Analisis de la Evolución Galáctica*

*Curso: Computación Paralela y Distribuida

1st Rodríguez Moscoso, Mauricio
Computer Science
201810642 100%
mauricio.rodriguez@utec.edu.pe

2nd Nicho Galagarza, Jorge
Computer Science
201810205 100%
jorge.nicho@utec.edu.pe

3rd Salazar Albán, Christian
Computer Science
201810716 100%
christian.salazar.a@utec.edu.pe

***Abstract*—Simular los sistemas estelares nos permiten entender el movimiento dinámico de los cuerpos existentes entro del universo. Para enfrentar este problema inicialmente sin buscaron fórmulas directas que nos permitan encontrar funciones que describan la trayectoria de diferentes cuerpos. Sin embargo, ante la imposibilidad de encontrar estas fórmulas, se plantearon modelos para poder simular billones de estrellas de la forma más realista posible, con resultados accesibles en un tiempo adecuado. Es por esto que en el presente trabajo se desarrollará un software de análisis de *performance* para evaluar una simulación de N-cuerpos realizada con hasta seis millones de cuerpos, con MPI-CUDA, y hardware GPU.**

***Index Terms*—N-cuerpos, *performance*, galaxias, MPI, CUDA, GPU**

I. Introducción

El problema de los N-cuerpos es, quizás, el problema sin resolver más antiguo y fructífero de la historia de la ciencia. Su origen se remonta a la necesidad del hombre de medir el paso del tiempo para anticiparse a la migración de animales y, posteriormente, a ciclos agrícolas. Los antiguos encontraron su primer calendario en el cielo. Para leerlo, los primeros astrónomos desarrollaron modelos empíricos basados en los patrones regulares que son seguidos por planetas y estrellas. Sin saberlo, estaban desarrollando modelos para resolver un problema de N-cuerpos: El Sistema Solar.

En el siglo XVII, Isaac Newton formuló la forma clásica de la interacción gravitacional. A excepción de la ley de la reflexión especular y el principio de Arquímedes. La solución del problema de los dos cuerpos, llevó a Newton a la invención del Cálculo. El problema de los tres cuerpos, sin embargo, resultó intratable, pues menciona que la mente humana no es capaz de considerar tantas causas de movimiento al mismo tiempo.

En los últimos 100 años, el estudio del problema de los N-cuerpos ha conducido al desarrollo de otra área de investigación: La dinámica estelar.

En el presente documento se diseña un PRAM para el problema de N-cuerpos y calcular su complejidad, posteriormente ejecutaremos pruebas con diferentes cantidades de cuerpos y procesadores para medir el tiempo de ejecución y compararlo con el teórico, calcular la fuerza, medir el tiempo de comunicación entre procesos, y los GFlops. Finalmente se realizan comparaciones de los resultados obtenidos del *speedup* y eficiencia para determinar la escalabilidad del algoritmo. Esto a partir de un software que nos permite obtener estas métricas de *performance*.

II. Marco teórico

A. El problema de los N cuerpos

En general, el problema de los N cuerpos consiste en predecir el movimiento de un grupo de N objetos que interactúan entre sí de forma independiente a gran distancia (por lo general, gravitatoria o electrostáticamente). Formalmente, para un grupo de N objetos en el espacio, si las posiciones iniciales (x_0) y las velocidades (v_0) se conocen en tiempo t_0, es posible predecir las posiciones (x) y velocidades (v) de los N objetos en un momento posterior t.

A primera vista, la solución a este problema está en $O(N^2)$ porque cada uno de los N cuerpos interactúa con N-1 cuerpos. Sin embargo, debido a simulaciones en paralelo, esto se puede reducir a $O(N^2/p)$, siendo p la cantidad de procesos. Por otro lado, existe el algoritmo de Barnes-Hut, que es de *divide-and-conquer*, y funciona dividiendo el dominio en cubos, para luego subdividirlos en sub-grupos, de manera que logra una complejidad de $O(Nlog(N))$

La resolución de este problema estuvo motivada originalmente por la necesidad de comprender el movimiento del sol, los planetas y las estrellas visibles, pero se ha aplicado a galaxias, planetas, fluidos y moléculas.

