FACULTAD DE INFORMÁTICA | FACULTAD DE INGENIERÍA
UNIVERSIDAD NACIONAL DE LA PLATA

PRÁCTICA PROFESIONAL SUPERVISADA

Optimización del Cómputo de Caminos Mínimos en Grafos usando el Algoritmo Floyd-Warshall sobre Arquitecturas Multicore

---

Sergio Calderón
2285/4

Tutor: Dr Enzo Rucci

15 de marzo de 2023

1. Resumen

Los grafos han adquirido una relevancia significativa para modelar y resolver problemas en áreas diversas. Una operación frecuente es la búsqueda de los caminos mínimos entre sus vértices, siendo el algoritmo Floyd-Warshall (FW) una solución óptima. 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 sus requerimientos. En este informe, se muestra la adaptación y optimización de FW en arquitecturas multicore x86 de propósito general, a partir de un código diseñado para un acelerador específico (Xeon Phi KNL). Se parte desde una versión paralela que emplea una técnica de blocking, y luego se describen los incrementos aplicados, que son mejoras respecto a su versión predecesora. Adicionalmente, se propone una nueva optimización implementada a través de dos variantes, las cuales combinan el uso de directivas de OpenMP con mecanismos de sincronización de POSIX Threads (semáforos y variables condición). Las pruebas realizadas en un servidor equipado con 2×Intel Xeon Platinum 8276L mostraron una mejora máxima del 24 % respecto al código base y una estabilización del tamaño de bloque óptimo, independientemente del tipo de dato, cantidad de hilos o tamaño de entrada. Por último, el código fuente está disponible en un repositorio público para beneficio de la comunidad académica y científica (`https://bit.ly/pps-fw-xeonplatinum`).

Palabras clave: Floyd Warshall, multicore, HPC, técnica de blocking, OpenMP, semáforos, variables condición, grafo, caminos mínimos.

2. Lugar de Trabajo

2.1. Antecedentes

El Instituto de Investigación en Informática LIDI (III-LIDI) es una Unidad de Investigación, Desarrollo y Transferencia reconocida por el Consejo Superior de la Universidad Nacional de La Plata, que funciona en la Facultad de Informática de la UNLP.

Fue creado en Marzo de 1984 como Laboratorio de Computación (LAC) por el Departamento de Matemáticas de la Facultad de Ciencias Exactas (del que dependía Informática). Luego, en Diciembre de 1986, fue denominado Laboratorio de Investigación y Desarrollo en Informática (LIDI) dentro del nuevo Departamento de Informática de la Facultad de Ciencias Exactas. Más adelante, en 1995, fue aprobado formalmente como Laboratorio de la Facultad de Ciencias Exactas, por el Honorable Consejo Superior de la Universidad Nacional de La Plata (Expte. 700-39323). Cuatro años más tarde, con la creación de la Facultad de Informática, pasó a integrar dicha nueva Unidad Académica. Finalmente, en febrero de 2003 fue aprobado como Instituto por el Honorable Consejo Académico de la Facultad de Informática para ser ratificado en 2005 por el Honorable Consejo Superior de la UNLP.

Actualmente, el III-LIDI forma parte del Sistema Nacional de Computación de Alto Desempeño (SNCAD) del MINCyT por Resolución 065/12 de Diciembre 2012. Además, reviste como Instituto de Investigación Asociado a la Comisión de Investigaciones Científicas (CIC) desde Noviembre de 2016.

2.2. Objetivos

Entre sus objetivos se encuentran:

- Realizar investigación en Informática poniendo énfasis en las áreas tecnológicas cuyo conocimiento y desarrollo tengan significación para el país.
- Contribuir a la formación, actualización y especialización de recursos humanos en Informática.
- Realizar desarrollos concretos que signifiquen una transferencia de tecnología desde la Universidad a la sociedad.

2.3. Organización

La organización directiva del III-LIDI consta de una directora y de un director adjunto como máximas autoridades. Asimismo, el Instituto también cuenta con la participación de un director fundador y de un consejo directivo y otro científico-tecnológico.

El Consejo Directivo se encarga de tratar todas las funciones legislativas pertinentes al funcionamiento del Instituto, y un Consejo Científico-Tecnológico que conforma el órgano principal de asesoramiento a la dirección en lo referido a líneas de Investigación, Desarrollo y Transferencia del Instituto, así como en las gestiones de recursos. La Tabla 2.1 presenta las autoridades actuales.

Tabla 2.1

Autoridad	
Directora	Lic. Patricia Pesado
Director Adjunto	Dr Marcelo Naiouf
Director Fundador	Ing Armando De Giusti
Titulares del Consejo Directivo	Esp. Chichizola, Franco Dra. Lanzarini, Laura Dr. Pasini, Ariel Dr. Pousa, Adrián Dra. Sanz, Cecilia Lic. Boracchia, Marcos Mg. Encinas, Diego Lic. Rodríguez, Eguren Sebastián
Suplentes del Consejo Directivo	Mg. Thomas, Pablo Dr. Rucci, Enzo Ing. Medina, Santiago Sr. Torres, Juan Ignacio
Titulares del Consejo Científico-Tecnológico	Dra. De Giusti, Laura Mg. Esponda, Silvia Dr. Hasperué, Waldo

Además de los anteriores, el laboratorio se encuentra integrado por:

- Personal Científico y Académico formado A (con categoría docente-investigador I, II o III)
- Personal Científico y Académico formado B (sin categoría I, II o III)
- Personal Científico y Académico en formación.
- Ayudantes, Pasantes y Becarios Alumnos
- Personal técnico y de apoyo
- Personal administrativo

3. INTRODUCCIÓN

Desde sus inicios, los grafos y sus algoritmos se han vuelto muy importantes, ya que permiten modelar y resolver problemas de dominios muy diferentes como computación científica, minería de datos, procesamiento de imágenes, ruteo de redes, entre otros [1]. Entre estos algoritmos, se encuentra el de Floyd-Warshall (FW) [2, 3], el cual permite calcular (y hallar) los caminos mínimos entre todos los vértices de un grafo pesado. A este algoritmo en toda su historia se lo ha empleado en ámbitos diversos como el tráfico automovilístico [4], las redes de computadoras [5], bioinformática [6], computación gráfica [7], 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 sus requerimientos. Es por lo que la comunidad científica ha realizado múltiples esfuerzos con ese propósito.

Las primeras implementaciones aceleradas de FW estuvieron orientadas a arquitecturas monoprocesador. Tanto Penner y Prasanna [8] como Venkataraman et al. [9] 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 2x. En sentido similar, Han and Kang [10] y Han et al. [11] 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 [12], de memoria distribuida [13] y de memoria híbrida [14]. Por último, para las arquitecturas Xeon Phi de Intel se pueden mencionar algunas propuestas propuestas. Hou et al. [15] propuso una implementación OpenMP para coprocesadores KNC. Con la salida de la generación KNL, Rucci et al. [16] exploró su uso para acelerar FW, mientras que Costi lo extendió [17].

En esta PPS, se propone optimizar el cómputo de caminos mínimos en grafos usando el algoritmo FW sobre arquitecturas multicore. Tomando como base el mencionado trabajo realizado por Costi, se adaptará el código para arquitecturas multicore x86 y se medirá su rendimiento en un servidor de referencia ante diferentes escenarios de prueba. A continuación, se propondrá una posible optimización y se evaluará su impacto para cuantificar sus beneficios.

El resto del informe se organiza de la siguiente forma. La Sección 4 introduce el marco referencial para este trabajo. Luego, la Sección 5 describe la implementación realizada. A continuación, la Sección 6 muestra los resultados experimentales obtenidos mientras que la Sección 7 resume las conclusiones junto al trabajo futuro.

4. Marco referencial

4.1. Programación paralela en memoria compartida

Una arquitectura de memoria compartida se compone de múltiples procesadores y de un espacio de direccionamiento de memoria, que es compartido por todos ellos. Para acceder a la memoria compartida, los procesadores emplean un sistema de interconexión, como puede ser un bus o un conmutador de barras cruzadas (*crossbar*).

Estas arquitecturas se dividen según el tiempo de acceso a la memoria compartida por parte de los procesadores. Si este tiempo es igual para todos los procesadores, se denomina a la arquitectura como UMA (*Uniform Memory Access*). El caso opuesto ocurre en las arquitecturas NUMA (*Non Uniform Memory Access*), que es cuando el tiempo de acceso a la memoria depende de la ubicación del elemento a procesar. Más allá de esto, es usual que cada procesador disponga de una memoria local, denominada caché, para reducir el acceso a la memoria principal [18].

Por otra parte, la programación paralela refiere a la resolución de problemas mediante el uso de dos o más procesadores de manera simultánea. El objetivo de la paralelización es disminuir el tiempo de procesamiento mediante la distribución de las tareas en los procesadores disponibles. En el caso de memoria compartida, la técnica generalmente utilizada es la programación con hilos o threads, realizando sincronización por variables globales.

En la configuración propuesta, existen 2 formas básicas de sincronización: exclusión mutua y sincronización por condición. La primera se relaciona con asegurar que el acceso simultáneo a las secciones de memoria por parte de los procesos no altere el resultado esperado. La segunda tiene que ver con demorar a un proceso para que ejecute una tarea hasta tanto se haya cumplido una determinada condición. Ambos casos se resuelven mediante la definición de secciones críticas a partir del uso de mecanismos de sincronización como *mutex locks*, semáforos y variables condición [19].

4.2. Pthreads

Un hilo es un flujo de control secuencial que se ejecuta en el mismo espacio de memoria que el proceso que lo creó, por lo que puede acceder a variables globales y sus propias variables locales. Su creación y ejecución resultan operaciones más rápidas y livianas que la de procesos.

Pthreads es una librería para el manejo y sincronización de hilos POSIX, un estándar orientado a facilitar la creación de aplicaciones multi-hiladas portables. Por defecto, un proceso se considera como un único hilo principal en ejecución. Entre las funciones brindadas se destacan las siguientes:

- *pthread_create:* permite crear un hilo, especificando la función a ejecutar y argumentos.
- *pthread_join:* el hilo llamador se bloquea hasta la terminación del hilo indicado como argumento.

La biblioteca también provee funciones relacionadas a mecanismos de sincronización, específicamente de mutex locks y variables condición.

En el primer caso, una vez inicializada, se puede realizar *lock* y *unlock* sobre la variable mutex para definir el inicio y fin de una sección crítica. Cuando un hilo llama a la función *pthread_mutex_lock* se pausa el flujo secuencial del hilo y pasa a competir por la exclusión mutua. Una vez conseguido, procede a ejecutar la sección crítica. Finalmente, mediante *pthread_mutex_unlock* se libera el acceso de manera no bloqueante. Este método está orientado a la sincronización por exclusión mutua. Aunque también puede emplearse para sincronización por condición, presenta el inconveniente de la espera activa (*busy waiting*): los hilos en competición se encuentran en un bucle que implica cómputo no productivo para la resolución del problema (overhead).

