Librería C para Cómputo Paralelo de Caminos Mínimos en Grafos sobre Arquitecturas Multicore

Jerónimo Lambre[1] and Enzo Rucci[2,3] ✉

[1] Facultad de Informática, UNLP. La Plata (1900), Argentina
[2] III-LIDI, Facultad de Informática, UNLP – CIC. La Plata (1900), Argentina
erucci@lidi.info.unlp.edu.ar
[3] Comisión de Investigaciones Científicas (CIC). La Plata (1900), Argentina

Resumen Los grafos han adquirido una relevancia significativa para modelar y resolver problemas en diversas áreas. El algoritmo Floyd-Warshall (FW) permite hallar los caminos mínimos entre todos los vértices de un grafo pesado. Debido a su alta demanda computacional ($O(n^3)$), muchos esfuerzos se han realizado en las últimas 2 décadas para acelerarlo. Sin embargo, las propuestas existentes suelen obviar los aspectos funcionales para priorizar los vinculados al rendimiento, lo que limita seriamente su reúso. Es por lo que en este artículo se presenta el diseño y desarrollo de una librería en C que provee implementaciones optimizadas del algoritmo FW para arquitecturas multicore. La librería diseñada soporta grafos de cualquier tamaño, múltiples tipos de datos, y compatibilidad con diferentes formatos de archivo (JSON y CSV) y sistemas operativos (Windows y Linux). Adicionalmente, también permite configurar diferentes aspectos de la ejecución, tanto funcionales como de rendimiento. Los resultados experimentales muestran que es capaz de lograr un aprovechamiento alto de los recursos del sistema de prueba, a un muy bajo esfuerzo de programación por parte del usuario.

Keywords: FW · Librería · Caminos mínimos · OpenMP ·

1. Introducción

Desde sus inicios, la importancia de los grafos y sus algoritmos ha sido evidente, ya que proporcionan herramientas esenciales para modelar y abordar problemas en áreas muy diferentes [10]. El algoritmo de Floyd-Warshall (FW) [4,18] permite conocer los caminos mínimos entre todos los vértices de un grafo pesado. A lo largo de la historia, se lo ha empleado en ámbitos diversos como el tráfico automovilístico [8], las redes de computadoras [9], bioinformática [11], computación gráfica [17], entre otros. Sin embargo, FW es computacionalmente costoso ($O(n^3)$) y a medida que el tamaño del problema escala, el empleo de recursos de cómputo paralelo se vuelve necesario para poder satisfacer los requerimientos de tiempo y eficiencia.

El estudio de la aceleración de FW en CPU lleva más de 2 décadas. Las primeras implementaciones estuvieron orientadas a arquitecturas monoprocesador. Tanto

Penner y Prasanna [12] como Venkataraman et al. [16] mostraron que, bajo ciertas condiciones, es posible reordenar el cómputo de las celdas para implementar técnicas de *blocking*. Este reordenamiento permite una mayor explotación de la localidad de datos, lo que llevó a un aumento en el rendimiento de aproximadamente 2×. En sentido similar, Han and Kang [6] y Han et al. [5] demostraron que, el uso de instrucciones vectoriales (en particular de la familia SSE) en combinación con técnicas de desenrollado de bucle, pueden mejorar el rendimiento hasta 5.7×. Más adelante en el tiempo, se pueden encontrar implementaciones para arquitecturas multiprocesador de memoria compartida [19], de memoria distribuida [15] y de memoria híbrida [14]. Para las arquitecturas Xeon Phi de Intel, Hou et al. [7] propuso una implementación OpenMP para coprocesadores KNC. Con la salida de la generación KNL, Rucci et al. [13] exploró su uso para acelerar FW, mientras que Costi lo extendió [2]. En forma alternativa, Endo propuso un enfoque recursivo para resolver el problema [3]. Finalmente, Calderón et al. [1] adaptó el código de [2] para que pueda ser ejecutado en procesadores multicore de propósito general y propuso una nueva optimización a través de mecanismos de sincronización de grano más fino.

