Paralelizacion en APSP - All Pair Shortest Path

Guzman Pieroni, Gianfranco Stefanoli.

Facultad de Ingeniería, Universidad de la República
emails: guzman.pieroni@fing.edu.uy, gianfranco.stefanoli@fing.edu.uy.

***Resumen*— En este informe se aborda el desafío de determinar el peso mínimo entre dos nodos en grafos grandes utilizando técnicas de Computación de Alta Performance (HPC). El foco está en resolver el problema de encontrar las rutas más cortas entre todos los pares de nodos (APSP) mediante la paralelización del algoritmo de Floyd-Warshall con la ayuda de múltiples procesadores.**

I. DESCRIPCIÓN DEL PROBLEMA

El problema planteado se basa en encontrar el peso mínimo entre dos nodos cualesquiera de un grafo. El algoritmo de Floyd-Warshall es una técnica fundamental en la teoría de grafos que se utiliza para encontrar las rutas más cortas entre todos los pares de nodos en un grafo ponderado. Aunque es eficiente y versátil, su complejidad temporal de O(n^3) puede ser un obstáculo significativo cuando se trabaja con grafos de gran tamaño.

II. JUSTIFICACIÓN DEL USO DE HPC

En el contexto de nuestro estudio, el uso del procesamiento en paralelo se convierte en una herramienta esencial para manejar eficientemente estas tareas.

Complejidad Computacional: El algoritmo de Floyd-Warshall tiene una complejidad de tiempo de O(n^3), donde n es el número de vértices en el grafo. Para grafos grandes, esta complejidad puede llevar a tiempos de ejecución demasiado grandes en sistemas de un solo procesador o en sistemas de baja capacidad de cómputo, haciendo que HPC sea una opción adecuada para reducir el tiempo de ejecución.

Paralelización Eficiente: Se ofrece la posibilidad de paralelizar de manera efectiva el algoritmo de Floyd-Warshall, distribuyendo el cálculo entre múltiples procesadores. Esto permite que partes independientes del algoritmo se ejecuten simultáneamente, reduciendo significativamente el tiempo total requerido.

Manejo de Grandes Grafos: Los grafos extensos, que son comunes en aplicaciones del mundo real como redes de transporte, redes sociales, entre otros, pueden ser manejados más eficientemente en sistemas basados en computación de alta performance, debido a su capacidad de memoria y almacenamiento superior.

Optimización de Recursos: HPC permite utilizar recursos computacionales optimizados, como clusters de procesadores optimizados para cálculos matemáticos, que pueden acelerar aún más el algoritmo de Floyd-Warshall.

Resultados en Tiempo Real: Algunas aplicaciones pueden requerir resultados de APSP en tiempo real o en plazos muy cortos. Con la computación de alta performance, las soluciones pueden ser generadas en un marco de tiempo que satisfaga estas demandas, lo que sería inviable con hardware de menor rendimiento.

Escalabilidad: A medida que crece el tamaño del grafo o las demandas de rendimiento, los sistemas HPC ofrecen una escalabilidad que permite a los usuarios incrementar fácilmente la capacidad de cómputo para abordar problemas más grandes o más complejos sin necesidad de rediseñar la solución desde cero.

III. ESTRATEGIAS DE RESOLUCIÓN

Se indican debajo las estrategias a utilizar durante el proyecto para poder obtener y comparar tanto los resultados como las diferentes métricas brindadas.

Se aclara que para las estrategias usadas se opta por las matrices de adyacencia como representación de los grafos y distancias. El lenguaje de programación utilizado para implementar los algoritmos fue C. Por otro lado se utilizó Python para realizar la búsqueda de los nodos disponibles en el cluster de la FING para ser usados mediante SSH.

A. Procesamiento Serial

Este procesamiento ejecuta el algoritmo de Floyd-Warshall con una complejidad de O(n^3) utilizando un único hilo de ejecución.