El segundo mecanismo representa una técnica más eficiente para esta clase de sincronización. La variable condición representa una cola donde los hilos pueden pasar a un estado bloqueado mediante la invocación de *pthread_cond_wait(variable)* hasta que otro hilo produzca una señal sobre dicha variable. Para esto último existen 2 variantes: *pthread_cond_signal* y *pthread_cond_broadcast*, según se requiera despertar únicamente al primer hilo de la cola o a todos, respectivamente. De esta manera, se elimina la presencia de busy waiting.

4.2.1. OpenMP

OpenMP es una librería basada en Pthreads que permite la creación de hilos y su sincronización siguiendo una estrategia de desarrollo incremental [20]. Mediante el agregado de directivas a un código existente, a modo de prefijos, se permite indicar un tipo de comportamiento a aplicar sobre el bloque de código en cuestión, evitando así que el programador deba reescribir gran parte del algoritmo para su paralelización.

Utiliza un modelo de programación Fork-Join, donde el proceso cuenta inicialmente con un único hilo *máster* hasta la aparición de un constructor paralelo, donde se generan los nuevos hilos esclavos (fork), que se ejecutan en paralelo y se sincronizan al final de dicho constructor (join), continuando la ejecución del master.

Una directiva presenta el siguiente formato:

```
#pragma omp nombre-directiva [cláusulas] { ... }
```

El constructor paralelo se declara mediante la directiva *parallel*. El código entre llaves se lo denomina *región paralela*. Algunas de las cláusulas admitidas son:

- private(*lista variables*): establece que se debe crear una copia local de las variables indicadas para cada hilo, sin inicializarlas, al momento de la creación.
- firstprivate(*lista variables*): se debe crear una copia de las variables indicadas y deben estar inicializadas con el valor de la variable original, previo al constructor.
- shared(*lista variables*): asigna las variables indicadas como compartidas para los hilos.
- default: indica la visibilidad o alcance predeterminado de las variables. Si se requiere indicar explícitamente el caso de cada variable se establece *default(none)*.
- num_threads: indica explícitamente el número de hilos a crear en este constructor.

Por otra parte también se dispone del constructor *for*, declarado mediante la directiva homónima. Este permite repartir las iteraciones de un bucle entre los hilos que ya existen en una región paralela. Además de las cláusulas de alcance de variables, también admite:

- schedule: indica la forma en que se dividen las iteraciones del bucle, pudiendo ser *static* (asignación estática de un tamaño dado a cada hilo previo a la ejecución del bucle), *dynamic* (asignación de las iteraciones bajo demanda durante la ejecución del bucle), *guided* (similar dynamic pero decrementando la cantidad asignada a medida que avanza el bucle), y *runtime* (según variable de entorno OMP_SCHEDULE).
- nowait: elimina la barrera implícita de sincronización al final del bucle.
- collapse: indica cuántos bucles anidados deben tratarse como uno solo unificado.

4.3. ALGORITMO FW

El algoritmo de Floyd-Warshall tiene como objetivo la búsqueda del camino mínimo entre cada par de los **N** vértices de un grafo dirigido, también denominado digrafo. Tiene como resultado dos matrices de tamaño NxN, 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. La inexistencia de un camino se representa con valor ∞.
- 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.

Inicialmente, la matriz D se completa con las distancias a los vértices vecinos inmediatos, y la matriz P con el valor i en cada fila i, ya que todos los caminos son directos. Luego, 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 $P_{i,j} = k$.

Como se puede observar en el Algoritmo 1, el procedimiento es realizable mediante el anidamiento de 3 bucles, donde el más externo varía el vértice intermedio a tomar, y los internos a los vértices origen y destino. Los resultados se obtienen con una única ejecución del triple bucle anidado.

Algoritmo 1 Pseudocódigo del algoritmo FW clásico

```
for k ← 0 to N − 1 do
  for i ← 0 to N − 1 do
    for j ← 0 to N − 1 do
      if D_{i,j} ≥ D_{i,k} + D_{k,j} then
        D_{i,j} ← D_{i,k} + D_{k,j}
        P_{i,j} ← k
      end if
    end for
  end for
end for
```

El tiempo de ejecución del algoritmo es del orden $O(N^3)$, mientras que el espacio en memoria requerido es del orden $O(N^2)$.

Para paralelizar el algoritmo y así disminuir el tiempo de ejecución considerablemente, se analizan las dependencias que existen para cada iteración. En la Figura 4.1 se muestra el caso de la primera iteración del bucle externo ($k = 0$) para un grafo de $N = 4$ vértices. En amarillo se indica la celda $D_{i,j}$ en procesamiento, y con una **X** las celdas que se consultan para calcular el costo del camino alternativo $D_{i,k} + D_{k,j}$. Las celdas ya procesadas se indican en color gris.

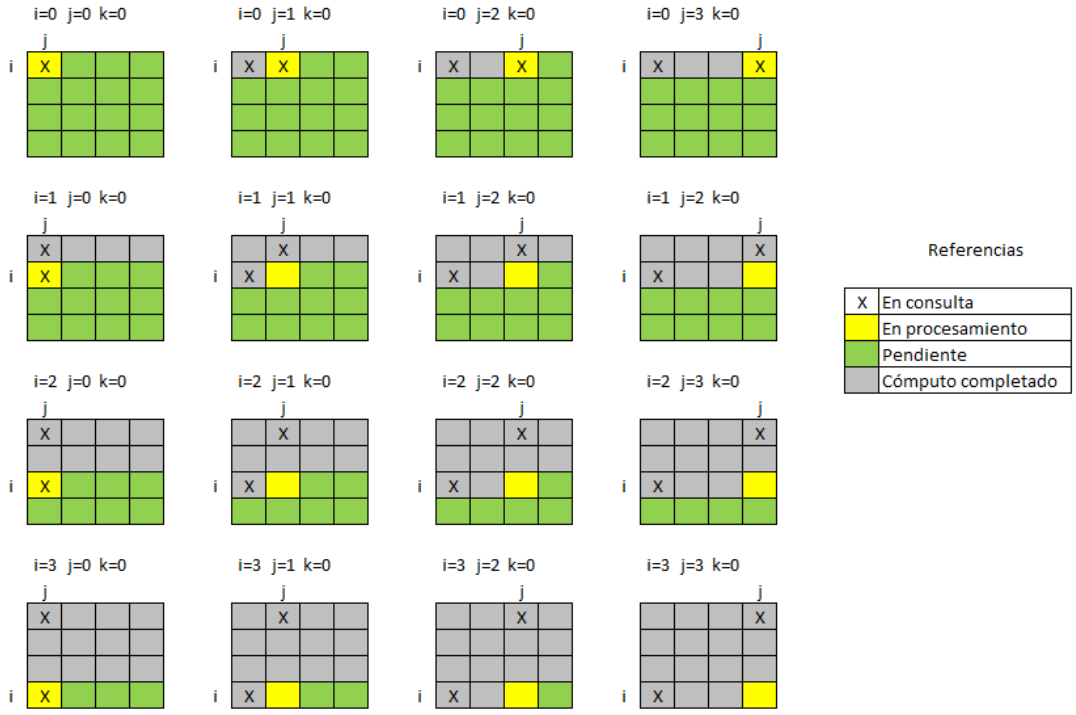

Figura 4.1: Evolución de las dependencias para $k = 0$ para un grafo de $N = 4$ vértices

De esta manera, se observa un patrón particular dentro de una misma iteración de k, en adelante, *ronda*.

1. La celda $D_{i,j}$ donde $i = j = k$ solo depende de si misma, es decir, no requiere que otras celdas de la misma ronda estén computadas. En este caso, $D_{i,j} = D_{i,k} = D_{k,j} = D_{k,k}$
2. Las celdas $D_{i,j}$ donde $i = k$ dependen de la celda $D_{k,k}$ procesada en el ítem 1.
3. Las celdas $D_{i,j}$ donde $j = k$ dependen de la celda $D_{k,k}$ procesada en el ítem 1.
4. El resto de las celdas $D_{i,j}$, donde i y j son distintos a k, tienen múltiples dependencias: $D_{k,k}$, $D_{k,j}$ y $D_{i,k}$, siendo celdas procesadas en los ítem 1, 2 y 3 respectivamente.

A cada ítem se lo puede considerar como *fase* de la ronda. De este modo, en cada ronda, la fase 1 debe ejecutarse primero, luego las fases 2 y 3 en paralelo, y por último la fase 4.

A estas instancias, con una fase 1 compuesta por una única celda cuya distancia $D_{k,k}$ siempre es cero pareciese que no tiene importancia y hasta puede omitirse, pero la lógica de las 4 fases cobra una mayor relevancia en la próxima optimización.

Una problemática presente en este algoritmo es que en grafos con muchos vértices (N grande) habrá una escasa explotación de la *localidad de datos* [21]. Las celdas consultadas pueden encontrarse muy distantes entre sí en memoria, produciendo sucesivos reemplazos de bloques en caché.

4.3.1. Optimización por bloques

La propuesta para aumentar la tasa de aciertos de caché en los procesadores es utilizar la técnica de *blocking* [8, 9]. Desafortunadamente, los tres bucles no pueden ser intercambiados libremente debido a las dependencias de datos entre una iteración y la siguiente en el bucle de k, es decir, únicamente los bucles de i y j pueden resolverse en cualquier orden. De todas maneras, bajo ciertas condiciones, el bucle de k puede insertarse dentro del de i y j, haciendo posible el uso de bloques (blocking).

La optimización por bloques de FW consiste en dividir la matriz de distancias D en bloques de un tamaño **BSxBS** (*Block Size*) determinado, que puedan caber en las memorias caché. De esta manera, se organiza el cómputo de manera que se aproveche la localidad temporal y espacial de datos.

Luego, las rondas de N^2 iteraciones se reducen a $(N/BS)^2$, que es la cantidad de bloques cuadrados de tamaño $BSxBS$ presentes en la matriz D, siendo BS un divisor de N. Por su parte, la cantidad de rondas también decrementa de N a $\mathbf{R} = N/BS$.

Se continúa aplicando la lógica de las fases según las dependencias de datos. La fase 1 consta de un único bloque, cuyo cómputo es paralelizable al constituirse de $BSxBS$ celdas independientes. Luego, en la fase 2 se procesan los bloques de la misma fila; y en la fase 3 aquellos que se encuentran en la misma columna. Por último, en la fase 4 se resuelven el resto de bloques. Una representación visual de la secuencia descrita se presenta en la figura 4.2, donde para cada fase, los bloques computados en la misma se muestran en amarillo, los pendientes en verde y aquellos ya procesados en gris.

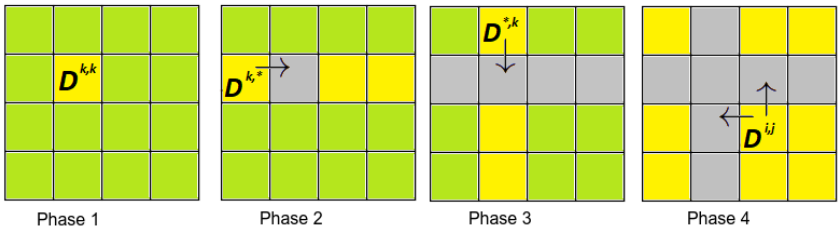

Figura 4.2: Fases y dependencias en una ronda de FW con blocking

La resolución de un bloque se lleva a cabo mediante la función **FW_BLOCK**. Esta recibe los 3 bloques a consultar de la matriz D para comparar las distancias: D^1, D^2 y D^3, que corresponden a $D_{i,j}$, $D_{i,k}$ y $D_{k,j}$, respectivamente. El procedimiento es la adaptación del Algoritmo 1 para bloques de tamaño $BS \times BS$, como se muestra en el Algoritmo 2.