Las propuestas existentes suelen obviar los aspectos funcionales para priorizar los vinculados al rendimiento, lo que limita seriamente su reúso. En este artículo se presenta el diseño y desarrollo de una librería en C que provee funciones para cómputo paralelo de caminos mínimos en grafos usando el algoritmo FW sobre arquitecturas multicore. Tomando como base el mencionado trabajo realizado en [1], se adaptó y extendió el código para que sea capaz de funcionar con grafos de cualquier tamaño y tipo de dato, provenientes de archivos reales, sin afectar al aprovechamiento eficiente de recursos. La creación de una librería especializada simplifica la programación y el reúso en aplicaciones de terceros, evitando la duplicidad de código y minimizando errores. También contribuye a una mayor agilidad, al reducir los tiempos y costos de desarrollo.

El resto del artículo se organiza de la siguiente forma. La Sección 2 introduce el marco referencial para este trabajo. Luego, la Sección 3 describe el diseño y desarrollo de la librería propuesta. A continuación, la Sección 4 muestra los resultados experimentales obtenidos mientras que la Sección 5 resume las conclusiones junto al trabajo futuro.

2. Marco Referencial

2.1. Algoritmo Floyd-Warshall

El algoritmo FW tiene como objetivo la búsqueda del camino mínimo entre cada par de los N vértices de un grafo dirigido. Tiene como resultado dos matrices de tamaño $N \times N$, descritas a continuación:

- Una matriz de distancias D, donde cada celda $D_{i,j}$ indica el costo mínimo entre cada par de vértices i y j.

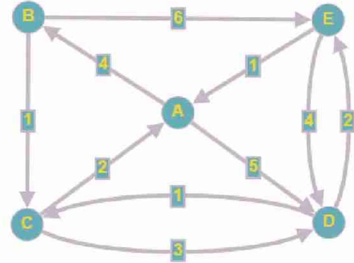
- Una matriz de reconstrucción del camino P, donde cada celda $P_{i,j}$ indica el ante-último vértice del camino mínimo desde el vértice i hasta el vértice j.

La matriz D se inicializa con las distancias a los vértices vecinos inmediatos, y la matriz P con el valor i en cada fila i, para aquellos vértices que tengan conexión. Luego, como se muestra en la Figura 1, para cada vértice origen i y vértice destino j, se realiza una comparación entre el costo mínimo $D_{i,j}$ conocido hasta el momento, y el costo obtenido al pasar por un vértice intermedio k, es decir, $D_{i,k} + D_{k,j}$. Si el segundo caso retorna una distancia menor, se actualiza el costo mínimo $D_{i,j}$ con dicho valor, así como también se asigna $P_{i,j} = k$.

Figura 1: Pseudocódigo del algoritmo FW clásico

for $k \leftarrow 0$ to $N-1$ **do**
  for $i \leftarrow 0$ to $N-1$ **do**
    for $j \leftarrow 0$ to $N-1$ **do**
      if $D_{i,j} \geq D_{i,k} + D_{k,j}$ **then**
        $D_{i,j} \leftarrow D_{i,k} + D_{k,j}$
        $P_{i,j} \leftarrow k$
      end if
    end for
  end for
end for

La Figura 2 presenta un ejemplo de aplicación del algoritmo FW a un grafo dirigido compuesto por 5 nodos. Tras la ejecución del algoritmo, se obtienen las matrices D y P como resultado.