Se agrega el pseudocódigo del algoritmo:

```
Algorithm Floyd-Warshall (A, n):
  for k ← 1 to n do
    for i ← 1 to n do
      for j ← 1 to n do
        A[i][j] ← min(A[i][j], A[i][k] + A[k][j])
```

Figura 1: Algoritmo Serial

Explicación de los Bucles:

Bucle Externo (k): Este bucle recorre todos los nodos del grafo y considera cada nodo como un nodo intermedio en el camino entre dos otros nodos. La variable k representa el índice del nodo intermedio.

A medida que k avanza, el algoritmo explora si pasar por el nodo k puede mejorar los caminos más cortos conocidos entre todos los pares de nodos i y j.

Bucle Medio (i): Este bucle recorre todos los nodos del grafo para utilizarlos como posibles nodos de origen de un camino.La variable i representa el índice del nodo de origen. Para cada valor de i, el algoritmo examina si existen caminos más cortos desde el nodo i hasta todos los demás nodos j del grafo, pasando por el nodo intermedio k.

Bucle Interno (j): Este bucle recorre todos los nodos del grafo para utilizarlos como posibles nodos de destino de un camino.La variable j representa el índice del nodo de destino. Bajo la consideración del nodo i como origen y k como posible intermedio, se evalúa si el camino desde i a j pasando por k es más corto que el camino más corto conocido previamente.

B. *Paralelismo con OpenMP*

OpenMP es una API diseñada para la programación paralela en sistemas multiprocesador. La versión paralela del algoritmo de Floyd-Warshall se encargará de distribuir el trabajo entre múltiples hilos, permitiendo que cada hilo procese una sección del grafo (representado como matriz de adyacencia) de manera simultánea. Esto aprovecha los recursos de los múltiples núcleos disponibles para mejorar el rendimiento.

Algorithm Parallel_Floyd_Warshall (A, n):
for $k \leftarrow 1$ to n do
  #pragma omp parallel for schedule(static) collapse(2)
  for $i \leftarrow 1$ to n do
    for $j \leftarrow 1$ to n do
      $A[i][j] \leftarrow \min(A[i][j], A[i][k] + A[k][j])$

Figura 2: Algoritmo Paralelo

La paralelización del algoritmo se logra utilizando la directiva *#pragma omp parallel for*, acompañada de los atributos schedule(static) y collapse(2) para los bucles internos (i y j). Esto permite que múltiples hilos de ejecución trabajen en diferentes partes de la matriz simultáneamente, reduciendo el tiempo de ejecución del algoritmo al aprovechar los núcleos múltiples disponibles en el sistema.

La directiva collapse(2) combina dos niveles de bucles anidados en uno solo. Esto significa que los bucles sobre los índices i y j se tratan como un único bucle, facilitando la distribución del trabajo entre los hilos y mejorando la carga de trabajo balanceada.

El atributo schedule(static) divide las iteraciones de los bucles en bloques de tamaño fijo y asigna cada bloque a un hilo de ejecución. Este enfoque minimiza la sobrecarga de la planificación en tiempo de ejecución y asegura una distribución equitativa de las iteraciones entre los hilos, lo que es especialmente útil cuando todas las iteraciones tienen aproximadamente el mismo tiempo de ejecución.

C. *Paralelismo con MPI*

Antes de hablar de la implementación de este caso, se aclara que se utilizó un script de Python para encontrar hosts accesibles en un rango específico y agregar sus claves públicas al archivo "*known_hosts"* dentro de la carpeta .ssh. En primer lugar el mismo obtiene una lista de hosts y para cada uno se intenta agregar su clave pública al archivo "*known_hosts*". Después, se abre un archivo llamado hosts.txt y si se verifica la accesibilidad a un host específico, el script escribe su nombre allí.

Para mejorar el paralelismo proporcionado por OpenMP, el cual en el clúster de la Facultad de Ingeniería (FING) estaba limitado a cuatro procesadores lógicos de cada máquina individualmente, inicialmente se decidió dividir la matriz de adyacencia del grafo en submatrices más pequeñas y hacer uso de MPI para enviar estas a diferentes nodos. Sin embargo, esta estrategia presentó dificultades debido a las precondiciones que deben cumplirse con los bloques (tamaño n/sqrt(cant_procesos)).