Algoritmo 2 Pseudocódigo de la función FW_BLOCK

```
for k ← 0 to BS − 1 do
  for i ← 0 to BS − 1 do
    for j ← 0 to BS − 1 do
      if D1_{i,j} ≥ D2_{i,k} + D3_{k,j} then
        D1_{i,j} ← D2_{i,k} + D3_{k,j}
        P_{i,j} ← base + k
      end if
    end for
  end for
end for
```

Algoritmo 3 Pseudocódigo del algoritmo FW con blocking

```
for k ← 0 to R − 1 do
  b ← k × BS

  // Fase 1
  FW_BLOCK(D_{k,k}, D_{k,k}, D_{k,k}, P_{k,k}, b)

  // Fase 2
  for j ← 0 to R − 1 do
    if j ≠ k then
      FW_BLOCK(D_{k,j}, D_{k,k}, D_{k,j}, P_{k,j}, b)
    end if
  end for

  // Fase 3
  for i ← 0 to R − 1 do
    if i ≠ k then
      FW_BLOCK(D_{i,k}, D_{i,k}, D_{k,k}, P_{i,k}, b)
    end if
  end for

  // Fase 4
  for i ← 0 to R − 1 do
    for j ← 0 to R − 1 do
      if i, j ≠ k then
        FW_BLOCK(D_{i,j}, D_{i,k}, D_{k,j}, P_{i,j}, b)
      end if
    end for
  end for
end for
```

Por otra parte, el programa principal que resuelve por completo las matrices D y P se puede observar en el Algoritmo 3. La lógica empleada es idéntica para cada una de las R rondas. La función FW_BLOCK es invocada en las 4 fases con los parámetros correspondientes.

Nótese que los bloques de una ronda no son procesados más de 1 vez. De esta manera, $D_{k,k}$ (computado en fase 1) se omite para las fases 2 y 3, que resuelven $R-1$ bloques; y además los procesados por estas fases se omiten en la fase 4, que computa $(R-1)^2$ bloques.

4.4. Código de base

Como código de base se empleó el de [17], el cual fue desarrollado específicamente para procesadores de la arquitectura Xeon Phi Knights Landing de Intel [22]. Para la implementación final del algoritmo FW con blocking en esta arquitectura paralela se realizaron diversos análisis.

Las mejoras encontradas se fueron agregando a modo de incrementos con la nomenclatura *Opt-X*, los cuales se describen a continuación.

- **Opt-0: Granularidad utilizada**. Se elige el *loop* a paralelizar, dando lugar a una paralelización intra-bloque (de grano fino) o inter-bloque (de grano grueso).
  - La primera opción es implementada en la fase 1, con la creación de una función ***FW_BLOCK_PARALLEL***, basada en FW_BLOCK, ya que computa un único bloque. El pseudocódigo se muestra en el Algoritmo 4, cuya una diferencia respecto al Algoritmo 2 es la directiva de OMP agregada para paralelizar las filas del bloque.
  - La paralelización inter-bloque se utiliza en el resto de fases, presentado en el Algoritmo 5. Las fases 2 y 3, al ser independientes entre sí, se pueden ejecutar en paralelo, por lo que se usa un bucle unificado que incluye a ambas. Mediante esta implementación, si R es menor a la cantidad de hilos T establecida, aquellos hilos ociosos de la fase 2 pasan a ejecutar directamente la fase 3. Por otra parte, en la fase 4 se especifica la cláusula *collapse(2)* con el objetivo de paralelizar los dos bucles anidados como si fuese uno solo, bajo el mismo objetivo de evitar ocio.

Algoritmo 4 Paralelización intra-bloque implementada con OpenMP

```
for k ← 0 to BS − 1 do
  #pragma omp for
  for i ← 0 to BS − 1 do
    for j ← 0 to BS − 1 do
      if D1_{i,j} ≥ D2_{i,k} + D3_{k,j} then
        D1_{i,j} ← D2_{i,k} + D3_{k,j}
        P_{i,j} ← base + k
      end if
    end for
  end for
end for
```

Algoritmo 5 Paralelización inter-bloque implementada con OpenMP

```
for k ← 0 to R − 1 do
  b ← k × BS
  // Fase 1
  FW_BLOCK_PARALLEL(D_{k,k}, D_{k,k}, D_{k,k}, P_{k,k}, b)

  // Fases 2 y 3
  #pragma omp for schedule(dynamic)
  for w ← 0 to 2 × R − 1 do
    if w < r then
      j ← w
      if j ≠ k then
        FW_BLOCK(D_{k,j}, D_{k,k}, D_{k,j}, P_{k,j}, b)
      end if
    else
      i ← w − r
      if i ≠ k then
        FW_BLOCK(D_{i,k}, D_{i,k}, D_{k,k}, P_{i,k}, b)
      end if
    end if
  end for

  // Fase 4
  #pragma omp for collapse(2) schedule(dynamic)
  for i ← 0 to R − 1 do
    for j ← 0 to R − 1 do
      if i, j ≠ k then
        FW_BLOCK(D_{i,j}, D_{i,k}, D_{k,j}, P_{i,j}, b)
      end if
    end for
  end for
end for
```

- **Opt-1: MCDRAM**. Es una memoria adicional presente en el procesador Xeon Phi KNL, caracterizada por brindar un ancho de banda alto (hasta 450 GB/s). Sin embargo, tiene una menor capacidad (16 GB) respecto a la DDR (384 GB), por tal motivo se empleó el comando ***numactl -p*** para permitir la asignación auxiliar en DDR. Es importante aclarar que no implica cambios en el código fuente.

- **Opt-2: Vectorización guiada por SSE**. Mediante la directiva ***simd*** de OpenMP 4.0, se indica al compilador que fuerce la vectorización de un bucle. Se aplicó al bucle más interno de *FW_BLOCK* y *FW_BLOCK_PARALLEL*. De esta manera, se utilizan conjuntos de instrucciones SSE de 128 bits, permitiendo procesar hasta 2 datos tipo *double* o 4 de tipo *float*.

- **Opt-3: Vectorización guiada por AVX2**. Usando el mismo código, se le indica al compilador el parámetro *-xAVX2* para que se utilice instrucciones AVX de 256 bits, lo que permite duplicar la cantidad de operaciones simultáneas respecto a SSE (siempre y cuando soporte esta clase de instrucciones).

- **Opt-4: Vectorización guiada por AVX512**. Ídem anterior, pero el parámetro indicado es *-xMIC-AVX512*, para emplear instrucciones de 512 bits. De esta forma, se pueden procesar hasta 8 datos tipo *double* u 16 tipo *float* en simultáneo en un procesador (siempre y cuando soporte esta clase de instrucciones).

- **Opt-5: Alineación de datos**. Se utilizó la función ***_mm_malloc()*** de Intel, similar a *malloc()* de *glibc*, con la diferencia de que reserva memoria de manera que los datos queden alineados, especificando el tamaño en bytes de la caché de datos L1. Esta optimización apunta a reducir la cantidad de acceso a la caché por datos "desalineados".

- **Opt-6: Predicción de saltos**. Se utilizó la función ***__builtin_expect(exp, c)*** en las frecuentemente accedidas *FW_BLOCK* y *FW_BLOCK_PARALLEL*, para indicar al compilador que el resultado más probable (*c*) en el *IF* de comparación (*exp*) es *false*. De esta manera, se reduce la cantidad de errores en la predicción de saltos.

- **Opt-7: Desenrollado guiado de bucles**. Se implementó para las mismas funciones que *Opt-6* mediante la directiva *unroll(FD)*. La cantidad de iteraciones del bucle más externo era BS, pero luego de la vectorización es *FD = BS / (SIMD_WIDTH / TYPE_SIZE)*.

- **Opt-8: Afinidad de hilos con núcleos**. Se puede indicar cómo realizar la distribución de hilos entre los núcleos asignando un valor a la variable de entorno *KMP_AFFINITY*, previo a la ejecución del algoritmo FW. Se especifica un tipo de afinidad, que puede ser *balanced*, *compact* o *scatter*, y la granularidad: *fine*, *core* o *tile*. Para el procesador Intel Xeon Phi KNL, la configuración óptima fue la afinidad *balanced* con granularidad *fine*.

El Algoritmo 6 muestra el estado final de la función *FW_BLOCK_PARALLEL* luego de las optimizaciones incrementales mencionadas anteriormente. En el caso de la función *FW_BLOCK*, el algoritmo es el mismo, salvo la directiva *for* de OpenMP, que no está presente.

Algoritmo 6 Paralelización intra-bloque hasta Opt-8

```
FD = BS / (SIMD_WIDTH / TYPE_SIZE)

for k ← 0 to BS − 1 do
  // Desenrollado manual de 2 iteraciones de i
  #pragma omp for
  for i ← 0 to (BS − 1) step 2 do
    // Desenrollado de bucles y vectorización (Opt-7 y Opt-2)
    #pragma unroll(FD)
    #pragma omp simd
    for j ← 0 to BS − 1 do
      // Predicción de saltos (Opt-6)
      if __builtin_expect(D1_{i,j} ≥ D2_{i,k} + D3_{k,j}, false) then
        D1_{i,j} ← D2_{i,k} + D3_{k,j}
        P_{i,j} ← base + k
      end if
    end for

    #pragma unroll(FD)
    #pragma omp simd
    for j ← 0 to BS − 1 do
      if __builtin_expect(D1_{i+1,j} ≥ D2_{i+1,k} + D3_{k,j} , false) then
        D1_{i+1,j} ← D2_{i+1,k} + D3_{k,j}
        P_{i,j} ← base + k
      end if
    end for
  end for
end for
```

5. IMPLEMENTACIÓN

5.1. ADAPTACIÓN DE CÓDIGO BASE A ENTORNO EXPERIMENTAL

El código de base explicado en la sección 4, cuyas optimizaciones se testearon en un procesador Intel Xeon Phi de segunda generación (KNL), es adaptado en este trabajo para su ejecución en un servidor constituido por dos procesadores Intel Xeon Platinum 8276L en Dual-Socket. Cada procesador posee 28 cores de frecuencia base 2.20 GHz, y cuenta con tecnología Hyper-Threading de modo que cada core puede ejecutar hasta 2 hilos hardware, resultando un total de 112 hilos disponibles en el servidor. Respecto al conjunto de instrucciones, el procesador provee soporte de vectorización para Intel SSE 4.2, AVX2 y AVX-512 [23]. Por último, el compilador utilizado es Intel ICC, parte de la suite oneAPI versión 2021.7.1.

