Optimización del algoritmo de Filtro de Partículas (Sequential Monte Carlo) mediante herramientas de Computación de alto rendimiento.

Sebastián Añazco Niklitschek

Abstract

En este estudio se implementó un filtro de partículas utilizando diferentes enfoques de High Performance Computing (HPC) para evaluar su escalabilidad en CPU y GPU. Partiendo de una versión base en Python secuencial (418.1 s para 1M partículas), se desarrollaron variantes optimizadas: vectorización con NumPy (60.9 s), Cython (6.9 s), *C++* mediante CBindings (44.9 s) y una versión paralelizada en Cython (5.8 s). Los resultados demostraron que la combinación de compilación (Cython/C++) y paralelización (multinúcleo CPU) logró las mayores mejoras, reduciendo el tiempo de ejecución en hasta dos órdenes de magnitud (97x más rápido para 10M partículas). Esto confirma que el uso estratégico de herramientas HPC permite optimizar significativamente algoritmos complejos como el filtro de partículas, siendo clave la selección de técnicas según los recursos disponibles (CPU/GPU) y el tamaño del problema. La escalabilidad observada sugiere que para conjuntos de datos más grandes, enfoques híbridos (Python para prototipado + C++/Cython para producción) podrían ofrecer el mejor balance entre desarrollo y rendimiento.

Index Terms

Filtro de partículas, HPC, CPU, GPU, Python, C++.

I. Introduction

LOS filtros de partículas se utilizan ampliamente en problemas de estimación de estados, como en el seguimiento de objetos, navegación robótica o reconstrucción de trayectorias, donde la incertidumbre del sistema y del sensor dificulta una solución determinista. Sin embargo, la implementación secuencial tradicional de estos algoritmos presenta limitaciones significativas en contextos donde se requiere un gran número de partículas o una alta frecuencia de actualización. Estas restricciones se vuelven especialmente problemáticas en aplicaciones en tiempo real o en sistemas con recursos computacionales limitados.

Una de las principales *dificultades* es el **alto costo computacional** asociado al cálculo de pesos, la propagación de partículas y el proceso de remuestreo, lo que ha sido ampliamente documentado en estudios previos [1][2]. Además, cuando se emplean entornos interpretados como Python, la sobrecarga adicional del intérprete y el Global Interpreter Lock (GIL) obstaculizan aún más el aprovechamiento del paralelismo [3]. Por otro lado, las soluciones vectorizadas con bibliotecas como NumPy ofrecen mejoras significativas, pero alcanzan un límite de eficiencia al no aprovechar de forma explícita los núcleos disponibles del procesador.

Frente a este escenario, surge la necesidad de explorar estrategias que permitan acelerar la ejecución de estos algoritmos sin comprometer la precisión de las estimaciones. En este contexto, las herramientas de Computación de Alto Rendimiento (HPC) como **Cython**, extensiones en **C/C++ OpenMP** y permiten paralelizar regiones críticas del código, superando las limitaciones del enfoque secuencial. Diversos trabajos han demostrado la efectividad de estas técnicas para reducir los tiempos de simulación en sistemas probabilísticos complejos [4].

En este informe se aborda este desafío mediante la implementación progresiva de un filtro de partículas en 2D, aplicando optimizaciones en distintos niveles del stack tecnológico. A partir de una versión básica en Python puro, se comparan mejoras con NumPy, Cython y paralelización tanto en CPU como en GPU, evaluando sus diferencias de rendimiento. Y por tanto, se formula la siguiente **Pregunta de investigación:**

¿Es posible optimizar el rendimiento de un filtro de partículas en 2D utilizando herramientas de Computación de Alto Rendimiento (HPC)?

II. Estado del arte

Actualmente, los filtros de partículas son ampliamente utilizados para el rastreo de estados en sistemas estocásticos. Su versatilidad les permite adaptarse a contextos en los que la incertidumbre y la no linealidad hacen inviable el uso de soluciones analíticas exactas. En particular, han sido aplicados en sistemas de navegación, monitoreo ambiental, bioinformática, y más recientemente, en el rastreo de vehículos aéreos no tripulados (UAVs) [5].

En cuanto al ámbito técnico, los filtros de partículas se ubican en el área de la inteligencia artificial aplicada y la inferencia bayesiana. Dentro de este campo, su aplicación en tareas de percepción sensorial y localización los posiciona como herramientas clave en robótica y sistemas autónomos. Un caso de estudio representativo es el uso del filtro de partículas para el seguimiento

de drones a partir de señales de radar o imágenes aéreas. En este ejemplo, cada partícula representa una hipótesis sobre la ubicación del dron, y el sistema actualiza estas hipótesis a medida que recibe nueva información sensorial [6].