Para superar esta limitación, se optó por dividir la matriz de adyacencia en filas y distribuirlas entre los diferentes procesos. Esta nueva estrategia permite mantener el modelo maestro-esclavo y romper la barrera de los 4 hilos simultáneos que teníamos con OpenMP, ya que cada proceso puede gestionar múltiples filas de manera más eficiente. De esta forma, se aprovechan mejor los recursos disponibles y se mejora la cooperación entre procesos, superando las restricciones impuestas por el hardware.

El proceso maestro distribuye fragmentos de la matriz de adyacencia a los procesos esclavos. Después de terminar los cálculos locales, los procesos esclavos envían sus resultados actualizados de vuelta al proceso maestro. El mismo recoge estos resultados parciales y actualiza la matriz de adyacencia completa. Se añade un pseudocódigo debajo.

```
Algorithm MPI_Distributed_Floyd_Warshall_Update (A, n, num_processes)

Initialization
  start ← rank * step
  end ← (rank == num_processes - 1) ? n : start + step

For each vertex k do
  responsible ← (k >= start and k < end) ? rank : -1
  MPI_Allreduce(responsible, MPI_MAX)
  If rank == responsible do
    row_k ← A[k]
    MPI_Bcast(row_k)
  For each row i in [start, end) do
    For each column j in [0, n) do
      If A[i][k] != INF and row_k[j] != INF do
        A[i][j] ← min(A[i][j], A[i][k] + row_k[j])

Gather results
  MPI_Gather(A[start], complete_matrix)
```

Figura 3: Algoritmo Distribuido

IV. PROPUESTA DE EVALUACIÓN EXPERIMENTAL

A. Análisis del Algoritmo Serial

Se realizará el análisis del comportamiento y el tiempo de ejecución del mejor algoritmo serial conocido (t1) para la tarea específica.

B. Análisis de Paralelismo con OpenMP

La propuesta de evaluación experimental busca aumentar la eficiencia del procesamiento de datos mediante la paralelización con OpenMP. A continuación se presentan los métodos para medir y analizar el rendimiento del algoritmo paralelo.

- Speedup Algorítmico (Sn):

Utilizar S1 como el tiempo de ejecución del algoritmo serial en un solo procesador.
Después ejecutar el algoritmo paralelo con 'n' procesadores de una sola máquina, calculando la matriz de adyacencia entera, y obtener el tiempo de ejecución Tn.
Luego calcular el speedup algorítmico como Sn = S1 / Tn.
Se busca lograr un speedup lineal en la misma máquina.

- Eficiencia Computacional (En):

Evaluar la eficiencia computacional con En = Sn / N, donde N es el número de procesadores.
La eficiencia ideal se aproxima a 1.

C. Distribución con MPI

La evaluación experimental se enfoca en medir el rendimiento del algoritmo utilizando únicamente MPI, permitiendo el uso de más núcleos y la distribución del trabajo entre múltiples nodos en un clúster. Los objetivos incluyen medir el Speedup Algorítmico, Eficiencia Computacional, Evaluación de la Paralelización, Escalabilidad y el Overhead de Comunicación.

- Speedup Algorítmico:

Usar T1 como el tiempo de ejecución del algoritmo serial en un solo procesador. Calcular el speedup algorítmico utilizando MPI comparando el tiempo de ejecución con N nodos. Se busca obtener un speedup cercano al ideal.

- Eficiencia Computacional:

Evaluar la eficiencia computacional dividiendo el speedup algorítmico por el número total de procesadores (N). La eficiencia ideal se aproxima a 1 y se buscará mantenerla alta a medida que se escalan los recursos.

- Escalabilidad:

Evaluar si el algoritmo mantiene una alta eficiencia al incrementar el número de nodos.

- Overhead de Comunicación:

Analizar el tiempo necesario para enviar y recibir trozos de la matriz entre nodos, la sincronización de los procesos y el desbalance de carga. Medir el impacto del overhead en el rendimiento.

D. Paralelización con MPI + OpenMP (Modelo Híbrido)

La evaluación experimental también se enfoca en medir el rendimiento del algoritmo utilizando un enfoque híbrido de MPI y OpenMP, que permite usar más núcleos y distribuir el trabajo entre múltiples nodos en un clúster.