El código fuente de C de cada versión incremental *Opt-X* se encuentran bajo un mismo directorio *floyd_versions*. A continuación, se detallan las modificaciones realizadas.

- **Opt-0:** no se realizaron cambios en las líneas de código referidas al tipo de paralelización, es decir, las directivas *for* de OpenMP, como tampoco se alteró el orden de las fases.
- **Opt-1:** esta optimización referida al uso de la memoria MCDRAM directamente es descartada, debido a que el procesador Intel Xeon Platinum no cuenta con este tipo de memoria adicional a la DDR. Por tal motivo, al momento de ejecutar las pruebas de las siguientes versiones, se omite el parámetro *numactl -p*.
- **Opt-2 a Opt-4:** la vectorización guiada por SSE, mediante directiva *simd* de OpenMP permanece sin cambios, de igual manera que el flag *-xAVX2* para *Opt-3*. En contraparte, el flag *-xMIC-AVX512* es reemplazado por *-xCORE-AVX512*, recomendado para Intel Xeon Platinum. Nuevamente, AVX512 es la máxima vectorización disponible.
- **Opt-5:** se mantienen los archivos de código común para reserva de memoria, llamados *aligned.c* y *no_align.c*, ubicados en el subdirectorio *malloc*. El primer archivo utiliza la función *_mm_malloc()*, para lo cual se establece el parámetro **MEM_ALIGN = SIMD_WIDTH / 8**, expresado en bytes, donde SIMD_WIDTH = 512. La versión *no_align.c* utiliza el tradicional *malloc()*.
- **Opt-6:** se mantienen las macros *likely(exp)* y *unlikely(exp)*, definidas en el archivo de código del algoritmo FW. Ambas utilizan la función *__builtin_expect(exp, c)* de Intel, pero difieren en el valor de predicción *c*, siendo *true* en *likely*, y *false* en *unlikely*.
- **Opt-7:** el desenrollado guiado de bucles requiere de la especificación del parámetro *FD* en la directiva *unroll*. Para ello, se reutiliza el parámetro *SIMD_WIDTH*, y se especifica *TYPE_SIZE* al compilar, cuyo valor en bits depende del tipo de dato utilizado para el grafo: 32 para simple precisión (*float*) y 64 para doble precisión (*double*).
- **Opt-8:** se prueban las distintas afinidades y granularidades para elegir la más óptima. Se encontró que la mejor configuración coincide con la utilizada para el procesador Intel Phi Xeon KNL: granularidad *fine* (cada hilo de OpenMP está relacionado a un único hilo hardware) y afinidad *balanced* (se asigna un hilo a cada core y se ubican los restantes con los de ID cercano). Resulta entonces *KMP_AFFINITY=granularity=fine,balanced*.

5.2. Optimización intra-ronda

Desde la optimización *Opt-2* hasta *Opt-8*, se aplicaron mejoras al algoritmo FW con blocking que no alteran las condiciones establecidas en *Opt-0* de inicio y fin de cada fase. En cada ronda, existe una paralelización intra-bloque e inter-bloque. La primera se encuentra aplicada en la función *FW_BLOCK_PARALLEL* de la fase 1, de modo que los hilos se distribuyen el procesamiento del único bloque; y la segunda en los bucles de las fases restantes, distribuyéndose cada bloque a computar mediante la función secuencial *FW_BLOCK*.

Por defecto, al aplicar una paralelización mediante directiva *for* existe una barrera implícita hasta que todas las iteraciones se hayan realizado. Esto permite que una fase no comience antes de que se procese por completo la fase anterior, respetando las dependencias:

- El bloque $D_{k,k}$ de la fase 1, una vez procesado, es consultado en cada una de las iteraciones de las fases 2 y 3, ya que el bloque a actualizar se encuentra en la k-ésima fila y k-ésima columna, respectivamente. Por tal motivo, la sincronización al término de la fase 1 es obligatoria.
- Sin embargo, el escenario es distinto para la fase 4. Los bloques $D_{i,j}$ a computar en esta etapa dependen siempre de 2 bloques computados anteriormente: uno en fase 2 ($D_{k,j}$) y otro en fase 3 ($D_{i,k}$). Como ambas fases procesan otros bloques además de los dos indicados, es posible evitar la sincronización previa al inicio de la fase 4. La barrera se elimina agregando la cláusula ***nowait*** a la directiva *for* de la fase 2-3.

Para garantizar un resultado final correcto, se debe asegurar que las dependencias para cada bloque de la fase 4 sean satisfechas. Es por lo que resulta necesario utilizar mecanismos de sincronización de un grano más fino, al tratar con bloques en lugar de fases. La Figura 5.1 ejemplifica la oportunidad de optimización, donde los bloques ya computados se muestran en color gris (5 de la fase 2-3), y aquellos en procesamiento en amarillo (6 de la fase 4).

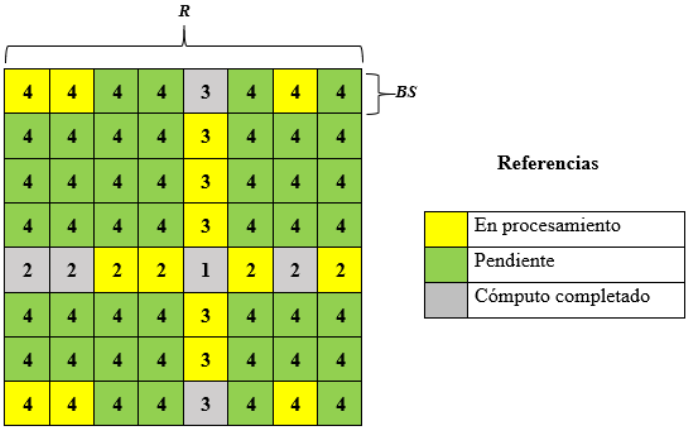

Figura 5.1: Ejemplo de posible escenario para $R = 8$ en la ronda $k = 4$

En las siguientes subsecciones se presentan dos soluciones a evaluar según su rendimiento.

5.2.1. Implementación con semáforos

Se añade una matriz de semáforos de dimensión $R \times R$, inicializados en cero. Cada celda representa a un bloque de la matriz D. Esta técnica permite 2 alternativas:

- **Consulta en dependencias**: para computar un bloque de fase 4, se realiza una operación **P** en cada una de las posiciones de los bloques que depende. Las operaciones **V** se realizan en la fase 2-3 sobre la misma posición del bloque que acaba de procesarse. La desventaja de este método es la competencia entre bloques de fase 4 de una misma fila o columna, que comparten dependencia, al realizar **P** en la misma posición.
- **Consulta *in-situ***: la habilitación para operar sobre un bloque de fase 4 está condicionada por n operaciones **P** en la posición correspondiente a dicho bloque, donde n es el número de dependencias que posee, en este caso 2. Dichas dependencias son las responsables de realizar las operaciones **V** correspondientes. De esta manera, se elimina la competencia dada en el caso anterior, por lo que se optó este método.

Cuando se completa el procesamiento de un bloque de fase 2, se realiza un **V** sobre cada semáforo de su misma columna j, excepto su propia posición. De igual modo, al terminar de procesarse un bloque de fase 3, se realiza un **V** sobre los otros semáforos de su fila i. Luego, para computar un bloque $D_{i,j}$ de la fase 4, se realizan dos **P** en su propia posición (i, j).

Al completarse una ronda todos los semáforos se encuentran nuevamente en cero. En la Figura 5.2 se muestra el valor de los semáforos previo al **P** por parte de la fase 4.

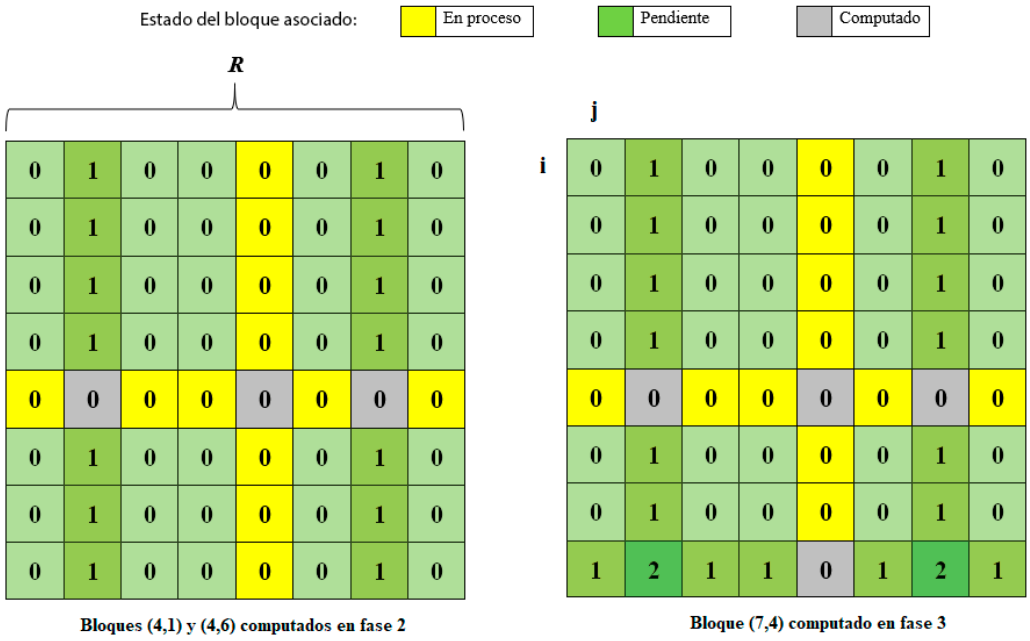

Figura 5.2: Ejemplo de semáforos decrementados durante la fase 2-3

Para el uso de semáforos en C se importa la biblioteca *semaphore.h*, que provee las funciones *sem_init()* para la inicialización del semáforo en un valor determinado, *sem_post()* para la operación **V**, y *sem_wait()* para la operación **P**.

El Algoritmo 7 presenta las modificaciones aplicadas en las rondas, indicadas en color azul, donde S es la matriz de semáforos. No se muestra la inicialización de la misma.

Algoritmo 7 Ejecución temprana de fase 4 mediante semáforos

```
for k ← 0 to R − 1 do
  b ← k × BS
  FW_BLOCK_PARALLEL(D_{k,k}, D_{k,k}, D_{k,k}, P_{k,k}, b)

  #pragma omp for schedule(dynamic) nowait
  for w ← 0 to 2 × R − 1 do
    if w < r then
      j ← w
      if (j == k) continue
      FW_BLOCK(D_{k,j}, D_{k,k}, D_{k,j}, P_{k,j}, b)
      for i ← 0 to R − 1 do
        if (i == k) continue
        V(S_{i,j})
      end for
    else
      i ← w − r
      if (i == k) continue
      FW_BLOCK(D_{i,k}, D_{i,k}, D_{k,k}, P_{i,k}, b)
      for j ← 0 to R − 1 do
        if (j == k) continue
        V(S_{i,j})
      end for
    end if
  end for

  #pragma omp for collapse(2) schedule(dynamic)
  for i ← 0 to R − 1 do
    for j ← 0 to R − 1 do
      if (i == k) or (j == k) continue
      P(S_{i,j})
      P(S_{i,j})
      FW_BLOCK(D_{i,j}, D_{i,k}, D_{k,j}, P_{i,j}, b)
    end for
  end for
end for
```

5.2.2. Implementación con variables condición

Se adiciona una matriz ***CV*** de variables condición, una matriz ***M*** de variables mutex y una matriz ***F***, todas de dimensión $R \times R$. Cada celda $F_{i,j}$ indica la cantidad de procesamientos que restan completarse para habilitar el cómputo de un bloque $D_{i,j}$, situado en fase 4.

