Adaptación de Algoritmo OpenMP para Computar Caminos Mínimos en Grafos en Arquitecturas x86

Sergio Calderón[1] [0000-0001-6736-7358], Enzo Rucci[1], and Franco Chichizola[1][0000-0001-8857-6343]

III-LIDI, Facultad de Informática, UNLP – CIC.
La Plata (1900), Bs As, Argentina
{scalderon,erucci,francoch}@lidi.info.unlp.edu.ar

Resumen Los grafos han adquirido una relevancia significativa para modelar y resolver problemas en diversas áreas. El algoritmo FloydWarshall (FW) permite hallar los caminos mínimos entre vértices. Es una solución de alta demanda computacional ($O(n^3)$), debiendo emplear cómputo paralelo cuando el tamaño del problema escala. En este trabajo, se presenta la optimización de FW en arquitecturas multicore x86 de propósito general, adaptando 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 las mejoras incrementales aplicadas. Las pruebas realizadas en un servidor con 2×Intel Xeon Platinum 8276L y en un equipo comercial con Intel Core i5-10400F muestran mejoras acumuladas de 7.31× y 6.98×, respectivamente. Todas las optimizaciones resultan beneficiosas, aunque con distinto impacto. Por último, se plantea la idea de una nueva optimización FW.

Keywords: Floyd-Warshall · Multicore · HPC · Caminos mínimos · Xeon · Xeon Phi Knights Landing · Core · OpenMP

1. Introducción

El algoritmo de Floyd-Warshall (FW) [2,16] permite calcular y hallar los caminos mínimos entre todos los vértices de un grafo pesado. Es por lo que se lo ha empleado en ámbitos diversos como el tráfico automovilístico [6], las redes de computadoras [7], bioinformática [8], computación gráfica [15], 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 [9,14,4,3,17,13,12]. En particular para las arquitecturas Xeon Phi de Intel, Rucci *et al.* [11] exploró su uso para acelerar FW en la primera generación (KNC, Knights Corner), mientras que Costi *et al.* [1] lo extendió a la segunda (Knights Landing, KNL).

En este artículo, se propone tomar el trabajo realizado por Costi *et al.* [1] y adaptar su código para que pueda ejecutarse en procesadores multicore Intel x86, abandonando la especificidad del Xeon Phi KNL. Para ello, se verificará una a una las optimizaciones propuestas por [1], realizando ajustes al código de base donde sea necesario, y

analizando su rendimiento en dos servidores de referencia ante diferentes escenarios de prueba. Resulta importante mencionar que este trabajo representa el primer paso hacia la propuesta de nuevas optimizaciones para FW y el posterior desarrollo de una librería de código especializada.

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 la implementación realizada. A continuación, la Sección 4 muestra los resultados experimentales obtenidos. Finalmente, la Sección 5 enumera las líneas a futuro y la Sección 6 resume las conclusiones del trabajo.

2. Marco referencial

2.1 Intel Xeon Phi

Xeon Phi es el nombre comercial que Intel usó para una serie de procesadores many-core orientados a HPC. En 2012, Intel lanzó la primera generación (KNC) que principalmente contaba con hasta 61 núcleos Pentium x86 con unidades vectoriales extendidas (512 bits, específicas para KNC) e Hyper-Threading (cuatro hilos de hardware por núcleo). Mientras KNC se conectaba al procesador host a través del bus PCI Express, la segunda generación (KNL) pudo funcionar como procesador independiente. Entre sus principales características, se pueden mencionar la mayor cantidad de núcleos con soporte para hyper-threading (hasta 72), la incorporación de las instrucciones vectoriales AVX-512 (también de 512 bit pero compatibles con procesadores Xeon) y la integración de una memoria de alto ancho de banda, denominada MCDRAM [10]. La última generación (Knights Mill, KNM) fue lanzada a finales de 2017, siendo una variante de KNL con instrucciones específicas para aprendizaje automático profundo. Finalmente, Intel anunció que discontinuaría la serie Xeon Phi en 2018 para dedicarse al desarrollo de placas gráficas (GPUs)[1].

2.2 Intel Xeon y Core

En la actualidad, Intel presenta dos gamas de procesadores de la familia x86: Xeon y Core. Los procesadores Intel Xeon están diseñados para tareas empresariales y de servidor que requieren alta potencia de procesamiento y confiabilidad, mientras que los procesadores Intel Core son ideales para uso general, incluidos juegos, aplicaciones ofimáticas y entretenimiento multimedia.