B. Ecuaciones

Se busca encontrar posiciones y movimientos de cuerpos en el espacio sometidos a fuerzas gravitatorias de otros cuerpos utilizando las leyes newtonianas del de Newton. La fuerza gravitatoria F entre dos cuerpos de masas m_a y m_b está dada por:

$$F_{ab} = \frac{Gm_a m_b}{r^2} \tag{1}$$

Donde G es la constante gravitatoria ($6,673 * 10^{-11} m^3 kg^{-1} s^{-2}$) y r la distancia entre los cuerpos, formada por la siguiente ecuación:

$$r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2} \tag{2}$$

Para un sistema de N partículas, la suma de las fuerzas es:

$$F = \sum_{i<j} F_{ij} = \sum_{i<j} \frac{Gm_i m_j}{r_{ij}^2} \tag{3}$$

Gracias a la Tercera Ley de Newton ("para cada acción hay una reacción reacción opuesta"), el número de interacciones se reduce a la mitad. Sin embargo, sigue siendo un problema complicado que sólo crece en complejidad con N.

C. Escalabilidad

El escalado es la forma en que cambia el rendimiento de una aplicación paralela a medida que aumenta el número de procesadores. Un marco ideal debería escalar linealmente.

1) Escalabilidad fuerte: El tamaño total del problema se mantiene igual a medida que aumenta el número de procesadores y el *speedup* aumenta. El escalado fuerte suele ser más útil y más difícil de conseguir que el escalado débil.

2) Escalabilidad débil: El tamaño del problema aumenta al mismo ritmo que el número de procesadores, manteniendo la misma cantidad de trabajo por procesador, y el *Runtime* permanece constante.

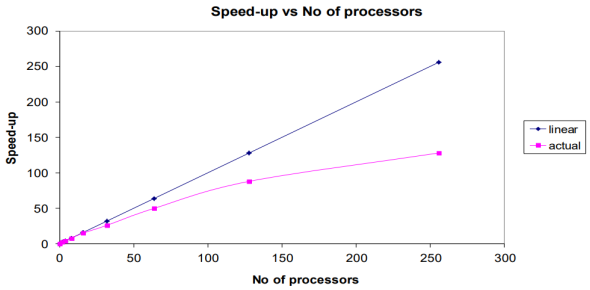

Fig. 1. Representación visual de la escalabilidad fuerte.

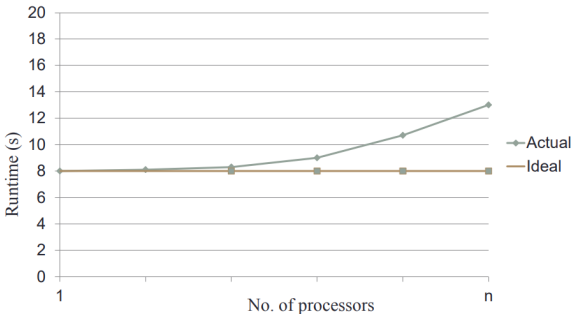

Fig. 2. Representación visual de la escalabilidad débil.

3) Speedup y Eficiencia: Las medidas de escalabilidad se dividen en dos grupos principales: *speedup* y eficiencia. El *speedup* S(N) se define como la relación entre el tiempo de cálculo secuencial T(1) y el tiempo de cálculo paralelo T(N) necesario para procesar una tarea con una carga de trabajo determinada cuando el algoritmo paralelo se ejecuta en N unidades de procesamiento paralelo (PUs) [1].

$$S(n) = \frac{T_1}{T_n} \tag{4}$$

Basándose en este *speedup* S(N), la eficiencia E(N) relaciona el aumento de velocidad con el número de PUs paralelos utilizados para lograr este *speedup*, y se define como:

$$E(n) = \frac{S_n}{n} = \frac{T_1}{p * T_n} \tag{5}$$

III. Trabajos relacionados

A. Astrophysical Particle Simulations with Large Custom GPU Clusters on Three Continents

En este trabajo se presentan simulaciones astrofíficas directas de N-cuerpos con hasta seis millones de cuerpos utilizando un código en paralelo

MPI-CUDA con grandes *clusters* en Pekín, Berkeley, y Heidelberg, usando diferentes tipos de hardware de GPU. Se logró alcanzar 1/3 del rendimiento máximo del código en un escenario de aplicación real con pasos temporales bloqueados jerárquicamente y una estructura de densidad *core-halo* del sistema estelar. El código y el hardware se utilizan para simular cúmulos estelares densos con muchas binarias y núcleos galácticos con agujeros negros supermasivos en los que pueden despreciarse las correlaciones entre partículas distantes [2].