Se evaluó una posible implementación de este caso. Se probó tanto a nivel local como con las máquinas de la FING pero no se presentaron mejoras. Se considera que el overhead de las máquinas puede ser el gran responsable en ese caso. Los tiempos de ejecución que brindaba dicho programa (sobre todo con matrices de gran tamaño) dieron indicios de que el modelo híbrido es un tanto más complejo que los demás,teniendo en cuenta la compenetración de OpenMP y MPI, y no se creyó de valor agregarlo, ya que los resultados no eran de aporte debido a que eran inferiores a la versión secuencial

V. ANALISIS EXPERIMENTAL

Evaluar el rendimiento de algoritmos de procesamiento de grafos, como el de Floyd-Warshall, necesita pruebas detalladas que reflejan escenarios complejos. Usar matrices de distintos tamaños, especialmente las grandes, es clave para obtener resultados precisos y significativos.

Las matrices pequeñas no siempre reflejan la complejidad y los desafíos del mundo real, donde los grafos suelen ser grandes y densos. Con matrices grandes, se pueden simular

mejor estas situaciones y ver cómo el algoritmo se comporta bajo cargas intensas.

Además, probar el rendimiento con matrices grandes permite analizar la escalabilidad del algoritmo. La escalabilidad es fundamental en algoritmos paralelos porque muestra cómo el algoritmo maneja el aumento del tamaño del problema sin perder mucho rendimiento. Usando matrices grandes, se pueden detectar y solucionar problemas de escalabilidad, optimizando así el algoritmo para su uso en aplicaciones prácticas. El uso de estas matrices también ayuda a medir el speedup y la eficiencia. La diferencia en los tiempos de ejecución entre la versión secuencial y las versiones paralelas del algoritmo es más evidente con problemas más grandes, proporcionando datos robustos para evaluar la efectividad de las técnicas de paralelización. Esto permite una interpretación más clara del rendimiento y la eficiencia del algoritmo en un entorno paralelo.

Como pequeña prueba para explicar la decisión tomada, se realizaron ejecuciones con un aumento gradual de tamaños, pasando de matrices pequeñas a otras con un tamaño considerablemente grande para computar (32x32, 128x128 y 4096x4096), ejecutando el algoritmo en las versiones secuencial y paralela (con 1 a 4 hilos).

Los resultados evidencian cómo los tiempos de ejecución varían según el tamaño de la matriz y el número de hilos utilizados:

	Tamaño Matriz		
Cant. Hilos	32x32	128x128	4096x4096
1 (Sec)	0.0001	0.0056	178.02
1	0.0001	0.0056	179.27
2	0.0002	0.0031	87.88
3	0.0002	0.0022	60.14
4	0.0002	0.0017	45.10

Tabla 1: Diferencia entre matrices

Estos datos demuestran la importancia de utilizar matrices grandes para evaluar la eficiencia y escalabilidad del algoritmo en situaciones que simulan aplicaciones del mundo real, ya que las de tamaño relativamente pequeño no nos brindan mejora alguna.

A partir de los resultados y comentarios anteriores, se tomó una matriz lo suficientemente grande para realizar los análisis experimentales propuestos anteriormente en el documento. Siendo esta de 8000 nodos (8000x8000)

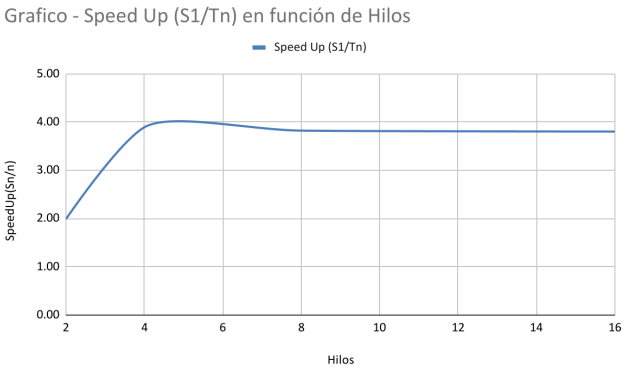
El tiempo de ejecución para S1 y T1, es decir, la ejecución secuencial del algoritmo y la ejecución del algoritmo paralelo en un solo hilo, fue:

- S1 = 1305.97 segundos
- T1 = 1305.68 segundos

En base a esos tiempos se realizaron los cálculos de Speedup y eficiencia computacional. Se observa que la variación entre ambos tiempos es mínima.

