computacionales, que puedan proporcionar soluciones de aceptable calidad, en tiempos compatibles con aplicaciones prácticas. Una combinación que ha demostrado buenas prestaciones en múltiples aplicaciones son las metaheurísticas aceleradas en placas gráficas (GPU). Sin embargo, las herramientas típicas que permiten la programación de aceleradores demandan mucho conocimiento del hardware por parte del usuario. En este trabajo se explora el desempeño de una implementación de la metaheurística PSO sobre hardware masivamente paralelo a través de un compilador "just in time" llamado JAX. Esta herramienta permite acelerar programas usando un lenguaje de programación orientado a la productividad tal como lo es Python. Se utiliza como caso de estudio el clásico problema de la mochila, el cual admite la generación de instancias arbitrariamente complejas de manera sencilla. Los resultados muestran que es posible lograr aceleraciones de hasta 83 veces empleando como plataforma arquitecturas de computadoras contemporáneas. Adicionalmente, se publican resultados en el uso de unidades tensoriales, los cuales muestran el potencial del uso de estos dispositivos.

**Palabras clave:** Optimización, PSO, Problema de la Mochila, GPU, TPU, JAX.

## 1 Introducción

Los problemas de optimización numérica requieren en muchos casos la exploración de espacios grandes e intrincados, demandando esfuerzos de cómputo elevados para encontrar soluciones óptimas, o incluso simplemente factibles, en tiempos compatibles con aplicaciones prácticas. Por esta razón, en los últimos años se han propuesto innumerables técnicas de exploración para resolver problemas complejos de optimización numérica de manera efectiva. En consecuencia, es posible encontrar numerosas plataformas de modelado y programación que resuelven problemas particulares de forma eficiente.

Sin embargo, el teorema de "no free lunch" en optimización establece que los optimizadores universales son imposibles. En concreto, no existe una estrategia que supere a todas las demás en todos los problemas posibles (Ho y Pepyne, 2002) [5]. Por ello, a pesar de los numerosos esfuerzos para desarrollar algoritmos de optimización generales, sólo se espera que éstos se desempeñen eficientemente en un subconjunto del universo de problemas de optimización.

Existen varias formas de realizar optimización numérica con la computadora. En la última década han cobrado gran popularidad las estrategias estocásticas basadas en evolución de poblaciones (Boussaid y col., 2013, Alba y col., 2013) ([1], [2]). Entre este tipo de metaheurísticas se destacan los "algoritmos genéticos" y la optimización con "enjambres de partículas". Conceptualmente, estas técnicas parten de un conjunto inicial de posibles soluciones en el espacio de búsqueda (población inicial), el cual va evolucionando por medio de reglas relativamente sencillas, usualmente inspiradas en procesos naturales, hasta converger a una población final que contenga la solución buscada.

Se pueden mencionar al menos tres ventajas que ofrecen este tipo de técnicas. En principio, las implementaciones son relativamente simples de programar. Además brindan mucha flexibilidad para ser combinadas con otros algoritmos, extender sus prestaciones y adaptarlos a un gran espectro de problemas. En tercer lugar, numerosas aplicaciones de éxito han sido reportadas en prácticamente todas las áreas de la ingeniería. Entre las debilidades más importantes de estas técnicas, se puede mencionar que se requiere de un gran número de evaluaciones de las funciones involucradas para encontrar soluciones de aceptable calidad. Típicamente se pueden esperar tiempos de ejecución prolongados. En consecuencia, parte de la investigación actual en optimización metaheurística se preocupa por diseñar técnicas eficientes de aceleración de los algoritmos.

Existen diferentes formas de acelerar un algoritmo dado, desde emplear una codificación más eficiente, adoptar un lenguaje de programación más "rápido", o paralelizar la implementación para que pueda ser ejecutado en hardware específico. En particular, los algoritmos de optimización basados en poblaciones se pueden categorizar como "naturalmente paralelos" y son aptos para ser ejecutados en aceleradores modernos, en particular las Unidades de Procesamiento Gráfico o GPUs por sus siglas en inglés (Mussi y col., 2011) [10].