En cuanto a las características arquitectónicas, los Xeon suelen tener más núcleos, tecnologías específicas para virtualización y seguridad, soporte para configuración en múltiples sockets, entre otras características avanzadas. En sentido contrario, los procesadores Core suelen tener frecuencias más altas, menor consumo energético y menor precio.

[1] https://hardzone.es/2018/07/25/intel-adios-xeon-phi-reemplazados-tarjetas-graficas/

La elección entre Xeon y Core dependerá de las necesidades específicas de la aplicación, el presupuesto y las preferencias del usuario.

2.3 Algoritmo FW

El pseudo-código de FW se muestra en la Fig. 1. Dado un grafo G de N vértices, FW recibe como entrada una matriz densa D de $N \times N$ que contiene las distancias entre todos los pares de vértices G, donde $D_{i,j}$ representa la distancia del nodo i al nodo j[2]. FW computa N iteraciones, evaluando en la k-ésima iteración todos los posibles caminos entre los vértices i y j que tienen a k como vértice intermedio. Como resultado, produce una matriz actualizada D, donde $D_{i,j}$ contiene ahora la distancia mínima entre los nodos i y j hasta ese paso. Complementariamente, FW va construyendo una matriz adicional P que registra los caminos asociados a las distancias mínimas.

Algoritmo FW por bloques. A primera vista, la estructura de triple bucle anidado de este algoritmo resulta similar al de la multiplicación de matrices densas (MM). Sin embargo, como las lecturas y escrituras se realizan sobre la misma matriz, los tres bucles no pueden ser intercambiados libremente, como es el caso de MM. A pesar de esto, el algoritmo FW puede ser computado por bloques bajo ciertas condiciones [14].

```
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
```

Figura 1: Pseudocódigo del algoritmo de FW

El algoritmo FW por bloques (FWB) divide a la matriz D en bloques de tamaño $TB \times TB$, totalizando $(N/TB)^2$ bloques. El cómputo se organiza en $R = N/TB$ rondas, donde cada ronda consta de 4 fases ordenadas de acuerdo a las dependencias de datos entre los bloques:

- Fase 1: actualizar el bloque $D^{k,k}$ debido a que sólo depende de sí mismo.
- Fase 2: actualizar los bloques de la fila k de bloques ($D^{k,*}$) debido a que cada uno de estos depende de sí mismo y de $D^{k,k}$.
- Fase 3: actualizar los bloques de la columna k de bloques ($D^{*,k}$) debido a que cada uno de estos depende de sí mismo y de $D^{k,k}$.
- Fase 4: actualizar los bloques restantes $D^{i,j}$ de la matriz porque cada uno depende de los bloques $D^{i,k}$ y $D^{k,j}$ de su fila y columna de bloques, respectivamente.

En la Fig. 2 se muestra gráficamente cada una de las fases de cómputo y las dependencias entre bloques. Los bloques en amarillo son aquellos bloques que están siendo computados, los grises los que ya se procesaron, y los verdes representan los que faltan computar.

[2] Si no existe un camino entre los nodos i y j, se asigna *infinito* a su distancia (usualmente representado como el valor positivo más grande)

2.4 Código de base

Como código de base se empleó el de [1], el cual fue desarrollado específicamente para procesadores de la arquitectura Xeon Phi KNL de Intel [10]. Para la implementación final del algoritmo FW con blocking en esta arquitectura paralela se realizaron diversos análisis. Las mejoras encontradas se fueron combinando 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 se implementa en la fase 1, ya que computa un único bloque. En el resto de fases se usa la forma inter-bloque, ejecutando en paralelo las fases 2 y 3 por ser independientes entre sí, y luego la fase 4.
- **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 poca capacidad, por lo que se usa el comando numactl -p para asignar de forma auxiliar en la DDR.
- **Opt-2: Vectorización guiada por SSE**. Mediante la directiva `simd` de OpenMP, se indica al compilador que fuerce la vectorización de un bucle. Habitualmente, los compiladores suelen utilizar por defecto el conjunto de instrucciones SSE de 128 bits, lo que permite procesar hasta 2 datos de 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 utilicen instrucciones AVX de 256 bits, lo que permite duplicar la cantidad de operaciones simultáneas respecto a SSE (si está soportado).