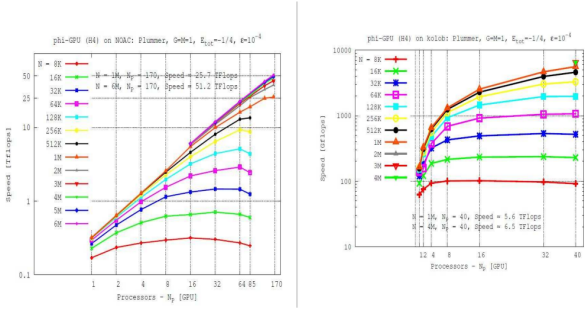

Fig. 3. Escalamiento fuerte para distintos tamaños de problema. Arriba: Cluster de GPUs del NAOC en Pekín; velocidad en Teraflop/s alcanzada en función del número de procesos, cada proceso con una GPU; se alcanzaron 51,2 Tflop/s sostenidos con 164 GPUs (3 nodos con 6 GPUs estaban caídos en el momento de la prueba). Abajo: Mismas simulaciones de referencia para el cluster Frontier kolob en ZITI Mannnheim, 6,5 Tflops/s alcanzados para cuatro millones de partículas en 40 GPUs. Cada línea corresponde a un tamaño de problema diferente (número de partículas), que se indica en la clave. Obsérvese que la curva lineal corresponde al escalado ideal.

La eficiencia global de paralelización de los códigos es muy buena. Como se puede visualizar en la Fig. 3, el aumento de la velocidad con simulaciones de mayor tamaño muestran un *strong scaling* casi ideal hasta su número máximo de 170 GPU's, sin que se observen indicios claros de *turnover* debido a la comunicación u otras latencias [2]. El *wall clock time* (T) necesitado se escala de la siguiente manera:

$$T = T_{host} + T_{GPU} + T_{comm} + T_{MPI} \qquad (6)$$

Donde T_{host} es el tiempo de computación empleado en el host, T_{GPU} es el de la GPU, T_{comm} es el de comunicación para enviar datos en el host y la GPU, y el T_{MPI} para el intercambio de datos MPI entre los nodos.

En esta implementación todos los componentes son bloqueantes, por lo que no se oculta la comunicación. El término dominante en la parte linealmente ascendente de las curvas de la fig. 3, es T_{GPU}, mientras que el cambio a plano está dominado por la comunicación MPI [2].

IV. Metodología

A. PRAM

Empezamos con el análisis teórico y tenemos en consideración que cada cuerpo (N_i) guarda su masa m_i, velocidad v_i, aceleracion (a_i) y posición (r_i) de manera local. A partir de esto podemos elaborar un PRAM basado en el algoritmo realizado por [2]:

Dentro de esta implementación [2], particularmente en el *main* del archivo *phi-GPU.cpp*, se observa la llamada a una función denominada *calc_force()*, en donde se calculan las fuerzas gravitacionales. Esta se encuentra implementada en *hermite4.h*. Nótese que N usa *Broadcast()*, previamente declarado en el *main* como *MPI_Bcast(nbody)* [2], para realizar la paralelización de cada cuerpo N_i en p procesos, porque todos son necesarios en todas partes para calcular las fuerzas. Mediante esta transmisión un proceso envía los mismos datos de su respectivo cuerpo N_i (incluyendo m_i, v_i, a_i, y r_i) a todos los procesos N_j.

Por otro lado, la comunicación entre procesos se produce cuando se hace la suma de fuerzas, ya que cada cuerpo requiere saber la fuerza gravitacional (F_i) de cada otro cuerpo, y en cada *timestep (t)*, por lo que se utiliza *Allreduce()*. Esto lo podemos observar dentro del *main* [2], luego de finalizar la llamada a la función *calc_force()*. De esta manera, los resultados reducidos de la fuerza gravitacional (F) se van a distribuir a todos los procesos.