En este trabajo se presenta una implementación del algoritmo de enjambre de partículas, conocido como PSO por sus siglas en inglés, utilizando la librería JAX[1]. Gracias a los beneficios de JAX, el mismo código escrito en el lenguaje de programación Python[2] se ejecutó en CPU (procesadores de propósito general), GPU y TPU (unidad de procesamiento tensorial). Como caso de estudio se empleó el problema de la mochila, un problema conocido de optimización combinatoria con múltiples aplicaciones prácticas. Como resultado, se han reportado ejecuciones hasta 83 veces más rápido que la implementación serie. Además, el código relacionado con la presente publicación se

[1] https://github.com/google/jax

[2] https://www.python.org/

publica en formato de código abierto con el fin de facilitar la colaboración entre investigadores interesado en la temática.

## 2 Problema de la mochila

El problema de la mochila es uno de los problemas de la optimización combinatoria más estudiados. Una formulación típica se presenta a través de la Ec. 1, donde $xj\in\{0,1\}$ representa variables binarias y $p_j$, $w_{ij}$, $M_i$ son parámetros. Los índices $j\in\{1,\ldots, n\}$ e $i\in\{1, \ldots,m\}$ corresponden a ítems y restricciones respectivamente. Físicamente la Ec. (1) describe el problema de seleccionar un número de ítems a partir de un conjunto dado para ser trasladados en una mochila, de manera de maximizar el valor total de los elementos seleccionados, verificando además una serie de restricciones que podrían asociarse a la capacidad de máxima de la mochila para contenerlos (volumen), al peso máximo que podría soportar la mochila, etc.

$$\mathrm{Max}_x \sum_j p_j x_j \text{ sujeto a } \sum_j w_{ij} x_j \leq M_i \quad (1)$$

Se trata de un problema discreto lineal que pertenece a la familia de problemas NP-duros. Debido a su gran interés, tanto teórico como práctico, este problema en sus diferentes versiones ha recibido una considerable atención por parte de comunidad de la Investigación Operativa a nivel internacional (Chu y Beasley, 1998) [3]. En particular, Zan y Jaros (2014) [15], lo han utilizado para testear el desempeño de un algoritmo PSO, paralelizado sobre GPU. Siguiendo la metodología de estos autores, en este trabajo se emplea una versión que penaliza las restricciones de acuerdo a la Ec. (2).

$$\mathrm{Max}_x \{\sum_j p_j x_j - P \sum_i \max[0, \sum_j (w_{ij} x_j - M_i)]\} \quad (2)$$

El valor P es un factor de penalización de la violación de las restricciones y se le asigna en este trabajo el valor $P=\sum_j p_j$ para todos los experimentos en este estudio. Una característica interesante de este modelo es la posibilidad de diseñar instancias arbitrariamente grandes y complejas empleando las siguientes fórmulas para la generación de los parámetros involucrados en la función objetivo y restricciones (Chu y Beasley, 1998) [3]: $wij=rand(0,1000)$; $Mi=0{,}75\sum j\ wij$, $pj=\{\sum j\ wij/m+500.rand(0,1)\}$, donde rand(l,u) es un número del intervalo [l,u] generado aleatoriamente.

## 3 Metaheurística PSO

La optimización por enjambre de partículas (PSO, por sus siglas en inglés) es una metaheurística basada en poblaciones inspirada en los principios de exploración empleados por los enjambres de insectos en la búsqueda de alimento (Marini y Walzack, 2015) [8]. Desde su propuesta inicial por Kennedy y Eberhart (1995) [6], esta técnica ha sido progresivamente mejorada y aplicada en infinidad de problemas debido a su sencillez de programación, flexibilidad y buenas características de rapidez y convergencia (Zhang y col., 2015) [16].