(a) Grafo de entrada (rep. visual)

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0

(b) Entrada: Matriz de adyacencias (D)

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

(c) Salida: Matriz de distancias mínimas (D)

	A	B	C	D	E
A	0	0	1	0	3
B	2	1	1	2	1
C	2	0	2	2	3
D	2	2	3	3	3
E	4	0	3	4	4

(d) Salida: Matriz de caminos mínimos (P)

Figura 2: Ejemplo de aplicación del algoritmo FW a un grafo dirigido de 5 nodos

2.2. Código de base

Se empleó el de [1], el cual fue desarrollado para ser ejecutado en procesadores multicore de propósito general. A continuación, se describen las optimizaciones contempladas:

- **Procesamiento por bloques**. Se implementa el reordenamiento en el cómputo propuesto en [12,16] para poder explotar localidad de datos. Esto implica que la matriz de distancias se divide en bloques de tamaño $BS \times BS$ y el algoritmo se organiza en $R = \frac{N}{BS}$ rondas, cada una compuesta por 4 fases. La fase 1 procesa el bloque diagonal principal, las fases 2 y 3 procesan los bloques de la misma fila y columna respectivamente, y la fase 4 se encarga del resto de los bloques. Este enfoque garantiza que las dependencias de datos se respeten y optimiza el uso de la memoria caché, mejorando el rendimiento del algoritmo.
- **Multi-hilado**. Se utiliza OpenMP para obtener una versión multi-hilada del algoritmo, distribuyendo bloques entre diferentes hilos mediante la directiva `for` con *scheduling* dinámico para mejorar el balance de carga y optimizar el uso del tiempo de CPU.
- **Vectorización**. Uso de la directiva `simd` de OpenMP en combinación con flags de compilación para garantizar aprovechamiento de instrucciones SIMD (SSE, AVX, AVX-512).
- **Alineación de datos**. Uso de `posix_memalig()` (Linux) y `_aligned_malloc()` (Windows) para asignar bloques de memoria alineados, optimizando las operaciones de lectura y escritura posteriores.
- **Predicción de saltos**. Optimización de la predicción de saltos en funciones frecuentemente accedidas mediante el uso de la macro `__builtin_expect`. Esta técnica ayuda al compilador a organizar el código de manera que se alinee con los patrones de ejecución más probables, reduciendo así los fallos en la predicción de saltos y, potencialmente, los fallos de caché resultantes.
- **Aumento de concurrencia intra-ronda**: En lugar de esperar a que las fases 2 y 3 terminen por completo para iniciar la fase 4, esta optimización adelanta el cómputo de aquellos bloques de la última fase cuyas dependencias ya fueron resueltas durante el procesamiento de las primeras. Para implementar esta sincronización de grano fino, se utilizan semáforos de la librería POSIX.

Las optimizaciones mencionadas reflejan un enfoque integral hacia la mejora del rendimiento de la implementación del algoritmo FW en procesadores multicore.

3. Implementación

En la siguiente sección se detalla el proceso seguido para construir la biblioteca propuesta, exponiendo los detalles técnicos y las decisiones de diseño adoptadas para una librería de software funcional y eficiente.

3.1. Interfaz de librería

La interfaz de la biblioteca que implementa el algoritmo paralelo está diseñada para ofrecer una estructura clara y eficiente para los usuarios, permitiéndoles interactuar con las funciones esenciales de manera intuitiva. La interfaz se define a través de dos archivos de cabecera principales.

`FW_Lib_CommonTypes.h` contiene las definiciones de tipos y constantes que son utilizados en toda la biblioteca. Entre las definiciones clave se incluyen:

- **DataType**: define los tipos de datos que pueden ser utilizados en las matrices y en el algoritmo: *TYPE_INT*, *TYPE_FLOAT*, *TYPE_DOUBLE*, y *UNDEFINED* (para autodetección).
- **FileType**: especifica los formatos de archivo soportados para entrada y salida: *CSV* y *JSON*.
- **FW_Matrix**: estructura principal de la librería que incluye punteros a las matrices de distancias (*dist*) y caminos (*path*), el tipo de archivo (*fileType*), la longitud de la parte decimal en caso de utilizar float o double (*decimal_length*), el tipo de datos de las distancias (*datatype*) y el tamaño normalizado de la matriz (*norm_size*).
- **FW_attr_t**: estructura que almacena atributos adicionales para la configuración de la ejecución del algoritmo, tales como *text_in_output* (para imprimir texto en las matrices de salida), *print_distance_matrix* (para imprimir la matriz de distancias), *no_path* (para no calcular la matriz de caminos), y *thread_num* (el número de hilos para la ejecución paralela).
- **Macros y Constantes**: Se definen macros para la exportación de funciones (*LIB_EXPORT*) y códigos de error (*EXIT_ALOCATION_FAILED*, *EXIT_OPEN_FILE_ERROR*).