Una vez que todos los cuerpos conocen la fuerza gravitacional (F), se procede a actualizar las velocidades y posiciones de cada uno de manera paralela con un *Broadcast()* a los N cuerpos. Esto se observa en [2] luego de realizar el *Allreduce()*, ya que dentro de una sentencia *For* se emplea el método *p.correct()* para cada cuerpo. Finalmente, todos estos procesos se juntan con un *Reduce()* [2], enviando el resultado reducido al proceso raíz. Por lo tanto, el PRAM se visualiza de la siguiente manera:

Algorithm 1: PRAM N-cuerpos

Data: Cuerpos (N_i), masa (m_i), aceleracion (a_i), posicion (r_i), velocidad (v_i).
Result: Fuerzas gravitatorias (F_i) y actualización de variables en N_i.

```
/* Broadcast a los cuerpos (N_i), con sus variables m_i, a_i, r_i, y v_i */
Broadcast(N);
while i < N do
    my_rx ← r_x[i];
    my_ry ← r_y[i];
    my_rz ← r_z[i];
    while j < N do
        if i ≠ j then
            /* Calcular Fuerza gravitatoria */
            dx ← r_x[j] − my_rx;
            dy ← r_y[j] − my_ry;
            dz ← r_z[j] − my_rz;
            a_x ← G * m[j]/(dx * dx);
            a_y ← G * m[j]/(dy * dy);
            a_z ← G * m[j]/(dz * dz);
        end
    end
    /* Asignar Fuerza gravitatoria */
    force[i].acc.x ← ax;
    force[i].acc.y ← ay;
    force[i].acc.z ← az;
end
Allreduce(F_i);
/* Actualizar velocidad y posicion */
Broadcast(N);
while i < N do
    /* Velocidad coordenada x, y, z */
    v_x[i] ← v_x[i] + a_x * dt;
    v_y[i] ← v_y[i] + a_y * dt;
    v_z[i] ← v_z[i] + a_z * dt;
    /* Posicion coordenada x, y, z */
    r_x[i] ← r_x[i] + v_x[i] * dt;
    r_y[i] ← r_y[i] + v_y[i] * dt;
    r_z[i] ← r_z[i] + v_z[i] * dt;
end
Reduce();
```

B. Performance del código

El código a analizar es el generador de datos iniciales de N-cuerpos (*gen-plum.c*), pues nos permite obtener el conjunto de datos para nuestro análisis en base a un parámetro N (cantidad de cuerpos). Luego utilizamos estos datos para ejecutar el archivo cpu-4th, que nos brinda métricas de *performance* como: speed, tiempo de comunicación, tiempo total de ejecución, tiempo en el cálculo de la fuerza, entre otros. Se utilizó Khipu, un cluster dedicado a la computación de alto desempeño, parte del Centro de Investigación para la Computación Sostenible (CICS) y el Departamento de Ciencia de la Computación (CS) de la Universidad de Ingeniería y Tecnología (UTEC). La infraestructura de este cluster cuenta con el procesador Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz de 20 *cores* por socket, y 40 por nodo. También contiene DRAM DDR4-1333 MHz con 128 GB por nodo, 480 de SSD con un total de 40 TB de HDD, y una Infiniband FDR MT4119 para la red. Finalmente para la GPU, contiene un NVIDIA Tesla T4 de 16 GB GDDR6, PCIe 3.0 x16 con 1 GPU por nodo.

A partir de las diversas ejecuciones con diferentes cantidades de cuerpos (N) y de procesadores (p) utilizados, se procede a realizar los siguientes pasos:

1) Guardar la información útil para el análisis en archivos .txt donde cada uno contiene resultados acorde a los parámetros utilizados (N, p).

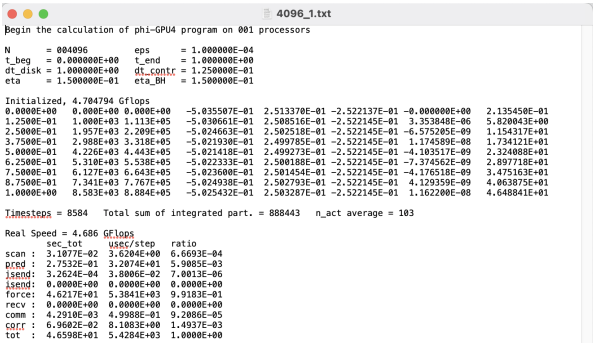

Fig. 4. Ejemplo de archivo utilizado para guardar la información de una ejecuión del problema. Existe un archivo por cada tamaño del problema (4k,8k,16k,32k,64k) y cada cantidad de procesos utilizados (1,2,4,8,16,32,64)

2) Por medio de de un script en python se generan nuevos archivos .txt que capturan la

información necesaria (Gflops, fuerza, total, y comunicación) dentro de los archivos .txt generales anteriores. Asimismo, también se generan archivos adicionales que incluyen información sobre speedups y eficiencia.