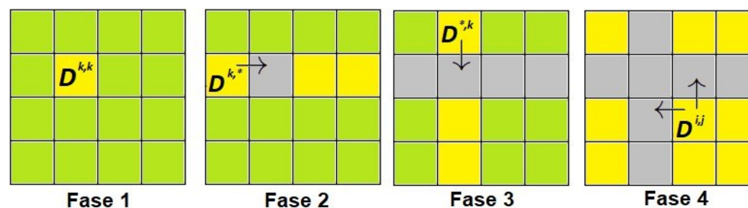

Figura2: Fases del cómputo de FWB y sus dependencias

- **Opt-4: Vectorización guiada por AVX-512**. ídem anterior, pero el flag indicado es *`-xMIC-AVX512`*, para emplear instrucciones de 512 bits. De esta forma, se pueden procesar hasta 8 datos tipo *double* o 16 tipo *`float`* en simultáneo en un procesador que lo soporte.
- **Opt-5: Alineación de datos**. Mediante la función `_mm_malloc()` de Intel se reserva memoria de modo que los datos se almacenen alineados respecto al comienzo de cada línea de caché. Indicando el ancho de línea de la cache L1D al reservar memoria, se optimizan las lecturas y escrituras posteriores.
- **Opt-6: Predicción de saltos**. La función `_builtin_expect(exp, c)` indica al compilador el resultado más probable (*c*) en una sentencia IF cuya expresión es (*exp*). La comparación entre distancias es un hotspot clave de FW, que en la

mayoría de veces será *false* (no se actualiza *D*). Mientras más acierte el planificador, más paralelismo a nivel de instrucciones podrá aprovechar el procesador.

- **Opt-7: Desenrollado guiado de bucles**. Realizado mediante la directiva `unroll(FD)`, se reduce la cantidad de iteraciones de un bucle según un factor convenientemente elegido para explotar la vectorización disponible.
- **Opt-8: Afinidad de hilos**. Se indica cómo realizar la distribución de hilos entre los núcleos según la variable de entorno *KMP_AFFINITY*. Se especifica un tipo de afinidad (`balanced, compact o scatter`) y la granularidad (*fine*, *core* o *tile*). En Xeon Phi KNL, la configuración óptima fue *fine, balanced*.

3. Implementación

El código de base explicado en la Sección 2 fue adaptado para su ejecución en los siguientes equipos de distintas prestaciones:

- Un equipo de gama comercial con procesador Intel Core i5-10400F de arquitectura x86. Posee 6 núcleos (2 hilos hw por núcleo) de frecuencia base de 2.90 GHz y 32 GB de memoria RAM. El sistema operativo es Debian GNU/Linux 11.
- Un equipo de alto rendimiento constituido por dos procesadores Intel Xeon Platinum 8276L en Dual-Socket. Cada procesador posee 28 cores de 2.20 GHz, nuevamente con 2 hilos hardware cada uno, resultando un total de 112 hilos. Dispone de un total de 250 GB de memoria RAM y la distribución de Linux instalada es Ubuntu 20.04 LTS.

En ambos casos, se empleó el compilador Intel ICC, parte de la suite oneAPI versión 2021.7.1. A continuación, se detallan las modificaciones realizadas a las optimizaciones para su correcta adaptación en cada equipo.

- **Opt-0:** se mantiene el tipo de paralelización original, por tanto, las directivas *for* de OpenMP y el orden de las fases permanecen sin cambios.
- **Opt-1:** descartado en ambos casos al no contar con memoria MCDRAM, por lo que se omite el comando `numactl -p` en las pruebas.
- **Opt-2 y Opt-3:** sin cambios para las vectorizaciones por SSE y AVX2.
- **Opt-4:** en Intel Xeon Platinum se reemplaza el flag asociado por el recomendado a dicho procesador (de `-xMIC-AVX512` a `-xCORE-AVX512`); mientras que en Intel Core i5 se descarta por no contar con este nivel de vectorización.
- **Opt-5:** en Intel Xeon Platinum se mantiene el parámetro SIMD WIDTH en el valor 512; mientras que en Intel Core i5 se reduce a 256 por ser AVX-2 la máxima vectorización disponible, además de usar el flag de Opt-3.
- **Opt-6:** sin cambios en las macros *likely(exp)* y *unlikely(exp)*.
- **Opt-7:** el factor *FD* depende de SIMD WIDTH y TYPE SIZE, parámetros definidos en compilación. El valor de este último es 32 para simple precisión (*float*) y 64 para doble precisión (*double*).
- **Opt-8:** se elige la afinidad y granularidad óptima de manera empírica.