`FW_Lib_Functions.h` proporciona las declaraciones de funciones que implementan las diversas operaciones de la biblioteca, las cuales se detallan a continuación:

- Gestión de datos de entrada
  1. **FW_Matrix fwl_matrix_create(DataType dataType, char *path, FW_attr_t *attr)**: Crea una estructura *FW_Matrix* basada en el tipo de datos (*dataType*), leyendo la matriz almacenada en la ruta *path* y ordenándola en bloques de tamaño *BS* (definido en compilación). Si el tipo de datos es *UNDEFINED*, la función lo autodetecta.
  2. **void fwl_matrix_free(FW_Matrix *)**: Libera la memoria asociada a las matrices de distancias y caminos de una estructura *FW_Matrix*, garantizando una correcta gestión de los recursos.
- Gestión de la ejecución del algoritmo FW
  1. **void fwl_matrix_parallel_search(FW_Matrix, FW_attr_t * attr)**: Realiza el cálculo del algoritmo FW de manera paralela, utilizando la can-

tidad de threads definida en *attr*, actualizando las matrices de distancias y caminos de la estructura *FW_Matrix* indicada como parámetro.

2. **void fwl_matrix_sequential_search(FW_Matrix, FW_attr_t *)**: Realiza el cálculo del algoritmo FW de manera secuencial, actualizando las matrices de distancias y caminos de la estructura *FW_Matrix* indicada como parámetro.
3. **FW_attr_t fwl_attr_new()**: Crea un nuevo objeto *FW_attr_t* con valores predeterminados, facilitando la configuración inicial.
4. **void fwl_attr_init(FW_attr_t *attr)**: Inicializa un objeto *FW_attr_t* con valores predeterminados, asegurando que todos los campos tengan valores válidos antes de su uso.

- Gestión de datos de salida
  1. **void fwl_matrix_save(FW_Matrix FW, char *path, char *name, FileType fileType, FW_attr_t *attr)**: Guarda la matriz resultante en la ruta *path* con nombre *name*_path o *name*_distances de tipo JSON o CSV según el parámetro *fileType*. Se pueden configurar opciones adicionales con el parámetro opcional *attr*.
  2. **char* fwl_matrix_get_info(FW_Matrix *element)**: Devuelve la información de un elemento de tipo *FW_Matrix* en forma de string.
  3. **char * fwl_attr_get_info(FW_attr_t * attr)**: Devuelve la información de un elemento de tipo *FW_attr_t* en forma de string.
- Análisis de rendimiento
  1. **double fwl_get_create_time()**: Devuelve el tiempo de la lectura del archivo de entrada y la creación de la estructura *FW_Matrix*.
  2. **double fwl_get_search_time()**: Devuelve el tiempo de procesamiento del algoritmo FW.
  3. **double fwl_get_save_time()**: Devuelve el tiempo de guardado de las matrices de distancias y caminos almacenadas en la estructura *FW_Matrix* recibida como parámetro en un archivo.
  4. **double fwl_get_total_time()**: Devuelve el tiempo total tomado por la librería, incluyendo creación, procesamiento y guardado, proporcionando una medida completa del rendimiento.
  5. **double get_fw_performance(FW_Matrix *matrix)**: Devuelve el número de operaciones de punto flotante por segundo (en GFLOPS) u operaciones de enteros por segundo (en GIOPS) que el sistema realizó durante la ejecución del algoritmo. Recibe una estructura *FW_Matrix* para realizar los cálculos según el tamaño de la matriz.

Esta interfaz proporciona una base sólida y flexible para la implementación y ejecución del algoritmo de FW en paralelo.