Fig. 5. Ejemplo de archivo generado. Se acumula la información por el tamaño del problema. Cada linea representa un tiempo, velocidad, etc según corresponda para una cantidad de procesos distinta

3) Mediante matplotlib, se grafican las variables necesarias (Gflops, fuerza, total, comunicación speedup y eficiencia) para nuestro análisis.

```
tamanos = sorted(datos.keys())
num_procesos = [1, 2, 4, 8, 16, 32, 64]

# Generar las curvas de la gráfica
colores = cm.get_cmap('viridis', len(tamanos))
formas = ['o', 's', 'D','P', 'H']
for i, tamano in enumerate(tamanos):
    datos_tamano = datos[tamano]
    curva = []
    for j, proceso in enumerate(num_procesos):
        curva.append(datos_tamano[j])

    # Crear la gráfica para el tamaño actual
    plt.plot(num_procesos, curva, '-'+formas[i%len(formas)], label=f"Tamaño {tamano}", color=colores(i))
    plt.scatter(num_procesos, curva, marker=formas[i%len(formas)], color=colores(i))

# Añadir la gráfica ideal si corresponde
if grafica_ideal:
    if valor_ideal is not None:
        plt.plot(num_procesos, [valor_ideal] * len(num_procesos), '--', color='red', label="Ideal")
    else:
        plt.plot(num_procesos, num_procesos, '--', color='red', label="Ideal")

# Etiquetas de los ejes y título
plt.xlabel("Cantidad de Procesos")
plt.ylabel(etiqueta_eje_y)
plt.title(titulo)

plt.legend()

plt.grid(True)

plt.show()
```

Fig. 6. Fragmento del script buildPlots.py utilizado para generar las gráficas.

4) Finalmente se implementa un software de análisis de *performance* en python que hace uso de los archivos previamente generados con el código fuente, python, y matplotlib para obtener métricas de *performance* que permitan concluir si el algoritmo es o no escalable.

V. Resultados

A partir de nuestro software se generan las siguientes gráficas:

A. Tiempo total de ejecución

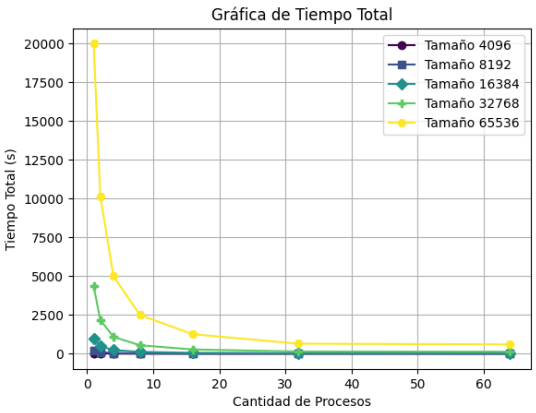

Fig. 7. Comparativa de los tiempos de ejecución para el problema N cuerpos. Cada curva corresponde a un tamaño distinto del problema. Se muestra la variación de tiempo (s) con respecto a la cantidad de procesos (p) los cuales fueron limitados hasta 64

Se aprecia que en todos los casos el tiempo total de ejecución disminuye al incrementarse la cantidad de procesos hasta llegar a un mínimo. Lo anterior se puede apreciar mejor para un tamaño de problema de 65,536 cuerpos, donde para un solo proceso le toma alrededor de 20,000 segundos y al incrementar la cantidad de procesos este tiempo es reducido a menos 2,000.