Speed Up - Open MP (8000x8000)		
Hilos	Tiempo (s)	Speed Up (S1/Tn)
2	656.42	1.99
4	335.45	3.89
8	341.58	3.82
16	343.26	3.80

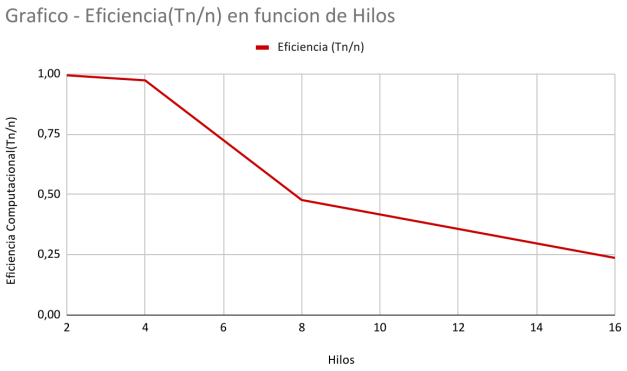
Tabla 2: Speedup en algoritmo paralelo



Gráfica 1: Speedup en función de hilos de algoritmo paralelo

Eficiencia Computacional - Open MP (8000x8000)		
Hilos	Speed Up (S1/Tn)	Eficiencia (Tn/n)
2	1.99	0.99
4	3.89	0.97
8	3.82	0.48
16	3.80	0.24

Tabla 3: Eficiencia computacional en algoritmo paralelo



Gráfica 2: Eficiencia computacional en función de hilos de algoritmo paralelo

Se observa que al aumentar el número de hilos de 2 a 4, hay una mejora evidente en el Speed Up y en la eficiencia. Sin embargo, a partir de 8 hilos, la mejora en el Speed Up no es tan significativa y la eficiencia disminuye considerablemente.

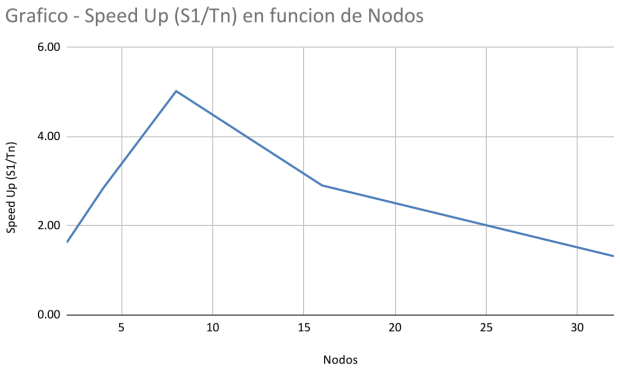
Esto se debe a que las máquinas de la Facultad de Ingeniería solo tienen 4 procesadores lógicos. Ya se había previsto que utilizar 8 y 16 hilos resultaría en cambios de contexto innecesarios, provocando una sobrecarga que no mejoraría el rendimiento.

Esa limitación se pudo sobrepasar usando una variación del algoritmo adaptado para MPI.

Se adjuntan debajo las tablas y gráficos vinculados a los resultados del SpeedUp y Eficiencia Computacional, calculados a partir del algoritmo de MPI en base a la variación de nodos.