Previo a la ejecución de la primera ronda, se inicializa toda la matriz F con el valor **2**, ya que los bloques de fase 4 poseen dos dependencias. Dicho valor se reasigna en cada ronda.

Cuando se finaliza el cómputo de un bloque de fase 2, primero se decrementan en uno las restantes $R-1$ posiciones de la matriz F que se encuentran en la misma columna j, con exclusión mutua, y luego se realiza un *signal* en las variables condición asociadas.

De manera análoga, luego de procesar un bloque de fase 3 se sustrae una unidad a las posiciones de la misma columna i en F, nuevamente con exclusión mutua y realizando un *signal* en las variables condición correspondientes. Resulta entonces F un contador opuesto al interno de los semáforos presentado anteriormente en la Figura 5.2, como se puede observar en la Figura 5.3, que ilustra el resultado con los mismos bloques procesados.

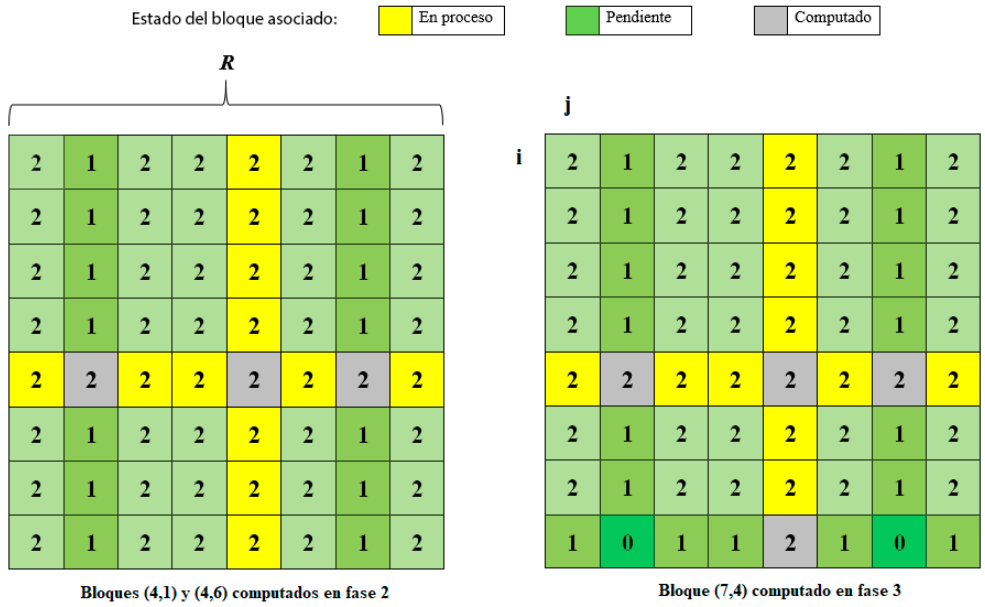

Figura 5.3: Ejemplo de valores en la matriz F durante la fase 2-3

La matriz F debe existir necesariamente en esta solución. Cuando se realiza un *signal* sobre una variable condición, este despertará al primer hilo de la cola de espera **en caso de existir**, sino no tendrá efecto alguno, es decir, no es acumulativo, a diferencia de la operación V de semáforos. Esto implica que la operación *wait* siempre produce una suspensión de la tarea, por lo que sólo debe ser invocado cuando existan dependencias por resolver. Adicionalmente, la actualización de F debe estar contenida en una sección crítica, debido a los conflictos de modificación entre bloques de la fase 2 y 3 (misma celda a escribir).

El Algoritmo 8 presenta las modificaciones realizadas para las fases 2-3, en color azul.

Algoritmo 8 Fase 2-3 de la optimización con variables condición

```
#pragma omp for schedule(dynamic) nowait
for w ← 0 to 2 × R − 1 do
  if w < r then
    j ← w
    if (j == k) continue
    FW_BLOCK(D_{k,j}, D_{k,k}, D_{k,j}, P_{k,j}, b)
    for i ← 0 to R − 1 do
      if (i == k) continue
      pthread_mutex_lock(M_{i,j})
      F_{i,j} ← F_{i,j} − 1
      pthread_cond_signal(CV_{i,j})
      pthread_mutex_unlock(M_{i,j})
    end for
  else
    i ← w − r
    if (i == k) continue
    FW_BLOCK(D_{i,k}, D_{i,k}, D_{k,k}, P_{i,k}, b)
    for j ← 0 to R − 1 do
      if (j == k) continue
      pthread_mutex_lock(M_{i,j})
      F_{i,j} ← F_{i,j} − 1
      pthread_cond_signal(CV_{i,j})
      pthread_mutex_unlock(M_{i,j})
    end for
  end if
end for
```

Por otra parte, para computar un bloque $D_{i,j}$ de fase 4, primero se consulta con exclusión mutua si existen dependencias por resolver ($F_{i,j} > 0$). En caso afirmativo, se realiza un *wait* sobre la variable condición del propio bloque, agregándose a la cola de espera. Esta operación utiliza la variable mutex internamente para la modificación atómica de la propia variable condición, realizando *unlock* luego de que el hilo se haya bloqueado/dormido.

Cuando se realiza un *signal*, si el procesamiento del bloque está suspendido, se realiza un *lock* y se vuelve a consultar por $F_{i,j}$, repitiéndose el proceso hasta que las dependencias estén resueltas, condición en la cual se habilita el procesamiento del bloque actual y se asigna nuevamente el valor **2** a $F_{i,j}$ para la próxima ronda.

El Algoritmo 9 resume el procedimiento explicado para la fase 4. Si se disponen de pocos hilos ($T < 2 \times R$), es más probable que al arribar a esta fase las dependencias estén resueltas ($F_{i,j} = 0$) y ambos *signal* ya hayan sucedido, no debiéndose invocar la operación *wait*. Por el contrario, con muchos hilos puede ocurrir que se arribe al bloque de fase 4 incluso antes del primer *signal*, lo que justifica el uso de una sentencia *while* para la consulta de F.

Algoritmo 9 Fase 4 de la optimización con variables condición

```
#pragma omp for collapse(2) schedule(dynamic)
for i ← 0 to R − 1 do
  for j ← 0 to R − 1 do
    if (i == k) or (j == k) continue

    pthread_mutex_lock(M_{i,j})
    while (F_{i,j} > 0) do
      pthread_cond_wait(F_{i,j}, M_{i,j})
    end while
    pthread_mutex_unlock(M_{i,j})

    FW_BLOCK(D_{i,j}, D_{i,k}, D_{k,j}, P_{i,j}, b)
    F_{i,j} = 2
  end for
end for
```

Un ejemplo de posible secuencia en la ejecución temprana de fase 4 se puede observar en la Figura 5.4. En este caso, la cantidad de hilos T es superior al número de bloques a computar en la fase 2-3, por lo que arriban hilos también a la última fase de ronda. El diagrama a la derecha es la representación simplificada del algoritmo usando variables condición, haciendo énfasis en tres bloques que actualizan o consultan sobre una misma posición de F.

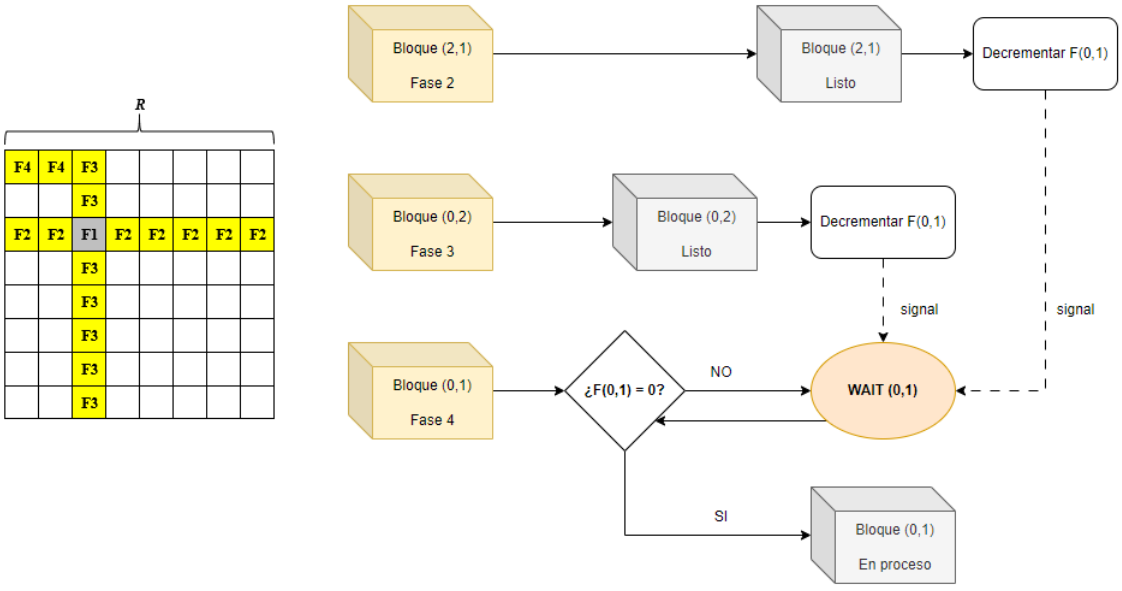

Figura 5.4: Mecanismo de sincronización con variables condición.

6. Resultados experimentales

6.1. Pruebas realizadas

Para comparar la mejora obtenida entre las distintas versiones, se debió medir el tiempo de ejecución del algoritmo FW entre diferentes tamaños de entrada N y de bloque BS.

El programa inicia con la carga del grafo para la creación de las matrices D y P. Se utiliza la misma entrada o *input* para todas las pruebas que se realicen con un determinado N y BS. Se estableció un porcentaje de completitud del 70 % para todos los casos y el tipo de dato (*float* o *double*) es definido en un archivo C al momento de la generación de la entrada.

Luego de establecerse el estado inicial de las matrices, comienza la medición del tiempo de ejecución (t_e) mediante la función *dwalltime()* y se ejecuta el algoritmo FW. Al finalizar el cómputo, se vuelve a invocar a *dwalltime()*, para posteriormente calcular el resultado t_e en segundos y en GFLOPS como $GFLOPS = \frac{2 \times N^3}{t \times 10^9}$.

Mediante un script, para cada combinación de versión, N, T, BS y tipo de dato se realizó un mínimo de 3 pruebas entre *Opt-0* y *Opt-4*, y de 6 a 10 pruebas para las siguientes optimizaciones. Hasta *Opt-7* la variable de entorno *KMP_AFFINITY* permanece sin asignar.