B. Tiempo de cálculo

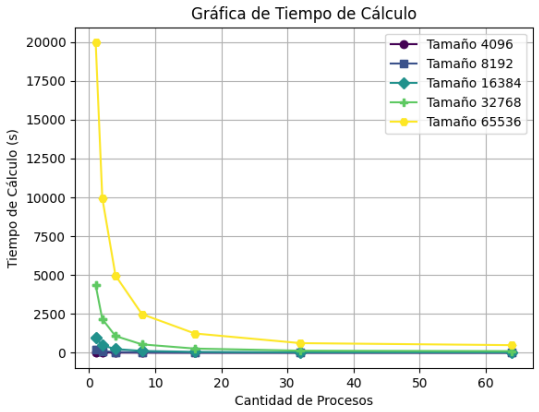

Fig. 8. Comparativa de los tiempos correspondientes al cálculo de fuerzas para el problema N cuerpos. Cada curva corresponde a un tamaño distinto del problema. Se muestra la variación de tiempo (s) con respecto a la cantidad de procesos (p) los cuales fueron limitados hasta 64

Como se observó en el código PRAM se realizan muchas operaciones de cálculo al resolver el problema N - body. Por ello, el tiempo de cálculo de fuerzas obtiene curvas similares a las vistas en la figura 7.

C. Tiempo de comunicación

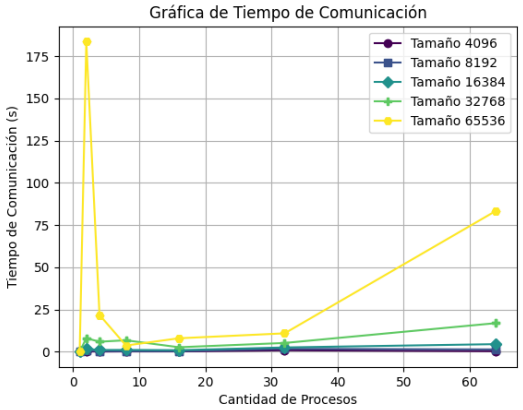

Fig. 9. Comparativa de los tiempos correspondientes a comunicación entre procesos para el problema N cuerpos. Cada curva corresponde a un tamaño distinto del problema. Se muestra la variación de tiempo (s) con respecto a la cantidad de procesos (p) los cuales fueron limitados hasta 64

Como se observó en el PRAM, hay menos operaciones de comunicación por lo que los tiempos correspondientes son mucho menores a los de las operaciones de cálculos. En nuestra experimentación hemos observado como al pasar de no haber comunicación (1 proceso) a que esta se realice (2 procesos) hay un incremento importante en el tiempo. Asimismo, hemos notado un patrón de una inicial disminución de tiempo de comunicación, que va al pasar de 2 a 4 procesos que se comunican, a un incremento en estos tiempos al pasar de 4 procesos a más. Lo anterior mencionado se puede ver mucho más reflejado en la curva amarilla, que muestra los experimentos para un tamaño del problema de 65,536 cuerpos.

D. Velocidad en GFlops

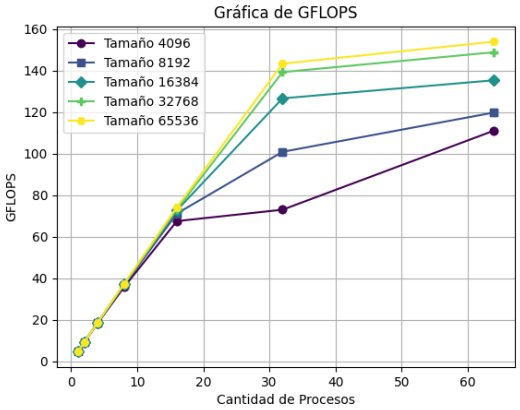

Fig. 10. Comparativa de operaciones flotantes por segundo ($10\hat{9}$/s) para el problema N cuerpos. Cada curva corresponde a un tamaño distinto del problema. Se muestra la variación de operaciones con respecto a la cantidad de procesos (p) los cuales fueron limitados hasta 64

La cantidad de Gflop/s aumenta en todos los casos al aumentar la cantidad de procesos. Asimismo, mientras mayor sea el tamaño del problema es normal que se deban realizar mayor cantidad de operaciones y por ende aumenten los Gflop/s. Si bien esta medida tiende a aumentar casi de forma lineal, luego de cierto punto esta sigue aumentado pero en menor medida. Ello se debe a que hay un límite de paralelismo efectivo que pueden lograr los algoritmos, es decir, que no se puede distribuir de manera efectiva el trabajo entre los procesos o hay mucho tiempo de comunicación de por medio, lo que limita el incremento en Gflops.