3.2. Soporte para tipos de datos

La biblioteca desarrollada posee la capacidad de manejar diversos tipos de datos, lo que resulta crucial para adaptarse a diferentes requisitos de precisión y tipos de datos específicos de las aplicaciones que la utilicen. En particular, se da soporte a (*int*) y a punto flotante tanto de precisión simple (*float*) como doble (*double*).

Implementar esta capacidad presentó diversos desafíos, principalmente relacionados con la gestión de memoria y la precisión de los cálculos. Cada tipo de dato exige diferentes consideraciones en términos de operaciones aritméticas, tamaño de almacenamiento y manejo de valores especiales como el infinito.

La librería tiene la capacidad de auto-detectar el tipo de dato de las entradas si no es especificado explícitamente. Este proceso se realiza durante la carga de matrices desde archivos, donde se analizan los datos para determinar si contienen puntos decimales o caracteres específicos que indiquen un tipo particular.

Las funciones anteriores permiten que la librería ajuste dinámicamente su comportamiento para optimizar el manejo de datos y la ejecución del algoritmo según el tipo detectado, lo que aumenta la eficiencia y la flexibilidad del sistema.

3.3. Soporte para formatos de archivo y normalización de dimensiones

La librería admite dos formatos principales de archivo permitiendo adaptabilidad a diferentes entornos y necesidades de los usuarios (JSON y CSV), siendo elegibles tanto para la entrada como para la salida del procesamiento.

En caso de que el tamaño de la matriz de entrada no sea múltiplo de *BS*, entonces será necesario *normalizar* sus dimensiones. El tamaño de la matriz puede ser leído contando la cantidad de comas que existen en una fila del archivo CSV o leyendo el atributo *size* en caso de JSON. Una vez obtenidas las dimensiones de la matriz original, se realiza la normalización de este valor chequeando cuál es el múltiplo de *BS* siguiente. Ambos valores se almacenan en una estructura *FW_Matrix* en los atributos *size* y *norm_size*. Por ejemplo, si *N=518* y *BS=32* entonces *size=518* y *norm_size=544*.

Una vez obtenida esta información, se procede con la lectura de los datos del archivo, la cual se realiza carácter por carácter. Si bien este enfoque puede penalizar el rendimiento, fue seleccionado para permitir que los buffers de lectura no se saturen y también poder manejar la memoria al usar grandes volúmenes de datos. Además, en esta etapa también se rellena con valores "infinitos" (INT_MAX, FLT_MAX o DBL_MAX según corresponda) los valores necesarios para llevar la matriz al tamaño normalizado ya calculado.

3.4. Soporte para sistemas operativos y construcción de la librería

FW_Lib está diseñada para ofrecer soporte tanto en plataformas Windows como Linux. El archivo `makefile` provisto es un componente crucial que automatiza

el proceso de construcción de la biblioteca, ya que es el encargado de compilar todos los módulos necesarios y de enlazarlos, gestionando dependencias internas y asegurando que todos los archivos objeto estén actualizados. Durante este proceso, se aplican las flags `-O3`, `-march=native` y `-fopenmp` para gestionar la optimización y habilitar el soporte para ejecución paralela. Finalmente, en este archivo es donde se debe configurar el sistema operativo destino (Linux, Windows) y el tipo de librería a generar (estática o dinámica).

3.5. Aspectos de rendimiento

Hay dos parámetros que resultan esenciales en el rendimiento final de la librería:

- La cantidad de hilos: La biblioteca permite ajustar este número para maximizar la utilización de los recursos del procesador. En particular, se puede modificar utilizando la estructura *FW_attr_t* ya que se envía como argumento a las funciones que procesan la matriz. Por defecto, el valor es la cantidad núcleos disponibles en el sistema (se usa la función `get_nprocs()` de la librería `sysinfo.h` para Linux y `GetSystemInfo()` de `windows.h` para Windows).
- El tamaño de bloque (*BS*): determina la granularidad del paralelismo y afecta directamente la eficiencia del acceso a la memoria y la carga de trabajo distribuida entre los hilos. Por defecto se ha establecido en 128 considerando que es el que mejor se ajusta en una variedad de equipos diferentes [1]. Sin embargo, el usuario final puede cambiarlo a otro valor que se adapte mejor a su sistema.