El algoritmo PSO se rige por dos ecuaciones básicas. La posición (Ec. 3) de cada individuo de la población **z**, se modifica en cada iteración (k) por medio de un término de velocidad **v**. El periodo de tiempo se asume igual a la unidad y por lo tanto no se explicita en (3). El término de velocidad (Ec. 4) posee a su vez un componente inercial que depende de la velocidad en el instante anterior, un término individual o cognitivo que dirige a la partícula al mejor lugar identificado por ella misma durante su recorrido hasta el momento (**p**) y un término social que dirige a la partícula a la mejor posición encontrada por todo el enjambre hasta el momento (**g**).

$$\mathbf{z}^{k+1} = \mathbf{z}^k + \mathbf{v}^k \quad (3)$$
$$\mathbf{v}^k = w\,\mathbf{v}^k + c_1\mathbf{r}_1\,(\mathbf{p}^k-\mathbf{z}^k)+c_2\mathbf{r}_2\,(\mathbf{g}^k-\mathbf{z}^k) \quad (4)$$

Las matrices **z**, **v**, **p** y **g** poseen tantas columnas como dimensiones (variables) tiene el problema de optimización (D) y tantas filas como partículas tienen la población (Np). Cabe aclarar que la matriz **g** posee todas sus filas idénticas e iguales a la posición del mejor individuo del enjambre en su historia y desde el punto de vista de la estructura de datos puede tratarse directamente como un vector.

Las constantes w, $c_1$ y $c_2$ son parámetros propios del método y las matrices $\mathbf{r}_1$ y $\mathbf{r}_2$ tienen elementos aleatorios entre 0 y 1 que proporcionan la componente estocástica a la búsqueda. El valor de la función objetivo y el grado de verificación de las restricciones del problema se encuentra de manera implícita en los términos **p** y **g**. Dado que la formulación anterior proporciona valores continuos de las variables, Kennedy y Eberhard (1997) [7] propusieron una versión para la formulación discreta (binaria), consistente en las siguientes transformaciones (Ecs. 5 y 6). Estas ecuaciones tienen básicamente la función de “redondear” a 0 o a 1 las variables continuas resultantes de las ecuaciones (3) y (4).

$$S(v_{ij})=1/(1+\exp(-v_{ij})) \quad (5)$$
$$x_{ij}=1 \text{ si } rand(0,1) < S(v_{ij}), 0 \text{ si no} \quad (6)$$

Actualmente existen numerosas implementaciones de este tipo de metaheurística. En particular, el proyecto PySwarms[3] se destaca por la madurez y la correcta documentación. El grupo de investigación que nuclea a los autores de la presente publicación, viene trabajando desde hace unos años con diferentes formulaciones y aplicaciones de PSO (Montain et al., 2013; Damiani y col., 2019; Salmieri y col., 2019) ([9], [4], [13]).

Según nuestra experiencia en la investigación en el área, no hemos encontrado una implementación de PSO en Python de código abierto que pueda ejecutarse en arquitecturas masivamente paralelas tales como GPUs y TPUs. Con el objetivo de desarrollar estudios de cómputo paralelo aplicados a optimización basada en metaheurísticas utilizando los dispositivos mencionados, se consolidó una versión que implementa las ecuaciones (4)-(6) en Python empleando funciones puras de Numpy que denominaremos en lo sucesivo PSO-BHI.

[3] https://pyswarms.readthedocs.io/en/latest/

## 4 Aceleración en CPU, GPU y TPU

En lo que concierne a la aceleración del algoritmo, durante los últimos años el cómputo de propósito general usando GPUs ha cobrado una gran relevancia (Nickolls y Dally, 2010) [11]. Las GPUs son procesadores masivamente paralelos, de relativo bajo costo, que se han creado originalmente para acelerar juegos de computadora 3D. Actualmente, han encontrado numerosas aplicaciones en computación científica. El rápido crecimiento de este tipo de dispositivos ha permitido gran poder de cómputo en estaciones de trabajo de uso general y han contribuido a un salto de desempeño de al menos un orden de magnitud en el poder de cálculo (Papadrakakis y col., 2011) [12]. Las arquitecturas de GPU contemporáneas disponen de cientos de procesadores con capacidad de ejecutar tareas en forma paralela permitiendo acelerar una gran cantidad de aplicaciones. Debido a estas interesantes prestaciones, en los últimos años se las ha utilizado para mejorar el desempeño de muchas metaheurísticas (Tan y Ding, 2016) [14].