- N: {4096, 8192, 16384}
- BS: {32, 64, 128}
- T: {56, 112}

Por último, todos los códigos fuente se encuentran en un repositorio público, accesible mediante el siguiente enlace: `https://bit.ly/pps-fw-xeonplatinum`.

6.2. Resultados para código base adaptado

Considerando un tamaño de entrada intermedio ($N = 8192$) y tipo *double*, se presentan los resultados obtenidos en la Figura 6.1. Las columnas indican la versión utilizada, y en las filas se agrupa por número de hilos T y luego se indica el tamaño de bloque BS.

Promedio de GFlops	opt_0_1	opt_2	opt_3	opt_4	opt_5	opt_6	opt_7	opt_8
56								
32	64,92	78,24	85,86	173,26	165,83	237,92	261,37	341,27
64	57,49	78,67	115,67	175,85	193,65	269,17	353,84	498,55
128	68,34	70,03	152,69	185,71	205,76	329,13	303,84	484,54
112								
32	80,46	90,96	111,21	153,75	155,56	300,89	318,97	325,94
64	62,98	100,13	108,85	269,86	294,18	356,95	440,25	460,73
128	94,85	87,76	119,81	287,53	342,96	404,38	447,07	473,59

Figura 6.1: GFLOPS promedio hasta Opt-8 para N=8192 y tipo *double*

Se puede observar que, para cada versión, el mejor resultado se puede dar con 56 o 112 hilos, y a su vez, con bloques de 64x64 o 128x128. Esto indica que se deben elegir los valores óptimos

de T y BS de cada *Opt-X* para comparar entre versiones adecuadamente, ya que ciertas combinaciones pueden resultar contraproducentes en algunos casos. Una mejor visualización con el BS óptimo se muestra en las Figuras 6.2 y 6.3, para 56 y 112 hilos respectivamente.

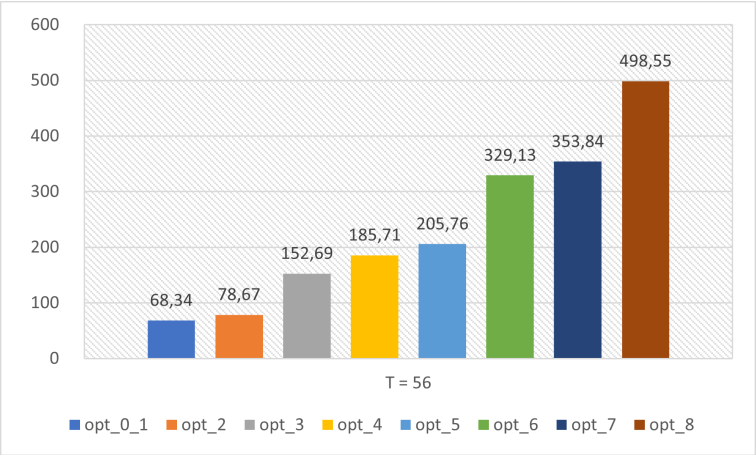

Figura 6.2: GFLOPS con BS óptimo para N=8192, T=56 y tipo *double*

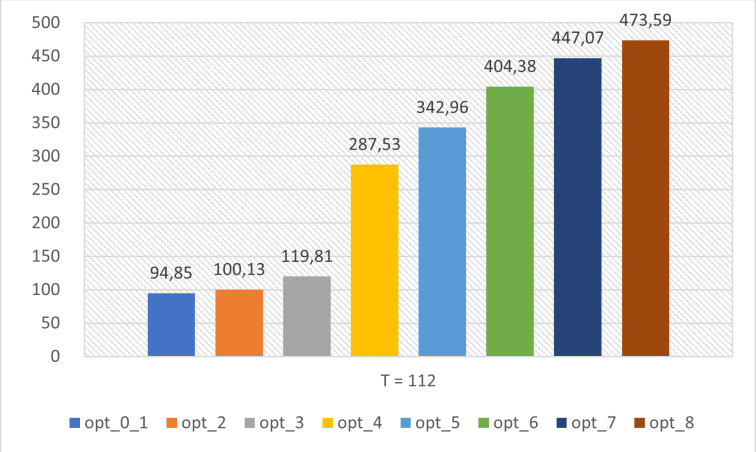

Figura 6.3: GFLOPS con BS óptimo para N=8192, T=112 y tipo *double*

Efectivamente, hay un incremento de los GFLOPS medidos al avanzar de optimización para ambas cantidades de threads. Tomando el promedio, la mejora obtenida en cada *Opt-X* respecto a su anterior y a la primera versión *Opt-0* es la siguiente:

	Opt-2	Opt-3	Opt-4	Opt-5	Opt-6	Opt-7	Opt-8
Respecto a anterior	1.10x	1.57x	1.81x	1.15x	1.39x	1.09x	1.23x
Respecto a Opt-0	1.10x	1.73x	3.13x	3.60x	5.00x	5.45x	6.73x

Tabla 6.1: Mejora obtenida hasta Opt-8 para N=8192 y tipo double

El mayor salto de optimización encontrado según la Tabla 6.1 es **1.81x** y se encuentra en la versión *Opt-4* mediante la vectorización con *AVX-512*. En segundo lugar se encuentra la versión anterior: AVX-2 con una mejora de **1.57x** respecto al anterior. De esta manera, se demuestra que la vectorización permite una aceleración del **3.13x** del algoritmo FW.

Otra versión a destacar es la mejora del **1.39x** en la predicción de saltos de *Opt-6*. Esta mejora se implementa en las funciones FW_BLOCK y FW_BLOCK_PARALLEL que, al ejecutarse sucesivas veces en cada ronda, terminan representando un *hotspot* de la aplicación. Hasta *Opt-8*, se tiene una mejora acumulada de **6.73x**.

Por otra parte, las pruebas realizadas con el tipo de dato *float* se ejecutaron en menor tiempo en todas las versiones, como era de esperar. En la Figura 6.4 se muestran los GFLOPS obtenidos para el tamaño de entrada intermedio ($N = 8192$).

Promedio de GFlops	opt_0_1	opt_2	opt_3	opt_4	opt_5	opt_6	opt_7	opt_8
56								
32	62,03	111,06	51,07	185,12	222,85	346,40	374,78	494,90
64	62,95	81,35	32,37	189,76	230,82	472,82	480,55	710,57
128	87,15	110,83	43,55	215,53	334,13	507,52	559,67	831,70
112								
32	79,23	157,39	224,92	261,30	273,38	422,60	440,06	463,17
64	78,10	101,26	402,67	447,31	491,73	587,05	611,58	664,20
128	124,09	163,40	425,86	489,67	593,25	700,64	766,82	866,31

Figura 6.4: GFLOPS promedio hasta Opt-8 para N=8192 y tipo *float*

Considerando el tamaño *BS* óptimo, se presentan los resultados para 112 hilos en la Figura 6.5, donde se puede observar que la vectorización brinda una mejora total del **3.95x**, y se alcanza un acumulado del **6.98x** hasta *Opt-8*.

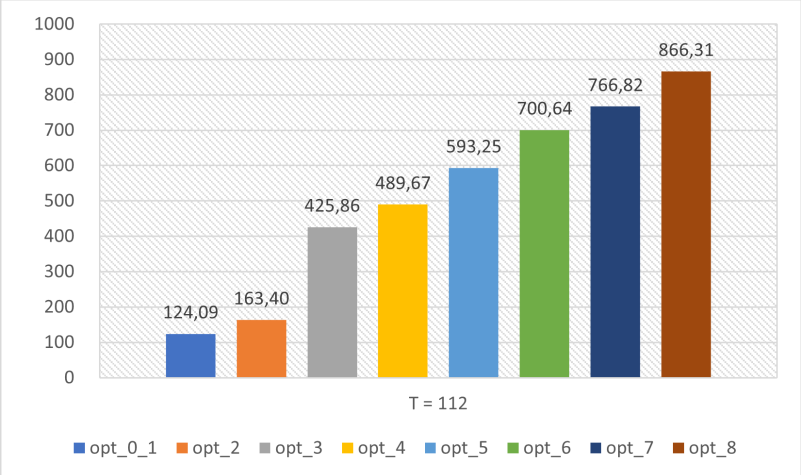

Figura 6.5: GFLOPS con BS óptimo para N=8192, T=112 y tipo *float*

Para la versión *Opt-8* se probaron las 6 combinaciones de granularidad (core, fine) y afinidad (balanced, compact y scatter) [24]. Se encontró que la mejor configuración es **fine-balanced**, por leve diferencia sobre *scatter*, como se observa en la Figura 6.6 para 56 hilos y tipo float. Finalmente, la Tabla 6.2 resume los valores completos de la versión Opt-8.

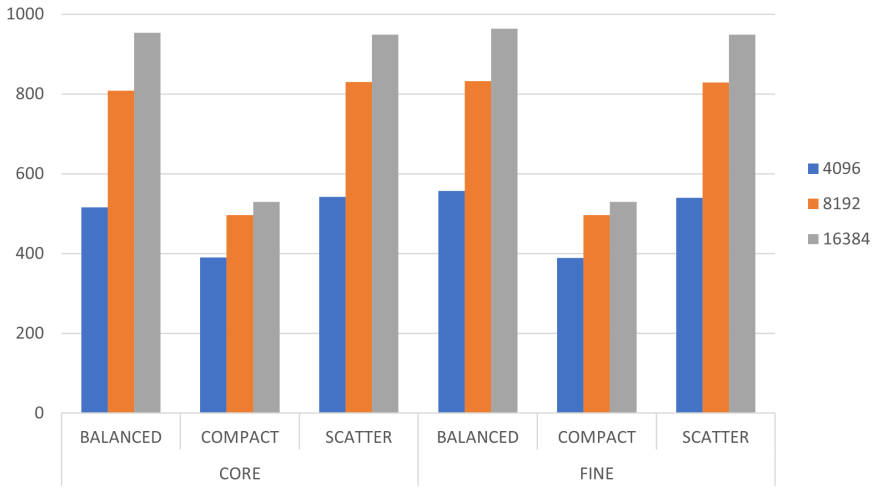

Figura 6.6: GFLOPS para distintas afinidades y granularidades

N	T	BS	GFLOPS (double)	GFLOPS (float)
4096	56	32	279.82	430.17
		64	393.14	542.78
		128	367.28	556.79
4096	112	32	239.68	361.75
		64	332.86	487.93
		128	347.05	578.53
8192	56	32	341.27	494.90
		64	498.55	710.57
		128	484.54	831.70
8192	112	32	325.94	463.17
		64	460.73	664.20
		128	473.59	866.31
16384	56	32	479.45	632.02
		64	587.34	840.99
		128	516.25	963.49
16384	112	32	469.83	626.36
		64	560.44	807.53
		128	514.65	1021.22

Tabla 6.2: Resultados completos de la versión Opt-8

Las dos propuestas para la ejecución temprana de la fase 4 son candidatas a brindar una mejora respecto al código base. Como a priori se desconoce cuál puede aportar mejores resultados, se decidió utilizar el nombre **Opt-9** para ambas versiones, diferenciándose en el sufijo según el mecanismo de sincronización empleado.