3.6. FW-App

Si bien la utilización de la librería resulta sencilla (sólo requiere de una pocas líneas de código), se desarrolló una pequeña aplicación que hace uso de FW_Lib para facilitar aun más su adopción mediante la provisión de un ejemplo. Adicionalmente, se brinda al mismo tiempo una herramienta final para computar caminos mínimos en grafos según FW.

FW-App admite diferentes parámetros que permiten configurar la ruta a la ubicación del archivo de entrada, el tipo de datos, la cantidad de hilos y el tamaño de bloque. El archivo `FW_app.c` contiene los llamados a la librería FW_Lib, como se muestra en la Figura 3. Se puede notar que la matriz de entrada debe ser cargada en una estructura de tipo *FW_Matrix* mediante la función `fwl_matrix_create()` para poder ser enviada posteriormente a la función `fwl_matrix_parallel_search()`. Finalmente, el resultado se puede persistir mediante el llamado a la función `fwl_matrix_save()`.

4. Resultados

4.1. Funcionales

```
// Arguments Init
FW_attr_t attr;
fwl_attr_init(&attr);

printf("\nFW-App v1.0\n\n");
printf("Input file: %s\n\n", path);

printf("Execution mode: \n");
printf("%s", fwl_attr_get_info(&attr));

printf("\nLoading Graph ...\n");
FW_Matrix data = fwl_matrix_create(dataType, path, &attr); // Read
printf("%s", fwl_matrix_get_info(&data));
printf("Done\n\n");

printf("Computing FW Algorithm ...\n");
fwl_matrix_paralell_search(data, &attr); // Process
printf("Done\n\n");

printf("Saving Results ...\n");
fwl_matrix_save(data, "./output/", "ResultParalell.csv", CSV, &attr); // Save
printf("Done\n\n");

fwl_matrix_free(&data); // Free memory
```

Figura 3: Extracto de código del archivo fuente de FW-App

Para ilustrar los resultados del funcionamiento de la librería, se hace uso de FW-App con el grafo dirigido de la Figura 2a como caso de prueba. La Figura 4 muestra el contenido del archivo de entrada en formato JSON correspondiente al grafo mencionado mientras que la Figura 5 exhibe la ejecución de FW-App al tomarlo como entrada. Se puede notar que la salida muestra el procesamiento realizado por la librería en cuanto a la detección del tipo de dato y a la re-organización de los datos para su posterior procesamiento optimizado. Finalmente, la Figura 6 exhibe el resultado de la ejecución en formato CSV (como fue indicado en el código), el cual coincide con el presentado en la Figura 2c.

```
cat ./examples/5.json

{
  "type": "int",
  "size": 5,
  "matrix": [
    [0, 4, "INF", 5, "INF"],
    ["INF", 0, 1, "INF", 6],
    [2, "INF", 0, 3, "INF"],
    ["INF", "INF", 1, 0, 2],
    [1, "INF", "INF", 4, 0]
  ]
}
```

Figura 4: Contenido del archivo JSON de entrada.

4.2. Rendimiento

Las pruebas de rendimiento se realizaron en un sistema equipado con procesador Intel Core i5-10400F (6 cores), 32 GB de memoria RAM, sistema operativo Debian 11 y `gcc` v10.1. Además, se contempló la variación del tamaño de las matrices de entrada, del tipo de datos y de la cantidad de hilos. En particular, se emplearon matrices de tamaño N=$\{2000,4000,8000\}$ con un porcentaje de completitud del 70 % (es decir, 30 % de las posibles conexiones

```
./apps/linux/FW.bin -p ./examples/5.json


FWL v1.0

Input file: ./examples/5.json

Execution mode:
=> 8 threads
=> Including paths
=> Infinite outputs as INF

Loading Graph ...
-> File format: JSON
-> Datatype: INT
-> Matrix Size: 5
-> Matrix Normalized Size: 6
-> Block Size: 2
Done

Computing FW Algorithm ...
Done

Saving Results ...
Done

Compute Time: 0.000471
Compute Speed: 0.000917 GIOPS
```