Existen varias formas de extraer el potencial que brindan las GPU. Una manera es programar directamente empleando la API (Application Programming Interface) que proporciona el fabricante del hardware, por ejemplo, la plataforma CUDA[4] desarrollada por la empresa Nvidia para las placas gráficas de esa marca. Alternativamente es posible utilizar OpenCL[5] que permite aprovechar la arquitectura paralela de cualquier procesador. Ambos métodos requieren del programador un conocimiento profundo de la arquitectura de la GPU, nociones de asignación de memoria y manejo de lenguaje de programación C. Aunque el resultado del uso de CUDA produce implementaciones eficientes, es deseable desde el punto de vista del programador el uso de lenguajes orientados a productividad como Python. Es posible encontrar proyectos que buscan facilitar este proceso, por ejemplo, la plataforma PyCuda[6]. En nuestra experiencia (Damiani y col., 2019) [4] aún demanda el conocimiento del manejo de recursos paralelos dentro del GPU.

Recientemente, han surgido paradigmas de compilación “just in time” (o JIT) para acelerar código Python sin necesidad de interactuar con los aceleradores de manera directa. Entre estos compiladores se destacan Numba[7] y JAX[8]. En este caso, el código desarrollado en Python/Numpy[9] es convertido a código acelerado mediante el uso de “decoradores”. La compilación JIT genera código específico destinado al hardware en que se va a ejecutar de manera automática. Esta alternativa permite focalizarse exclusivamente en el desarrollo y extraer el potencial del acelerador sin necesidad de programarlo directamente.

En este trabajo se emplea JAX, que además de soportar GPUs, posibilita el empleo de los múltiples núcleos de los CPUs y además puede ejecutar programas en TPUs. Los

---

[4] https://developer.nvidia.com/cuda-zone
[5] https://www.khronos.org/opencl/
[6] https://documen.tician.de/pycuda/
[7] http://numba.pydata.org/
[8] https://jax.readthedocs.io/en/latest/notebooks/quickstart.html
[9] https://numpy.org/

TPUs son chips diseñados por Google[10] para soporte específico de TensorFlow[11], una librería orientada al desarrollo de proyectos de aprendizaje automático.

## 5 Casos de estudio

Para investigar la velocidad de cómputo de diferentes versiones de algoritmos PSO se compararon las siguientes implementaciones de la metaheurística:

- PySwarms (CPU, ejecución en serie)
- PSO-BHI (CPU, ejecución en serie)
- PSO-BHI JAX (CPU, ejecución en paralelo)
- PSO-BHI JAX GPU (GPU, ejecución en paralelo)
- PSO-BHI JAX TPU (TPU, ejecución en paralelo)

Para todas las implementaciones se utilizaron las instancias del problema de la mochila indicadas en la Tabla 1. Allí se reportan el número de ítems y el número de restricciones, los cuales corresponden a problemas progresivamente más grandes. Respecto del número de partículas del enjambre (Np), no existe un consenso sobre el número óptimo para cada tipo de problema. Diferentes estudios indican que, si bien problemas más complejos suelen requerir una mayor cantidad de individuos, la capacidad de exploración del algoritmo no se incrementa significativamente más allá de cierto número. Para los problemas P1 a P4 se utilizó el heurístico $Np=10+NroItems^{0.5}$ para proporcionar enjambres de tamaño apropiado a cada instancia. A su vez, para testear el desempeño de los algoritmos con otros tamaños de poblaciones, se realizaron experimentos adicionales con la instancia P4 empleando número de individuos crecientes (problemas P41, P42 y P43). Finalmente, los problemas P5 y P6 corresponden a instancias muy grandes que resultan restrictivas para los recursos disponibles en ciertas implementaciones.