E. Speedup

El speedup (tiempo secuencial/tiempo paralelo) incrementa en todos los casos, siendo que mientras

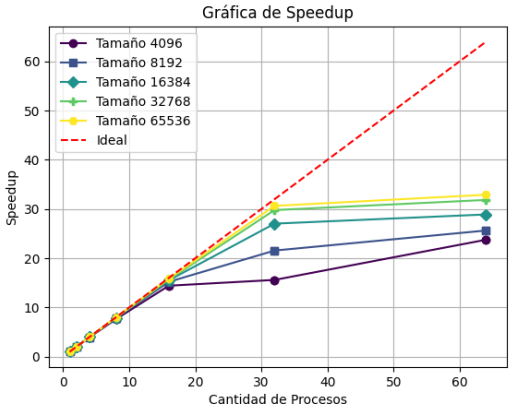

Fig. 11. Comparativa de Speedup para el problema N cuerpos. Cada curva corresponde a un tamaño distinto del problema. La cantidad de procesos (p)fueron limitados hasta 64

mayor sea el tamaño del problema es más cercano al speedup ideal. En todos los casos también se observa que hay una cercanía a la recta ideal hasta determinada cantidad de procesos que se utilicen. Esta disminución del speedup, al igual que con los gflops, se puede dar principalmente debido a una inefectiva distribución del trabajo así como por el incremento de los tiempos de comunicación.

F. Eficiencia

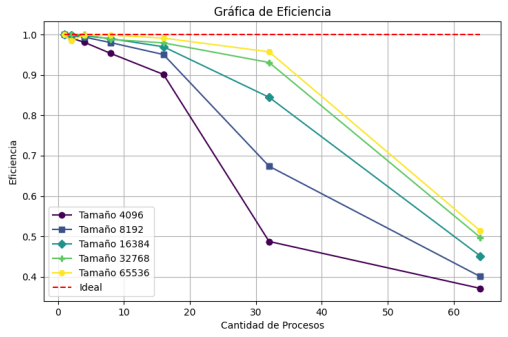

Fig. 12. Comparativa de eficiencia para el problema N cuerpos. Cada curva corresponde a un tamaño distinto del problema. Se muestra la variación de eficiencia con respecto a la cantidad de procesos (p) los cuales fueron limitados hasta 64

Los resultados de eficiencia son un reflejo de lo visto en la figura 11. Todas las curvas mantienen inicialmente hay una cercanía inicial a la recta ideal que va a alejándose al aumentar la cantidad de procesos. Asimismo, mientras mayor sea el tamaño del problema, mayor será la eficiencia para este problema. La eficiencia es una muestra de los problemas de distribución del trabajo mencionados en los puntos anteriores, siendo que mientras mayor sea la cantidad de procesos menor es el trabajo efectivo que realiza cada uno de ellos.

VI. Optimización

Se desarrolló una interfaz gráfica en Python utilizando las librerías de TKinter y Matplotlib. El usuario puede escoger un N determinado y visualizar cómo cambia la eficiencia por cantidad de procesos.

A. Escalabilidad a determinado N

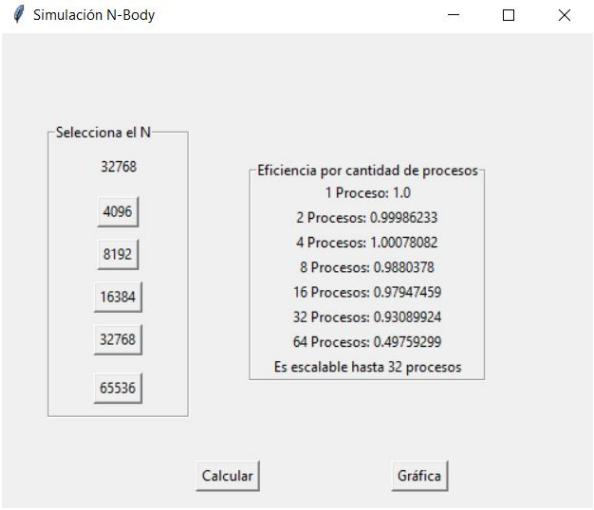

Fig. 13. Interfaz gráfica desarrollada con TKinter para mostrar al usuario interactivamente las eficiencias por tamaño de problema (N) y determinar el número máximo de procesos a los que es escalable

La forma en la que se determina hasta qué número de procesos(P), puede seguir siendo escalable un determinado tamaño de problema(N), es obteniendo la eficiencia con estas variables la cual fue mostrada en la sección de resultados. Esta debe ser mayor o igual a 0.7, teniendo 1.0 como la eficiencia ideal.