Figura 5: Ejecución de FW-app y salida.

```
cat ./output/ResultParalell_distances.csv

0,4,5,5,7
2,6,0,3,5
3,7,1,0,2
1,5,5,4,0

cat ./output/ResultParalell_path.csv

0,0,1,0,3
2,1,1,2,1
2,0,2,2,3
2,2,3,3,3
4,0,3,4,4
```

Figura 6: Resultados en formato CSV

entre nodos no existen). Por último, cada prueba particular fue repetida 10 veces y se computó el promedio de sus valores para el análisis de resultados.

La Figura 7 presenta la eficiencia lograda por FW-App para T=6 al variar N y el tipo de datos. Se puede observar que los valores de eficiencia se mantienen altos para todos los casos (por encima del 95 %), lo que denota el destacado aprovechamiento de recursos que el código alcanza.

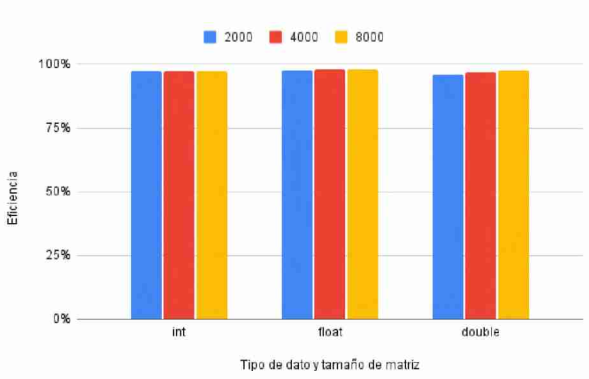


Figura 7: Eficiencia de FW-App para T=6 en equipo de pruebas

5. Conclusiones y Trabajo Futuro

En el presente trabajo, se desarrolló una librería en C para el cómputo paralelo de caminos mínimos en grafos utilizando el algoritmo FW sobre arquitecturas multi-core. La librería fue diseñada para soportar grafos de cualquier tamaño, múltiples tipos de datos, y compatibilidad con diferentes formatos de archivo (JSON y CSV) y sistemas operativos (Windows y Linux). Adicionalmente, también permite configurar diferentes aspectos de la ejecución, tanto funcionales como de rendimiento.

Los resultados experimentales muestran que la librería es capaz de lograr un aprovechamiento alto de los recursos del sistema de prueba, a un muy bajo esfuerzo de programación por parte del usuario. Al ponerla a disposición en un repositorio público [1], se espera que ésta contribuya a reducir los tiempos y costos de desarrollo de aplicaciones que requieran cómputos de caminos mínimos en grafos.

Como trabajos futuros, se proponen las siguientes ideas:

- Extender las pruebas realizas considerando grafos más grandes y equipos con otras características de harwdare.
- Incorporar soporte para otros algoritmos de caminos mínimos y manejar grafos dinámicos, donde los pesos de las aristas pueden cambiar durante la ejecución del algoritmo. También dar soporte a otros formatos de archivo.
- Desarrollar bindings para lenguajes como Python y Java, facilitando la integración y uso de la librería en diversos proyectos y ampliando su accesibilidad.