En todos los casos el algoritmo se ejecutó un número máximo de iteraciones proporcional al tamaño del problema específico. El valor de Kmax adoptado en cado caso fue investigado mediante experimentos independientes hasta identificar un número adecuado a partir del cual no se observaron mejoras significativas en los valores de la función objetivo.

Finalmente, para facilitar las comparaciones y la programación de todas las opciones se utilizó Colab[12]. Cabe destacar que Colab es un ambiente ofrecido por Google que permite programar y ejecutar código Python desde el navegador de internet y utilizar los recursos de cómputo paralelo proporcionados por Google, en particular GPUs y TPUs.

**Tabla 1**. Instancias del problema de la mochila

[10] https://cloud.google.com/tpu/docs
[11] https://www.tensorflow.org/
[12] https://colab.research.google.com/notebooks/intro.ipynb#recent=true

| Problema | Nro. Ítems | Nro. Restricciones | Np | Kmax |
|---|---|---|---|---|
| P1 | 20 | 5 | 15 | 100 |
| P2 | 200 | 50 | 25 | 150 |
| P3 | 2000 | 500 | 55 | 200 |
| P4 | 10000 | 1000 | 110 | 300 |
| P41 | 10000 | 1000 | 220 | 300 |
| P42 | 10000 | 1000 | 440 | 300 |
| P43 | 10000 | 1000 | 880 | 300 |
| P5 | 20000 | 1000 | 440 | 300 |
| P6 | 50000 | 1000 | 440 | 300 |

# 6 Resultados

A continuación, se presentan los resultados obtenidos de aplicar cada una de las implementaciones de PSO descriptas a las diferentes instancias del problema de la mochila de la Tabla 1. El código y los experimentos pueden consultarse y reproducirse desde un sitio público[13]. Debido a la componente aleatoria de la búsqueda metaheurística, para proporcionar resultados estadísticamente significativos, cada problema se ejecutó 30 veces. Los resultados reportados en esta sección, corresponden entonces al promedio de las 30 optimizaciones, tanto en lo que respecta a los tiempos de cómputo, como a los valores de la función objetivo. En la Tabla 2 se proporcionan los valores de la función objetivo (Ec. 2). En todos los casos los valores corresponden a soluciones factibles, esto es, la violación de las restricciones es cero. Sin embargo, no es posible asegurar que las soluciones encontradas sean globalmente óptimas. Las discrepancias en los valores obtenidos se atribuyen a que las diferentes búsquedas producen soluciones algo diferentes debido a la componente estocástica de la exploración.

Los experimentos reportados en este trabajo se realizaron sobre el hardware provisto por Colab. Dependiendo del usuario y el momento de disponibilidad de los recursos, el hardware puede variar. Los resultados se obtuvieron sobre un CPU Intel Xeon @ 2.20GHz, 13 GB de memoria RAM, un GPU Tesla T4 con 16 GB de RAM y un TPU de segunda generación.

[13] El código fuente está disponible públicamente en https://github.com/BHI-Research/PSO-JAX/tree/master/examples/knapspack.

**Tabla 2**. Función objetivo

| Problema | PySwarms | PSO-BHI | JAX | JAX GPU | JAX TPU |
|---|---|---|---|---|---|
| P1 | 9601 | 9268 | 9224 | 9238 | 9363 |
| P2 | 101758 | 108472 | 108406 | 108794 | 108991 |
| P3 | 933472 | 1107598 | 1107884 | 1107961 | 1106932 |
| P4 | 4614806 | 5584111 | 5580013 | 5580680 | 5578310 |
| P41 | - | 5582846 | 5582829 | 5583981 | 5582138 |
| P42 | - | 5586368 | 5584218 | 5585590 | 5586682 |
| P43 | - | 5588383 | 5590125 | 5589315 | 5588977 |
| P5 | - | 11163628 | 11166246 | 11169239 | 11186215 |
| P6 | - | 27972928 | 27973334 | 28017744 | 28010250 |