B. Gráfica de la eficiencia a determinado N

Una vez seleccionado el tamaño del problema deseado, se puede visualizar la gráfica tomando en cuenta un único N, para corroborar que la información proporcionada por la opción ”Calcular” de la interfaz es correcta.

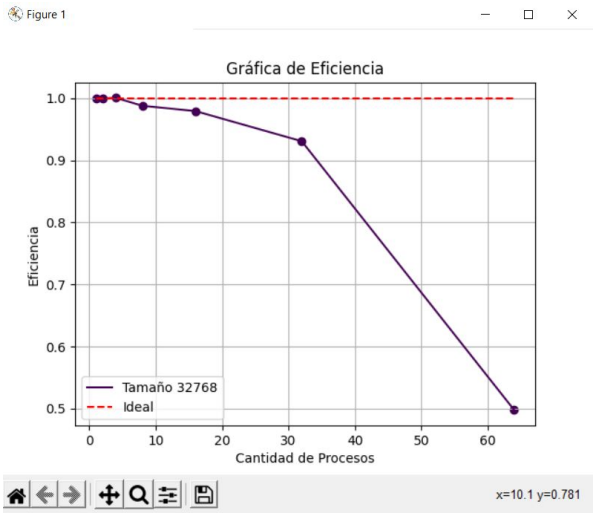

Fig. 14. Gráfica de eficiencia generada por la interfaz gráfica para visualizar cómo se comporta la eficiencia con un N en específico.

VII. Conclusiones

- La implementación del problema [2] es más eficiente que otras, puesto que el tiempo de comunicación es mínimamente significativo, incluso al utilizar 64 procesos. Ello porque se prioriza que todos los procesos tengan la información necesaria de forma inicial para que se reduzcan los tiempos de comunicación.
- Si bien esta implementación [2] es más eficiente respecto al tiempo de comunicación. Consideramos que al aumentar la cantidad de procesos (ej. 128) seguirá disminuyendo, aunque de forma mínima, el tiempo de cálculo. También, se incrementaría el tiempo de comunicación de forma cada vez más significativa y llegaría a afectar bastante el tiempo total de ejecución. Por ende, idealmente deberían utilizarse entre 32 - 64 procesadores según la cantidad de recursos que tengamos disponibles.
- Parece ser que el tamaño ideal del problema en base a nuestros experimentos es de 32,768 cuerpos. Si bien al duplicar el tamaño del problema se mejoran muchas medidas (speedup, eficiencia, Gflops), no lo hacen de forma significativa. Además hay otro grupo de medidas que empeoran (tiempo de comunicación, tiempo total).
- El código es escalable en la mayor cantidad de casos evaluados a excepción en los que la cantidad de procesos es 64, teniendo como patrón una eficiencia de entre 0.37 y 0.52, la cual no nos permite considerarla como escalable. Luego tenemos los casos en los que N es relativamente pequeño (4096-8192) y la cantidad de procesos es 32, en el cual la eficiencia es también menor a 0.7, que es la cantidad que usamos como límite para determinar que una prueba es escalable.
- Se puede determinar que el algoritmo de N-cuerpos que utiliza MPI [2], sigue una complejidad $O(\frac{n^2}{p})$, mostrando una fuerte similitud con el análisis teórico a partir del PRAM.

VIII. Código

Puede encontrar el código base de N - Body así como el resto de scripts utilizados para este informe en el proyecto: Evolución Galáctica - Computación Paralela

References

[1] Schryen, Guido. (2022). Speedup and efficiency of computational parallelization: A unifying approach and asymptotic analysis. 10.48550/arXiv.2212.11223.

[2] Spurzem, Rainer & Berczik, Peter & Berentzen, I. & Nitadori, Keigo & Hamada, Tsuyoshi & Marcus, Guillermo & Kugel, Andreas & Männer, R. & Fiestas, J. & Banerjee, R. & Klessen, R.. (2011). Astrophysical particle simulations with large custom GPU clusters on three continents. Computer Science Research and Development. 26. 145-151. 10.1007/s00450-011 0173-1.