- **Opt-9-Sem**: implementación con semáforos y consultas in-situ.
- **Opt-9-Cond**: variante que utiliza variables condición.

Considerando las mejoras aplicadas hasta *Opt-8*, incluida la asignación de la variable de entorno *KMP_AFFINITY* con granularidad fine y afinidad balanced, se realizaron 10 pruebas para cada combinación de tamaño de entrada, de bloque, número de hilos y tipo de dato.

La medición del tiempo de ejecución, mediante *dwalltime()*, ahora también considera la reserva de memoria, inicialización y liberación de las matrices agregadas en *Opt-9*, por tratarse de estructuras adicionales que no forman parte de la carga inicial del grafo. Este nuevo *overhead* puede repercutir de manera contraproducente en ciertas combinaciones. Al igual que en la sección anterior, los valores a testear fueron los siguientes:

- N = {4096, 8192, 16384}
- BS = { 32, 64, 128}
- T = { 56, 112}

N	T	BS	GFLOPS (double)	GFLOPS (float)
4096	56	32	272.76	408.88
		64	397.35	485.91
		128	423.49	581.04
4096	112	32	245.19	357.67
		64	352.83	484.72
		128	402.84	586.41
8192	56	32	317.65	469.88
		64	494.51	639.82
		128	550.83	855.15
8192	112	32	315.18	436.02
		64	466.42	623.87
		128	585.31	905.83
16384	56	32	409.05	582.79
		64	584.32	717.50
		128	593.36	976.98
16384	112	32	414.55	534.50
		64	556.36	706.58
		128	629.38	1034.44

Tabla 6.3: Resultados obtenidos en la versión Opt-9-Sem

N	T	BS	GFLOPS (double)	GFLOPS (float)
4096	56	32	277.13	372.02
		64	398.82	479.04
		128	423.66	585.81
4096	112	32	252.80	350.46
		64	353.72	476.35
		128	387.35	594.38
8192	56	32	333.33	497.63
		64	499.41	738.00
		128	554.76	857.87
8192	112	32	326.49	483.08
		64	468.68	688.01
		128	549.29	909.03
16384	56	32	426.83	631.27
		64	583.84	904.71
		128	590.39	978.05
16384	112	32	429.33	627.63
		64	541.68	828.90
		128	589.54	1038.15

Tabla 6.4: Resultados obtenidos en la versión Opt-9-Cond

Las Tablas 6.3 y 6.4 muestran los resultados obtenidos para las versiones Opt-9-Sem y Opt-9-Cond, respectivamente. Se puede observar en ellas que el tamaño de bloque óptimo se encuentra estabilizado en $BS = 128$, independientemente del tipo de dato utilizado, cantidad de hilos o tamaño de entrada. Esto representa una ventaja frente a las versiones anteriores con BS óptimo variable, lo que dificultaba la elección del tamaño adecuado a cada situación.

La Tabla 6.5 resume los resultados para el tipo de dato *float* utilizando el valor BS óptimo, y se los compara con la versión inicial *Opt-0* y la última del código base, *Opt-8*.

N	T	Opt-0	Opt-8	Opt-9-Sem	Opt-9-Cond
4096	56	72.10	556.79	581.04	585.81
	112	90.28	578.53	586.41	594.38
8192	56	87.15	831.70	855.15	857.87
	112	124.09	866.31	905.83	909.03
16384	56	116.75	963.49	976.98	978.05
	112	133.67	1021.22	1034.44	1038.15

Tabla 6.5: GFLOPS usando BS óptimo hasta Opt-9 para tipo float

De esta manera, se evidencia que existe una mejora en la optimización intra-ronda frente al código base para los tamaños de entrada y números de hilos testeados. La Tabla 6.6 muestra la aceleración obtenida para ambas versiones de Opt-9 en cada combinación de N y T, con tipo de dato float, respecto a la versión *Opt-8*.

N	T	Opt-9-Sem	Opt-9-Cond
4096	56	1.0435x	1.0521x
	112	1.0136x	1.0274x
8192	56	1.0282x	1.0315x
	112	1.0456x	1.0493x
16384	56	1.0140x	1.0151x
	112	1.0129x	1.0166x

Tabla 6.6: Mejora de Opt-9 respecto a su predecesor para tipo float

La implementación con variables condición resulta levemente superior a la versión utilizando semáforos, si se considera el BS óptimo. En términos comparativos, se obtienen mejores resultados en el tamaño de entrada intermedio ($N = 8192$), descritos a continuación.

Para 56 hilos, se obtiene una aceleración de **1.03x** respecto a Opt-8, y la mejora acumulada respecto a Opt-0 pasa de 9.54x a **9.84x** utilizando variables condición. De manera similar, para 112 hilos se consigue una aceleración muy próxima a **1.05x** respecto a la mejor versión del código base, y el acumulado pasa de 6.98x a **7.32x** con la optimización intra-ronda. En las Figuras 6.7 y 6.8 se visualizan estas comparaciones para T=56 y T=112, respectivamente.

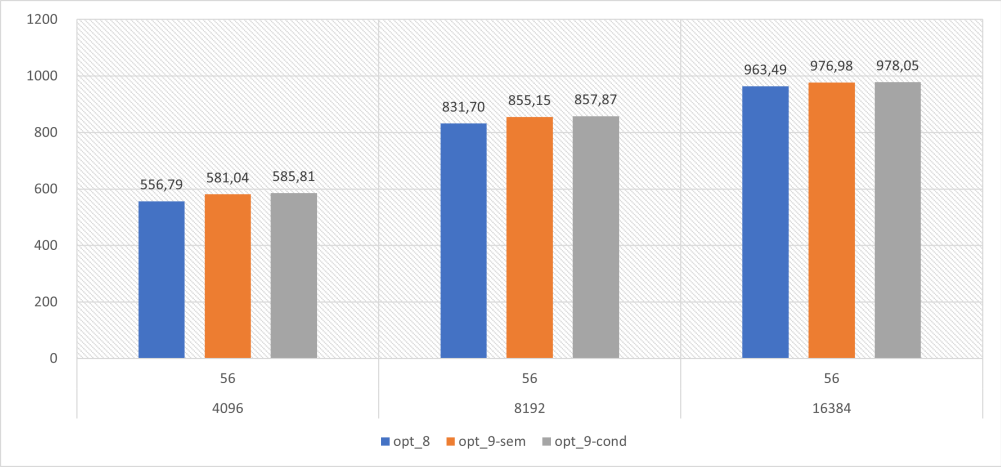

Figura 6.7: GFLOPS con BS óptimo para T=56 y tipo *float*

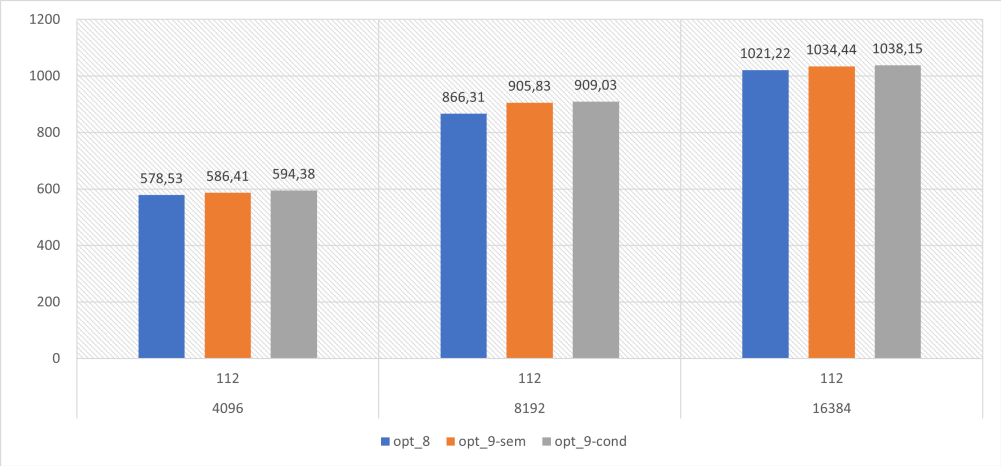

Figura 6.8: GFLOPS con BS óptimo para T=112 y tipo *float*

Por otra parte, para el caso del tipo de dato *double*, también se logra un menor tiempo de ejecución mediante la optimización intra-ronda (Opt-9), eligiendo el tamaño de bloque óptimo $BS = 128$, como se pudo observar en las Tablas 6.3 y 6.4. Un resumen comparativo respecto al código base se presenta en la Tabla 6.7, utilizando siempre el valor BS óptimo que corresponde en cada caso.

N	T	Opt-0	Opt-8	Opt-9-Sem	Opt-9-Cond
4096	56	55.12	393.14	423.49	423.66
	112	81.29	347.05	402.84	387.35
8192	56	68.34	498.55	550.83	554.76
	112	94.85	473.59	585.31	549.29
16384	56	78.06	587.34	593.36	590.39
	112	98.38	560.44	629.38	589.54

Tabla 6.7: GFLOPS usando BS óptimo hasta Opt-9 para tipo double

A diferencia de lo observado para el tipo float, ahora la implementación con semáforos brinda un mejor rendimiento que la versión con variables condición, y además, la mejora alcanza nuevos máximos, como se muestra en la Tabla 6.8.

N	T	Opt-9-Sem	Opt-9-Cond
4096	56	1.0772x	1.0776x
	112	1.1608x	1.1161x
8192	56	1.1049x	1.1127x
	112	1.2359x	1.1598x
16384	56	1.0102x	1.0052x
	112	1.1230x	1.0519x

Tabla 6.8: Mejora de Opt-9 respecto a su predecesor para tipo double

Tomando el caso de $N = 8192$ y $T = 112$, donde el tamaño de bloque óptimo coincide para las 3 versiones comparadas, la mejora obtenida en *Opt-9-Sem* respecto a *Opt-8* es **1.24x**, mientras que la alternativa *Opt-9-Cond* aporta una mejora de **1.16x**.

A modo ilustrativo, se visualizan los GFLOPS medidos para 56 y 112 hilos en las Figuras 6.9 y 6.10, respectivamente.