Speed Up - MPI (8000x8000)		
Nodos	Tiempo (s)	Speed Up (S1/Tn)
2	803.20	1.63
4	458.65	2.85
8	260.03	5.02
16	449.11	2.91
32	985.90	1.32

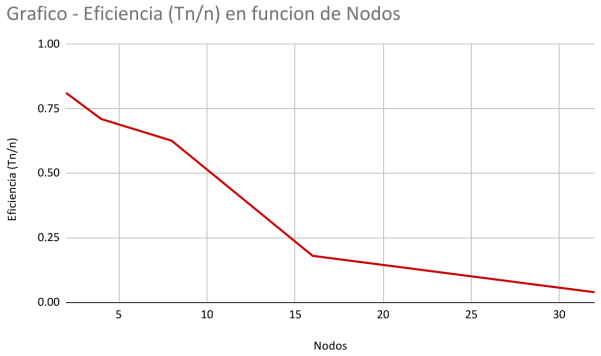
Tabla 4: Speedup en algoritmo distribuido



Gráfica 3: Speedup en función de nodos de algoritmo distribuido

Eficiencia Computacional - MPI (8000x8000)		
Nodos	Speed Up (S1/Tn)	Eficiencia (Tn/n)
2	1.63	0.81
4	2.85	0.71
8	5.02	0.63
16	2.91	0.18
32	1.32	0.04

Tabla 5: Eficiencia computacional en función de nodos de algoritmo distribuido



Gráfica 4: Eficiencia computacional en función de nodos de algoritmo distribuido

El análisis del rendimiento del algoritmo paralelo usando MPI, comparado con OpenMP, muestra información importante sobre la escalabilidad y la comunicación entre nodos. Usar MPI con más de 4 procesadores lógicos mejora bastante el rendimiento, alcanzando un speedup de 5 al utilizar 8 nodos simultáneamente. Sin embargo, cuando se usan más de 8 nodos, el rendimiento cae bastante. Por ejemplo, con 16 nodos el speedup baja a 2.91, y con 32 nodos baja aún más a 1.32. Esto sugiere que, aunque dividir la carga de trabajo entre varias máquinas ayuda, el costo de comunicación entre ellas puede ser mayor que los beneficios de la paralelización a partir de cierto punto.

El decremento en el rendimiento con más de 8 nodos se debe principalmente al costo de comunicación. La latencia y el tiempo dedicado a la sincronización entre nodos aumentan con el número de nodos. Con 16 nodos, la eficiencia cae a 18%, indicando que gran parte del tiempo se consume en comunicación en lugar de cálculo útil. Con 32 nodos, la eficiencia se reduce aún más, sugiriendo que casi todo el tiempo se dedica a la comunicación.

Además, el tamaño de las matrices está limitado por la memoria disponible en el nodo que se encarga de distribuirlas, lo que impide usar matrices más grandes que podrían compensar el tiempo adicional de comunicación en relación a un algoritmo secuencial.

En resumen, el algoritmo Floyd-Warshall implementado con MPI puede mejorar el rendimiento en configuraciones con un número moderado de nodos, pero su escalabilidad se ve limitada por el costo de la comunicación y las restricciones de memoria del nodo encargado de distribuir las matrices en el cluster de la FING. Es necesario un balance cuidadoso entre el número de nodos y el tamaño de las matrices para optimizar el rendimiento, ya que usar muchos nodos solo es ventajoso hasta cierto límite, más allá del cual el costo de comunicación supera los beneficios de la paralelización

VI. POSIBLES MEJORAS

- Una posible mejora para el proyecto es reintentar el enfoque híbrido de OpenMP y MPI, el cual no mostró mejoras significativas en los tiempos de ejecución en los intentos iniciales. Este enfoque combina las ventajas de la paralelización en memoria compartida de OpenMP con la distribución de tareas en memoria distribuida de MPI, y podría ser clave para aprovechar mejor los recursos del clúster. En este enfoque híbrido, MPI se utiliza para distribuir las submatrices o partes del problema a diferentes nodos del clúster. Luego, dentro de cada nodo, OpenMP se emplea para paralelizar las tareas entre los núcleos disponibles, permitiendo un uso eficiente de los recursos de cada nodo. Durante los primeros intentos, es posible que no se haya logrado un rendimiento óptimo debido a varios factores, como una carga de trabajo desequilibrada entre nodos, una sincronización ineficiente o una configuración subóptima de los parámetros de OpenMP y MPI. Reintentar este enfoque con una optimización cuidadosa podría llevar a mejoras significativas en el rendimiento.