Referencias

1. Calderón, S., Rucci, E., Chichizola, F.: Enhanced openmp algorithm to compute all-pairs shortest path on x86 architectures. In: Pesado, P., Panessi, W., Fernández, J.M. (eds.) Computer Science – CACIC 2023. pp. 46–61. Springer Nature Switzerland, Cham (2024)
2. Costi, U.: Aceleración del Algoritmo Floyd-Warshall sobre Intel Xeon Phi KNL. Tesina de Licenciatura en Informática, Universidad Nacional de La Plata (2020)
3. Endo, T.: Integrating cache oblivious approach with modern processor architecture: The case of floyd-warshall algorithm. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. p. 123–130. HPCAsia '20, ACM, New York, NY, USA (2020). https://doi.org/10.1145/3368474.3368477
4. Floyd, R.W.: Algorithm 97: Shortest path. Commun. ACM **5**(6), 345– (Jun 1962). https://doi.org/10.1145/367766.368168
5. Han, S.C., Franchetti, F., Püschel, M.: Program generation for the all-pairs shortest path problem. In: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques. p. 222–232. PACT '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1152154.1152189
6. Han, S., Kang, S.: Optimizing all-pairs shortest-path algorithm using vector instructions (2006)

[1] Disponible en: `https://github.com/JeronimoLam/PPS-FloydWarshallLib`

7. Hou, K., Wang, H., c. Feng, W.: Delivering parallel programmability to the masses via the intel mic ecosystem: A case study. In: 2014 43rd International Conference on Parallel Processing Workshops. pp. 273–282 (Sept 2014). https://doi.org/10.1109/ICPPW.2014.44
8. Jalali, S., Noroozi, M.: Determination of the optimal escape routes of underground mine networks in emergency cases. Safety Science **47**(8), 1077 – 1082 (2009). https://doi.org/http://dx.doi.org/10.1016/j.ssci.2009.01.001
9. Khan, P., Konar, G., Chakraborty, N.: Modification of floyd-warshall's algorithm for shortest path routing in wireless sensor networks. In: 2014 Annual IEEE India Conference (INDICON). pp. 1–6 (Dec 2014). https://doi.org/10.1109/INDICON.2014.7030504
10. L. R. Foulds: Graph Theory Applications. Springer (1992), https://doi.org/10.1007/978-1-4612-0933-1
11. Nakaya, A., Goto, S., Kanehisa, M.: Extraction of correlated gene clusters by multiple graph comparison. Genome Informatics **12**, 44–53 (2001)
12. Penner, M., Prasanna, V.K.: Cache-friendly implementations of transitive closure. In: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques. pp. 185–. PACT '01, IEEE Computer Society, Washington, DC, USA (2001)
13. Rucci, E., De Giusti, A., Naiouf, M.: Blocked All-Pairs Shortest Paths Algorithm on Intel Xeon Phi KNL Processor: A Case Study. In: De Giusti, A.E. (ed.) Computer Science – CACIC 2017. pp. 47–57. Springer Int. Pub., Cham (2018)
14. Solomonik, E., Buluç, A., Demmel, J.: Minimizing communication in all-pairs shortest paths. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. pp. 548–559. IEEE (2013)
15. Srinivasan, T., Balakrishnan, R., Gangadharan, S., Hayawardh, V.: A scalable parallelization of all-pairs shortest path algorithm for a high performance cluster environment. In: 2007 International Conference on Parallel and Distributed Systems. pp. 1–8 (2007). https://doi.org/10.1109/ICPADS.2007.4447721
16. Venkataraman, G., Sahni, S., Mukhopadhyaya, S.: A Blocked All-Pairs Shortest-Paths Algorithm, pp. 419–432. Springer Berlin Heidelberg (2000). https://doi.org/10.1007/3-540-44985-X_36
17. Wang, L., Springer, M., Heibel, H., Navab, N.: Floyd-warshall all-pair shortest path for accurate multi-marker calibration. In: 2010 IEEE International Symposium on Mixed and Augmented Reality. pp. 277–278 (2010). https://doi.org/10.1109/ISMAR.2010.5643605
18. Warshall, S.: A theorem on boolean matrices. J. ACM **9**(1), 11–12 (Jan 1962). https://doi.org/10.1145/321105.321107
19. Zhang, L.y., Jian, M., Li, K.p.: A parallel floyd-warshall algorithm based on tbb. In: 2010 2nd IEEE International Conference on Information Management and Engineering. pp. 429–433 (2010). https://doi.org/10.1109/ICIME.2010.5477752