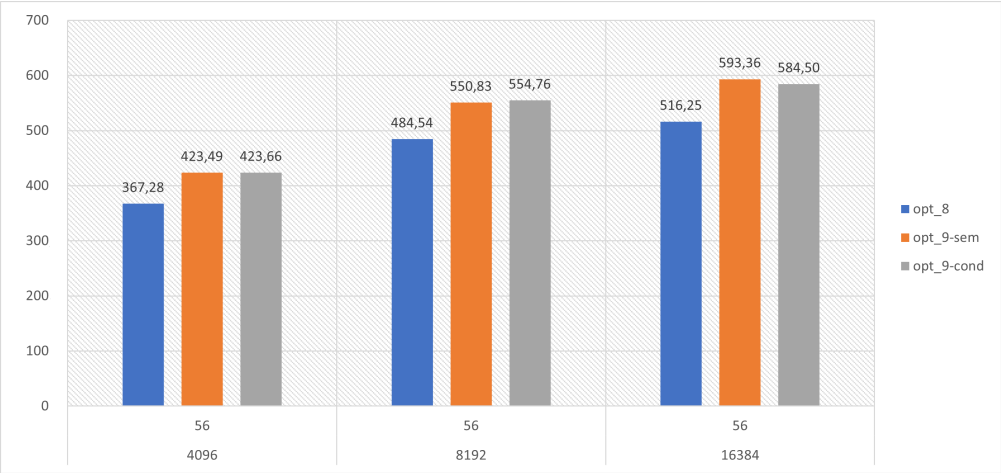

Figura 6.9: GFLOPS con BS=128 para T=56 y tipo *double*

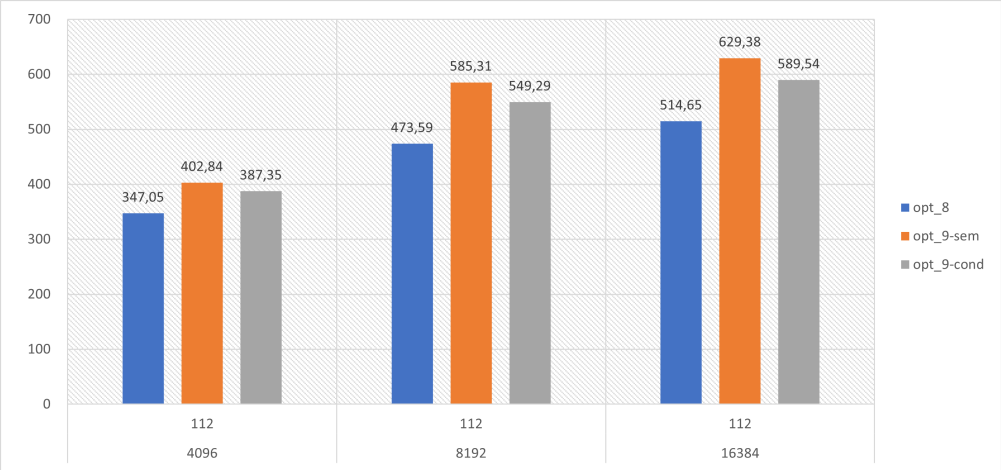

Figura 6.10: GFLOPS con BS=128 para T=112 y tipo *double*

7. Conclusiones y Trabajos Futuros

7.1. Conclusiones generales

En el presente trabajo, se describieron las características de las arquitecturas paralelas de memoria compartida y su programación; en particular, considerando las librerías: Pthreads y OpenMP. Luego, se explicó el algoritmo FW para cómputo de caminos mínimos en grafos y las dependencias de datos presentes en el mismo. Con estas consideraciones, se presentó la técnica de blocking como solución a la baja explotación de la localidad de datos y se analizaron diferentes optimizaciones paralelas, aplicadas en el código base. Como estas fueron realizadas para el procesador Intel Xeon Phi KNL, por tal motivo se debieron realizar modificaciones en varios casos para su correcto funcionamiento en el multicore Intel Xeon Platinum 8276L.

A continuación, se propuso una nueva optimización paralela para mejorar el tiempo de resolución del algoritmo, la cual fue implementada de dos maneras diferentes. Posteriormente, se realizaron las correspondientes pruebas de verificación con diferentes tamaños de entrada, de bloque, tipo de dato y cantidad de hilos.

A partir de los resultados obtenidos y su posterior análisis, se pueden mencionar las siguientes conclusiones:

- Respecto a la adaptación del código base:
  - Cada versión incremental del código base también reflejó mejoras similares de rendimiento en el nuevo procesador. Las mayores ganancias se obtienen en las optimizaciones que añaden la vectorización de instrucciones (Opt-2 a Opt-4).
  - Entre las adaptaciones realizadas se encuentra la actualización de un *flag* de compilación, precisamente el utilizado para la vectorización con AVX-512 en la versión Opt-4; y la supresión de Opt-1 debido a la inexistencia de la memoria adicional MCDRAM.
  - Al igual que en el caso del Xeon Phi KNL, la mejor combinación de granularidad y tipo de afinidad resultó ser *fine-balanced*.
- La nueva propuesta consistió en una optimización intra-ronda con dos implementaciones: una mediante el uso de semáforos (Opt-9-Sem) y otra con variables condición (Opt-9-Cond). La primera consiguió un mayor rendimiento para el tipo de dato *double*, mientras que la segunda resultó levemente superior para tipo *float*.
- Ambas versiones de Opt-9 superan en GFLOPS en todos los casos testeados a Opt-8, siempre que se utilice el tamaño de bloque óptimo mencionado. Para el tipo de dato float, la mejora promedio es del 1.03×, alcanzando un máximo de 1.05×. Sin embargo, los mejores resultados se observan para el tipo de dato double, donde la aceleración promedio sube a 1.12× y la máxima a 1.24×.
- Más allá de la reducción en el tiempo de ejecución, un beneficio indirecto de la optimización propuesta resulta la eliminación en la variación de los parámetros óptimos que estaba presente en el código base. El nuevo incremento Opt-9 ahora aporta el menor tiempo de ejecución para un tamaño de bloque estable ($BS = 128$) independientemente

del tamaño de entrada, cantidad de hilos y tipo de dato. Esto simplifica la ejecución de las pruebas necesarias para el uso de los algoritmos.

Respecto al cumplimiento del objetivo propuesto, se considera que ha sido alcanzado de manera satisfactoria. En primer lugar, se logró adaptar el código base realizado por Costi al nuevo procesador realizando cambios menores, y luego se pudo agregar una nueva optimización al programa existente para reducir aún más el tiempo de ejecución del algoritmo FW. Si bien la aceleración respecto a Opt-8 es modesta para float (hasta 1.05×), sí resulta significativa para double (hasta 1.24×). En cualquier caso, sigue representando un impacto positivo en el rendimiento general de los casos testeados, sumado a las optimizaciones previamente aplicadas desde Opt-0.

Finalmente, vale la pena destacar que el código fuente se encuentra público en un repositorio de GitHub, accesible mediante el siguiente link: `https://bit.ly/pps-fw-xeonplatinum`

7.2. Próximos pasos

Como trabajos futuros que se desprenden a partir de esta investigación, se sugieren las siguientes ideas:

- Proponer nuevas versiones incrementales como una **optimización inter-ronda**, a modo de eliminar la barrera de sincronización al término de cada ronda, y realizar las pruebas correspondientes para determinar si se pueden obtener nuevas mejoras.
- Realizar modificaciones al código con el fin de posibilitar su compilación mediante el nuevo compilador **ICX** de Intel, diseñado para los nuevos procesadores de dicha marca, manteniendo la filosofía de las optimizaciones incrementales (una versión siempre agrega una mejora respecto a su predecesor).
- Desarrollar una librería para facilitar la inclusión y utilización del algoritmo paralelo FW en programas de C/C++, con sus optimizaciones internamente implementadas, que permita especificar los parámetros de aplicación que resulten necesarios.

REFERENCIAS

[1] L. R. Foulds, *Graph Theory Applications*. Springer, 1992, https://doi.org/10.1007/978-1-4612-0933-1.

[2] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, pp. 345–, Jun. 1962.

[3] S. Warshall, "A theorem on boolean matrices," *J. ACM*, vol. 9, no. 1, pp. 11–12, Jan. 1962.

[4] S. Jalali and M. Noroozi, "Determination of the optimal escape routes of underground mine networks in emergency cases," *Safety Science*, vol. 47, no. 8, pp. 1077 – 1082, 2009.

[5] P. Khan, G. Konar, and N. Chakraborty, "Modification of floyd-warshall's algorithm for shortest path routing in wireless sensor networks," in *2014 Annual IEEE India Conference (INDICON)*, Dec 2014, pp. 1–6.

[6] A. Nakaya, S. Goto, and M. Kanehisa, "Extraction of correlated gene clusters by multiple graph comparison," *Genome Informatics*, vol. 12, pp. 44–53, 2001.

[7] L. Wang, M. Springer, H. Heibel, and N. Navab, "Floyd-warshall all-pair shortest path for accurate multi-marker calibration," in *2010 IEEE International Symposium on Mixed and Augmented Reality*, 2010, pp. 277–278.

[8] M. Penner and V. K. Prasanna, "Cache-friendly implementations of transitive closure," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 185–.

[9] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, *A Blocked All-Pairs Shortest-Paths Algorithm*. Springer Berlin Heidelberg, 2000, pp. 419–432.

[10] S. Han and S. Kang, "Optimizing all-pairs shortest-path algorithm using vector instructions," 2006.

[11] S.-C. Han, F. Franchetti, and M. Püschel, "Program generation for the all-pairs shortest path problem," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 222–232. [Online]. Available: https://doi.org/10.1145/1152154.1152189

[12] L.-y. Zhang, M. Jian, and K.-p. Li, "A parallel floyd-warshall algorithm based on tbb," in *2010 2nd IEEE International Conference on Information Management and Engineering*, 2010, pp. 429–433.

[13] T. Srinivasan, R. Balakrishnan, S. Gangadharan, and V. Hayawardh, "A scalable parallelization of all-pairs shortest path algorithm for a high performance cluster environment," in *2007 International Conference on Parallel and Distributed Systems*, 2007, pp. 1–8.

[14] E. Solomonik, A. Buluç, and J. Demmel, "Minimizing communication in all-pairs shortest paths," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 548–559.

[15] K. Hou, H. Wang, and W. c. Feng, "Delivering parallel programmability to the masses via the intel mic ecosystem: A case study," in *2014 43rd International Conference on Parallel Processing Workshops*, Sept 2014, pp. 273–282.

[16] E. Rucci, A. De Giusti, and M. Naiouf, "Blocked All-Pairs Shortest Paths Algorithm on Intel Xeon Phi KNL Processor: A Case Study," in *Computer Science – CACIC 2017*, A. E. De Giusti, Ed. Cham: Springer Int. Pub., 2018, pp. 47–57.

[17] U. Costi, "Aceleración del Algoritmo Floyd-Warshall sobre Intel Xeon Phi KNL," Tesina de Licenciatura en Informática, Universidad Nacional de La Plata, 2020.

[18] B. Wilkinson and M. Allen, *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*, 2nd ed. Prentice Hall, 2004.

[19] G. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley, 2000.

[20] "The OpenMP API specification for parallel programming." [Online]. Available: https://www.openmp.org/specifications/

[21] Grama, Gupta, Karypis, and Kumar, *Introduction to Parallel Computing*. Addison Wesley, 2003.

[22] J. Reinders, J. Jeffers, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming Knights Landing Edition*. Boston, MA, USA: Morgan Kaufmann Publishers Inc., 2016.

[23] Intel Corporation, "Intel® Xeon® Platinum 8276L Processor." [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/192475/intel-xeon-platinum-8276l-processor-38-5m-cache-2-20-ghz/specifications.html

[24] ——, "Thread Affinity Interface." [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming/openmp-support/openmp-library-support/thread-affinity-interface.html