- Otra posible mejora para el rendimiento del proyecto es la paralelización del algoritmo de Dijkstra utilizando MPI. Esta técnica podría reducir el tiempo de comunicación y aumentar la eficiencia del algoritmo distribuyendo el trabajo entre los nodos del clúster. El algoritmo de Dijkstra se podría paralelizar dividiendo el grafo en subgrafos y asignando cada uno a un nodo diferente. Esta división permite que múltiples nodos trabajen en paralelo, reduciendo significativamente el tiempo de ejecución. Cada nodo inicializa sus propias estructuras de datos, calcula las distancias mínimas en cada iteración y luego intercambian esta información con los otros nodos mediante operaciones colectivas de MPI como “MPI_Allreduce”. Este proceso de comunicación y actualización se repite hasta que todas las distancias mínimas se han calculado y los nodos se han visitado.
- Como última opción, se podría intentar reducir el overhead en el algoritmo MPI actual para mejorar su rendimiento, sin cambiar el algoritmo de Floyd-Warshall en sí. Ajustar los parámetros de comunicación de MPI podría tener un impacto en el rendimiento. Se ha descubierto que configurar correctamente el tamaño del buffer y las políticas de enrutamiento, como el tipo de red y el algoritmo de enrutamiento, puede optimizar la transferencia de datos significativamente. Aunque ya se han probado algunas configuraciones, podría ser útil investigar más a fondo y experimentar con diferentes ajustes finos para encontrar la configuración óptima.

VII. CONCLUSIONES

Los resultados experimentales han demostrado que la paralelización puede mejorar significativamente el rendimiento del algoritmo de Floyd-Warshall, reduciendo los tiempos de ejecución en comparación con la versión serial. Sin embargo, también se ha observado que el rendimiento escalable tiene límites, especialmente cuando la comunicación entre procesos se convierte en un cuello de botella.
Al utilizar OpenMP, se obtuvo un speedup cercano al óptimo con hasta cuatro hilos, lo que coincide con el número de procesadores lógicos disponibles en las máquinas de prueba. Sin embargo, el rendimiento no mejoró al aumentar el número de hilos más allá de este punto, lo que sugiere que el paralelismo en memoria compartida tiene un límite práctico basado en el hardware disponible.

La implementación con MPI superó las restricciones de paralelización de OpenMP, permitiendo el uso de más núcleos y la distribución del trabajo entre varios nodos de un clúster. Aunque se logró un speedup significativo con hasta ocho nodos, el rendimiento disminuyó al escalar a más nodos debido al overhead de comunicación y las restricciones de memoria.
Las posibles mejoras para futuros trabajos incluyen la reconsideración de un enfoque híbrido MPI-OpenMP, que podría ofrecer un mejor balance entre la paralelización en memoria local y distribuida, así como la adaptación del algoritmo de Dijkstra para MPI, lo cual podría ofrecer una alternativa más eficiente para ciertos escenarios, y por

último se pensó en un ajuste significativo de la codificación actual aprovechando un uso más enriquecedor del MPI.
En conclusión, el informe resalta la importancia de la computación de alto rendimiento en la resolución de problemas complejos en la teoría de grafos y demuestra que, aunque la paralelización es una estrategia poderosa, su aplicabilidad efectiva depende del diseño y una comprensión profunda del problema y del entorno de ejecución.
Con el avance continuo en la tecnología de HPC y una mayor comprensión de las técnicas de paralelización, se espera que futuras investigaciones puedan superar las barreras actuales y ofrecer soluciones aún más eficientes para el problema APSP en grafos de gran escala.

VIII. REFERENCIAS

[1] APSP. University of Toronto [Online] http://www.cs.toronto.edu/~lalla/373s16/notes/APSP.pdf.

[2] Algoritmo de Floyd Warshall. Medium [Online] . https://medium.com/algoritmo-floyd-warshall/algoritmo-de-floyd-warshall-e1fd1a900d8.

[3] Time and Space Complexity of Floyd Warshall Algorithm. Geeksforgeeks [Online]. https://www.geeksforgeeks.org/time-and-space-complexity-of-floyd-warshall-algorithm

[4] Estudio de Eficiencia. NLHPC [Online] https://wiki.nlhpc.cl/Estudio_de_Eficiencia.