Algorithm 1: Filtro de partículas general con secciones paralelizables

Input: N (número de partículas), T (número de pasos de tiempo)
Output: Estimación de posición final

```
inicializar_partículas(N)
for t ← 1 to T do
    predecir_partículas()                          // Movimiento + ruido
    z ← obtener_observación()
    for i ← 1 to N do
        calcular_peso(p[i], z)                     // (1) Paralelizable
    end
    normalizar_pesos()
    resample ← remuestreo_sistemático()            // (2) Paralelizable
    redistribuir_partículas(resample)
end
return estimación
```

III. Solución propuesta

El objetivo principal de esta investigación es evaluar si es posible mejorar el rendimiento de un filtro de partículas utilizando herramientas de High Performance Computing (HPC). Para ello, se propone implementar y comparar distintas versiones del algoritmo, aplicando estrategias de optimización tanto a nivel de software como de hardware. La solución se desarrollará en cinco etapas de optimización sobre una base secuencial escrita inicialmente en Python puro.

A. Python secuencial

Se parte de un algoritmo de filtro de partículas escrito completamente en Python, con bucles for y estructuras básicas, el cual sirve como referencia de rendimiento y claridad.

B. Python vectorizado

La primera mejora aplica operaciones vectorizadas con NumPy para ESULTADOSacelerar etapas como la predicción de partículas, el cálculo de distancias y el peso de cada partícula.

C. Funciones C++ (C bindings)

Para las partes más críticas del algoritmo, como el cálculo de pesos o el remuestreo, se implementan versiones equivalentes en C y se integran usando ctypes o cffi. Esta técnica permite comparar directamente el impacto del lenguaje compilado en el rendimiento.

D. Cython y Paralelizado en CPU

Luego, se traslada el código a Cython, optimizando los tipos estáticos y desactivando verificaciones de límites. Posteriormente, se aplica paralelización con prange y directivas de OpenMP, permitiendo distribuir la carga de trabajo sobre múltiples núcleos de CPU.

E. Paralelizado en GPU

Finalmente, se utiliza la herramienta Hidet, que permite convertir funciones Python en kernels CUDA automáticamente. Esto permite llevar la ejecución masivamente paralela a GPUs, lo cual es ideal para tareas como el cálculo de pesos o el remuestreo en grandes cantidades de partículas.

IV. METODOLOGÍA

Para evaluar el rendimiento de cada optimización aplicada al filtro de partículas, se diseñó un conjunto de experimentos controlados. En todos los casos, se mantuvo la misma lógica algorítmica para asegurar una comparación justa. Las pruebas se realizaron en un entorno controlado, ejecutando cada versión con diferentes cantidades de partículas: 1.000.000, 5.000.000 y 10.000.000. Cada experimento se repitió 10 veces para reducir el impacto del ruido estadístico, y se registró el tiempo promedio de ejecución como principal métrica de desempeño. Además, se verificó que la precisión estimada del algoritmo se mantuviera estable en todas las versiones, evaluando que las optimizaciones no afectaran el resultado esperado del filtro. A su vez, se seleccionó una cantidad de partículas definida, 5.000.000, y se probó con cada uno de los algoritmos para obtener una curva de rendimiento entre todas las opciones.

Las pruebas de rendimiento y desarrollo del algoritmo de filtro de partículas se realizaron en un entorno con las siguientes características:

- Sistema Operativo: Ubuntu 22.04 LTS (64-bit)
- Procesador (CPU): AMD Ryzen 7 5700U with Radeon Graphics (8 núcleos / 16 hilos)
- Memoria RAM: 5 GB DDR4
- Unidad de almacenamiento: SSD NVMe de 512 GB
- Tarjeta Gráfica (GPU): NVIDIA RTX A4000 *(no utilizada en pruebas locales)*
- Versión de Python: 3.12.2 (entorno Conda)
- Compilador C: GCC 11.4.0 con soporte OpenMP

Todo el código fue ejecutado en un entorno aislado mediante Conda, bajo el ambiente llamado **hpc**, **excepto** la ejecución paralela en GPU, la cual se realizó en el servidor remoto Patagón.