**Tabla 3**. Tiempos de cómputo (segundos)

| Problema | PySwarms | PSO-BHI | JAX | JAX GPU | JAX TPU |
|---|---|---|---|---|---|
| P1 | 0.49 | 0.64 | 1.01 | 1.1 | 8.7 |
| P2 | 0.85 | 0.99 | 1.51 | 1.5 | 13.6 |
| P3 | 12.4 | 2.89 | 3.16 | 3 | 16.7 |
| P4 | 593.9 | 31 | 30.1 | 3.3 | 26.3 |
| P41 | - | 50.9 | 48.9 | 3.4 | 25.6 |
| P42 | - | 89.07 | 85.41 | 3.28 | 27.4 |
| P43 | - | 168.1 | 165.1 | 3.87 | 27.8 |
| P5 | - | 175.09 | 173.08 | 3.2 | 28.6 |
| P6 | - | 424.3 | 434.1 | 5.1 | 34.5 |

En la Tabla 3 se reportan los tiempos de cómputo promedio en segundos. Una primera lectura permite observar que la implementación PSO-BHI (Numpy, cómputo en serie) reporta tiempos muy inferiores a la versión de PySwarms para los problemas P3 y P4. Por otra parte, si se toma como referencia la solución proporcionada por PySwarms (Tabla 2), las soluciones halladas por nuestra metaheurística (PSO-BHI) son, en promedio, un 3.4% inferiores para el problema P1 pero un 6.8, 18.7 y 20.9 % superiores en los problemas P2, P3 y P4, respectivamente. Para el resto de los problemas PySwarms ya no proporciona soluciones por excederse la memoria del sistema.

Para analizar las aceleraciones logradas sobre el algoritmo PSO-BHI, se calculan los speedup (SU) obtenidos. El SU (7) se define como el cociente entre el tiempo de la versión serie (ts) y el correspondiente a la versión acelerada (ta).

$$SU=ts/ta \quad (7)$$

En la Fig. 1 se presentan gráficamente los speedup conseguidos. Como implementación de referencia de empleó PSO-BHI (NP CPU). La implementación de JAX ejecutada en CPU (barra azul) no presenta variaciones importantes respecto la implementación de referencia. Se evidencia solo una leve ralentización para los problemas más pequeños (SU<1.0) y una leve aceleración (SU>1.0) en las instancias mayores.

La versión JAX GPU (barra naranja) produce leves ralentizaciones en los problemas más chicos (P1 y P2) respecto a la referencia debido básicamente a que las demoras entre la transferencia de datos entre CPU y GPU no alcanza a compensar el paralelismo. Para el problema P3 se evidencia una leve aceleración y para el P4 ya se obtiene un notable SU de aproximadamente 10.0. La implementación en TPU produce básicamente una ralentización en las instancias P1 a P3 y solo en el caso P4 se logran equiparar los tiempos correspondientes a la versión de referencia. Esto se atribuye fundamentalmente a que debido a la relativa pequeña escala de estos los problemas no se logra aprovechar la arquitectura del dispositivo.