V. Resultados

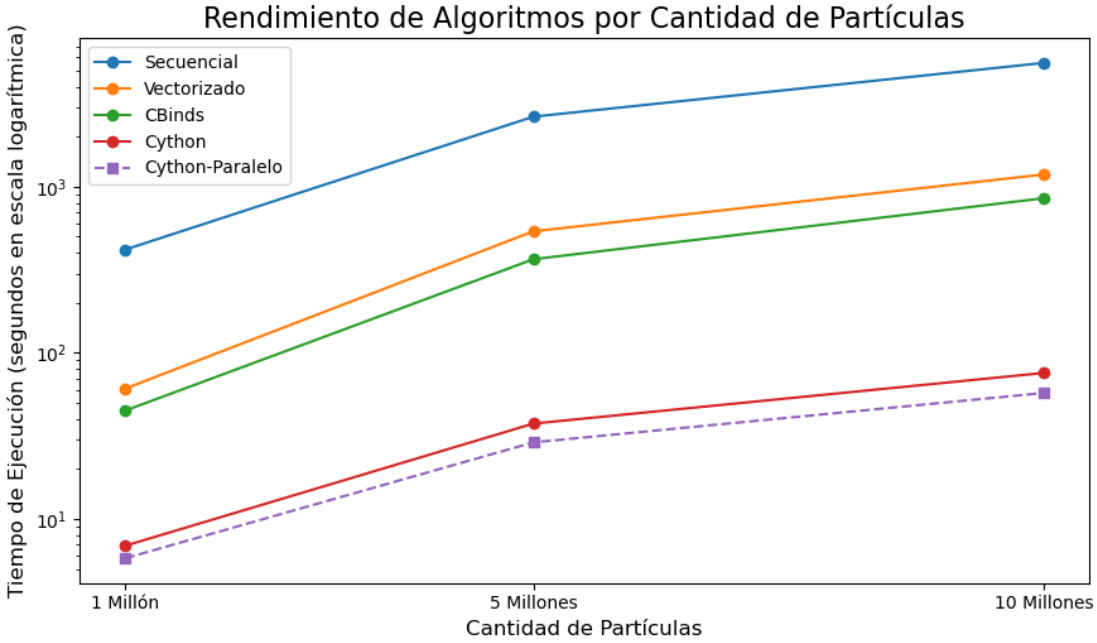

Fig. 1. Tiempo por cantidad de partículas para cada algoritmo

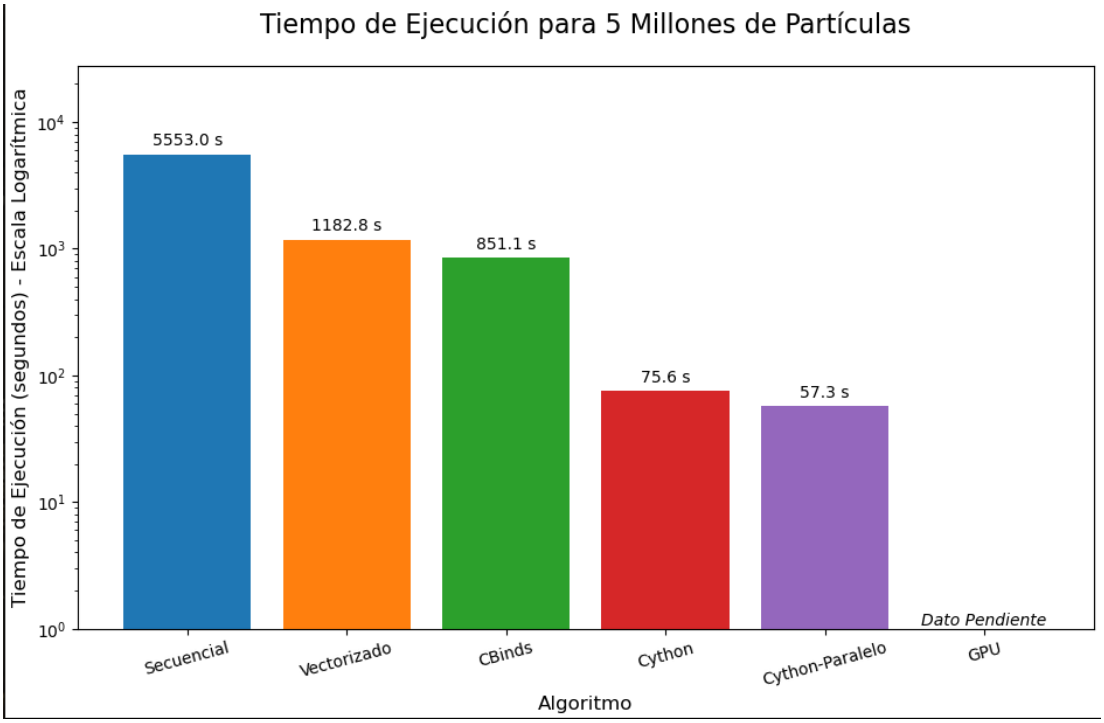

Fig. 2. Curva de rendimiento

Los resultados muestran una clara mejora en el rendimiento a medida que se optimiza el algoritmo de simulación de partículas. El enfoque secuencial básico es el menos eficiente, con tiempos de ejecución significativamente altos (5553 segundos para 10 millones de partículas). La vectorización y el uso de CBinds reducen considerablemente estos tiempos (1182.8 y 851.1 segundos, respectivamente), lo que demuestra la importancia de optimizar el código. Sin embargo, las implementaciones con Cython marcan un salto drástico en el rendimiento, especialmente la versión paralelizada (75.6 segundos para Cython y 57.3 segundos para Cython-Paralelo con 10 millones de partículas), destacando la ventaja de combinar compilación con paralelización. La escala logarítmica en los gráficos enfatiza las diferencias abismales entre los métodos, sugiriendo que, para

problemas de gran escala, las soluciones compiladas y paralelas son esenciales. La ausencia de datos para GPU deja abierta la posibilidad de una optimización aún mayor, dado el potencial de aceleración hardware que ofrecen las tarjetas gráficas. En conclusión, la elección del algoritmo impacta radicalmente en la eficiencia, siendo crítico adoptar técnicas avanzadas para manejar volúmenes elevados de datos.

Al observar los gráficos, se evidencia una mejora exponencial en el rendimiento a medida que aumenta la cantidad de partículas, especialmente en las versiones optimizadas con Cython y paralelización, lo que sugiere que la ventaja será aún más pronunciada en simulaciones a mayor escala.

VI. Conclusion

Los resultados demuestran claramente que las herramientas de High Performance Computing (HPC) —como vectorización, Cython y paralelización— permiten mejorar drásticamente el rendimiento de algoritmos intensivos en cómputo, especialmente al manejar grandes volúmenes de datos. La paralelización en particular destaca por su capacidad de reducir los tiempos de ejecución de forma casi lineal al aumentar los recursos disponibles (como núcleos de CPU). Sin embargo, la optimización óptima depende de:

- La naturaleza del problema: Algoritmos con operaciones independientes (como en simulaciones de partículas) se benefician más de la paralelización.

- La infraestructura: Combinar técnicas (ej: Cython + paralelización + GPU) puede lograr mejoras exponenciales, pero requiere adaptarse a los recursos disponibles (CPU multinúcleo, aceleradores GPU, etc.).

En conclusión, los resultados confirman que sí es posible optimizar significativamente un particle filter utilizando herramientas de HPC. La combinación estratégica de técnicas como vectorización, compilación con Cython y paralelización demostró reducir los tiempos de ejecución en órdenes de magnitud, especialmente al escalar a millones de partículas. Esto valida que, incluso para algoritmos complejos como los filtros de partículas, el uso adecuado de paradigmas de HPC permite lograr ganancias de rendimiento sustanciales, adaptándose a las capacidades del hardware disponible. La clave está en seleccionar e integrar las herramientas que mejor se ajusten a las características computacionales del problema y a la infraestructura específica.

References

[1] A. Doucet, S. Godsill, and C. Andrieu, “On sequential Monte Carlo sampling methods for Bayesian filtering,” Statistics and Computing, vol. 10, no. 3, pp. 197–208, Jul. 2000, doi: 10.1023/A:10089354100
[2] S. Thrun, W. Burgard, and D. Fox, Probabilistic Robotics, Cambridge, MA, USA: MIT Press, 2005.
[3] G. van Rossum and F. L. Drake, The Python Language Reference Manual, Python Software Foundation, 2009.
[4] S. Rahimi and F. Zhao, “Parallel particle filter tracking with OpenMP and CUDA,” Journal of Computational Science, vol. 28, pp. 336–343, Jan. 2018, doi: 10.1016/j.jocs.2018.01.008.
[5] M. Musial and M. Brzozowski, ”Particle Filter Application for UAV Tracking,” Sensors, vol. 21, no. 3, p. 824, 2021.
[6] D. Mandal and I. N. Kar, ”Target tracking using particle filter with sensor fusion for autonomous vehicles,” ISA Transactions, vol. 76, pp. 308–319, 2018.