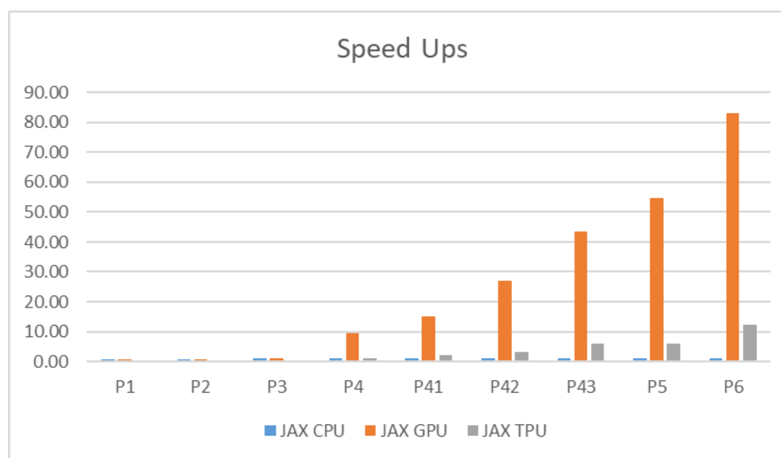


**Fig. 1**. Aceleraciones logradas por cada implementación.

Con el objeto de estudiar el desempeño de las distintas implementaciones frente a tamaños de población crecientes se realizaron tres experimentos adicionales sobre el problema P4 (P41, P42 y P43). De la Tabla 2 se observa que el incremento en el tamaño de la población contribuye a mejorar la función objetivo encontrada por el algoritmo PSO-BHI en la gran mayoría de los casos. Como era de esperarse, se obtienen importantes aceleraciones de la versión JAX GPU (barra naranja) para las instancias P41, P42 y P43 con SU cercanos a 15.0, 27.0 y 43.0 respectivamente. Es interesante el hecho de que la versión JAX TPU también presenta aceleraciones superiores a 2.0, 3.0 y 6.0 para estos problemas. Evidentemente la aceleración del TPU comienza a manifestarse en instancias importantes en términos de número de pobladores.

La representación gráfica de los tiempos de la serie de problemas P4 (Fig. 2) permite observar que las velocidades de cómputo en los dispositivos paralelos GPU (línea naranja) y TPU (línea gris) se mantienen prácticamente invariantes con el número de individuos que conforman la población. Este comportamiento, reportando en numerosos estudios de metaheurísticas aceleradas en GPU, es una excelente característica en aquellos problemas donde la eficiencia de la búsqueda y por lo tanto la calidad de la solución, mejora con el tamaño de la población.

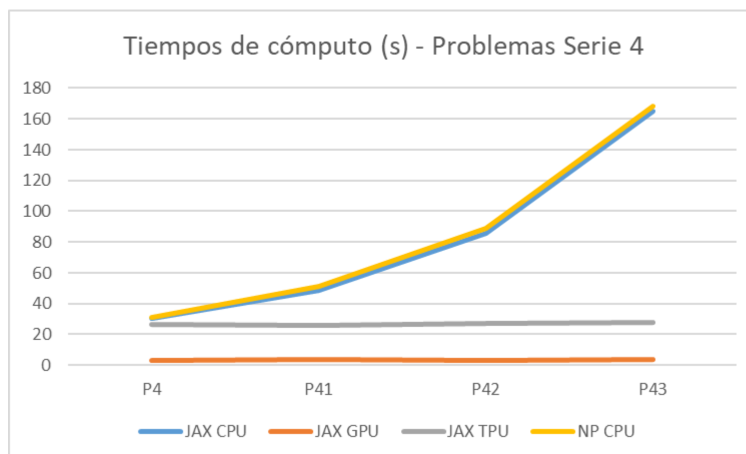


**Fig. 2**. Tiempos de cómputo de los problemas de la serie P4

Finalmente, los problemas P42, P5 y P6 se diferencian básicamente por el número de variables presente (número de ítems), manteniéndose invariantes el número de restricciones y el tamaño de la población (440 individuos). Puede apreciarse (Fig. 1) que los SU obtenidos con CPU (barra naranja) son de aproximadamente 27.0, 54.0 y 83.0 para P42, P5 y P6 respectivamente. A su vez, con TPU (barra gris) los SU alcanzados son de aproximadamente 3.0, 6.0 y 12.0 para P42, P5 y P6 respectivamente. Evidentemente, la posibilidad de aceleración del algoritmo también depende significativamente del tamaño de la instancia del problema bajo estudio.

# 7 Conclusiones

A partir de los experimentos realizados en el presente trabajo, es posible concluir que es posible acelerar significativamente implementaciones de PSO sobre GPU y TPU usando un lenguaje orientado a la productividad. Dichas aceleraciones se manifiestan en tamaños de instancias "grandes". En problemas pequeños dominan los tiempos de transferencia de datos entre dispositivos por sobre la ganancia por paralelizar las operaciones. JAX proporciona una manera eficiente y transparente de hacerlo. Es importante tener en cuenta que la implementación debe programarse en términos de funciones "puras" y se empleen estructuras de datos apropiadas de Numpy.

Se observó que los tiempos de cómputo que demanda nuestra implementación PSO-BHI son inferiores en la mayoría de los casos a los de la implementación del estado del arte PySwarms para instancias pequeñas y medianas (problemas P1 a P4). Para instancias más grandes (P41 a P6), PySwarms ya no proporcionó soluciones por excederse la memoria disponible, mientras que las versiones PSO-BHI si lo hicieron.

En lo que respecta a las aceleraciones propiamente dichas, se comprobó la tendencia reportada en muchos trabajos sobre metaheurísticas aceleradas relacionados con un notable incremento del SU a medida que crece el número de individuos en la población. Este comportamiento es positivo en problemas en los cuales un tamaño de población

mayor se traslada efectivamente a una mayor capacidad exploratoria del algoritmo y por lo tanto a la identificación de soluciones de mejor calidad.

Finalmente se concluye que la plataforma Colab proporciona una excelente herramienta para programar en Python/Numpy/JAX y utilizar de manera transparente el hardware proporcionado para experimentación. El intercambio con investigadores se facilita a partir de repositorios con código abierto y soluciones como Colab de libre acceso.

## Referencias

1. Alba, E., Luque, G., Nesmachnow, S.: Parallel metaheuristics: recent advances and new trends. International Transactions in Operational Research, 20(1), 1-48 (2013).
2. Boussaïd, I., Lepagnot, J., Siarry, P.: A survey on optimization metaheuristics. Information Sciences, 237, 82-117 (2013).
3. Chu, P.C., Beasley, J.E.: A genetic algorithm for the multidimensional knapsack problem. Journal of Heuristics, 4 (1), 63–86 (1998).
4. Damiani, L., Diaz, A.I., Iparraguirre J., Blanco A.M.: Accelerated particle swarm optimization with explicit consideration of model constraints; Cluster Computing; https://doi.org/10.1007/s10586-019-02933-1. (2019)
5. Ho, Y. C., Pepyne, D. L.: Simple explanation of the no-free-lunch theorem and its implications. Journal of optimization theory and applications, 115(3), 549-570 (2002).
6. Kennedy, J., Eberhart, R.: Particle Swarm Optimization, in IEEE International Conference on Neural Networks, 4, 1942–1948 (1995).
7. Kennedy, J., Eberhart R.: A discrete binary version of the particle swarm algorithm, in IEEE International Conference on Systems, Man, and Cybernetics, 5, 4104–4108 (1997).
8. Marini, F., Walczak, B. Particle swarm optimization (PSO). A tutorial. Chemometrics and Intelligent Laboratory Systems, 149, 153-165 (2015).
9. Montain, M. E., Blanco, A. M., Bandoni, J. A.: Optimal drug infusion profiles using a particle swarm optimization algorithm. Computers & Chemical Engineering, 82, 13-24 (2015).
10. Mussi, L., Daolio, F., Cagnoni, S.: Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture. Information Sciences, 181(20), 4642-4657 (2011).
11. Nickolls, J., & Dally, W. J.; The GPU computing era. IEEE micro, 30(2) (2010).
12. Papadrakakis, M., Stavroulakis, G., Karatarakis, A.: A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures. Computer Methods in Applied Mechanics and Engineering, 200(13-16), 1490-1508 (2011).
13. Salmieri, L., Frutos, M., Iparraguirre, J., Blanco, A.M.: Optimización PSO paralelizada para scheduling de flow-shop; Memorias SIIIO, JAIIO 2019, Salta, Argentina (2019).
14. Tan, Y., Ding, K.: A survey on GPU-based implementation of swarm intelligence algorithms. IEEE transactions on cybernetics, 46(9), 2028-2041 (2016).
15. Zan D., Jaros J.: Solving the Multidimensional Knapsack Problem using a CUDA accelerated PSO. IEEE Congress on Evolutionary Computation (2014).
16. Zhang, Y., Wang, S., Ji, G.: A Comprehensive Survey on Particle Swarm Optimization Algorithm and Its Applications. Mathematical Problems in Engineering Volume 2015, Article ID 931256. (2015).