4. Trabajo experimental y resultados obtenidos

4.1 Pruebas realizadas

Las pruebas se realizaron sobre los equipos descritos en la Sección 3 una vez aplicadas las modificaciones. Para ambos equipos se consideró la variación del tamaño de la matriz de distancias (N = {4096, 8192, 16384}), el tamaño de bloque (TB = {32, 64, 128, 256}) y el tipo de dato (*float, double*). Además, se emplearon diferente cantidad de hilos OpenMP, considerando dos estrategias: un hilo por núcleo físico y uno por núcleo lógico (T_{i5} = {6,12}, T_{Xeon}={56,112}).

El programa inicia con la carga del grafo para la creación de las matrices D y P. Se utiliza la misma entrada para todas las pruebas que se realicen con un determinado N y TB. Se estableció un porcentaje de completitud del 70% para todos los casos. Cada prueba particular fue repetida 8 veces, calculando los promedios para reducir la variabilidad.

4.2 Resultados experimentales

Como métrica de rendimiento, se emplea $GFLOPS = \frac{2 \times N^3}{t \times 10^9}$, donde N es la cantidad de vértices del grafo, t es el tiempo de ejecución (en segundos), y 2 representa la cantidad de operaciones en punto flotante que se ejecutan en cada iteración del bucle más interno del algoritmo.

Considerando un tamaño de entrada intermedio (N = 8192) y tipo *float*, se presentan los resultados obtenidos para cada equipo en las Fig. 3 (Core i5) y 4 (Xeon). 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 TB. Se puede observar que el mejor rendimiento para ambos equipos se logra usando 1 hilo por núcleo lógico y TB =128 desde Opt-3 en adelante.

Promedio de GFlops	opt_0_1	opt_2	opt_3	opt_5	opt_6	opt_7	opt_8
6							
32	12,35	20,62	61,92	65,95	78,89	103,11	103,44
64	14,37	19,38	73,58	78,41	95,47	136,61	137,42
128	19,99	23,40	77,06	83,50	104,07	146,66	146,70
256	20,34	24,61	55,09	56,99	86,83	110,86	111,34
12							
32	16,24	28,15	66,85	70,26	87,15	112,24	112,65
64	19,99	25,52	78,29	81,36	101,56	146,24	147,05
128	21,09	29,92	78,41	82,82	107,65	154,22	154,29
256	21,22	32,13	60,51	62,12	94,19	110,68	110,80

Figura 3: GFLOPS promedio en Core i5 para N=8192 y tipo *float*

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	116,07	185,12	222,85	346,40	374,78	494,90
64	62,95	81,35	111,03	189,76	230,82	472,82	480,55	710,57
128	87,15	110,83	132,55	215,53	334,13	507,52	559,67	831,70
256	87,90	111,65	112,06	185,94	232,88	441,03	444,93	641,24
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
256	124,47	169,32	336,85	354,80	372,23	432,98	456,85	470,91

Figura 4: GFLOPS promedio en Xeon Platinum para N=8192 y tipo *float*

En la Tabla 1 se especifica la mejora obtenida en cada *Opt-X* respecto a su anterior, incluyendo también la comparación con [1]. Las Fig. 5 y 6 muestran el rendimiento alcanzado utilizando la configuración óptima mencionado en el equipo Core i5 y el Xeon, respectivamente. Se puede observar en estas figuras que efectivamente hay un incremento de los GFLOPS obtenidos al avanzar de optimización en ambos equipos. Se observa que el mayor salto de aceleración se encuentra en la versión *Opt-3* mediante la vectorización con *AVX-2*, siendo de **2.6x** aproximadamente. Luego, al considerar la opción de mayor vectorización frente a la que no aprovecha esta optimización (*Opt-0*), se obtiene una mejora total de **3.96x** en Xeon Platinum y **3.72x** en Core i5 (se debe tener en cuenta que este último no admite la vectorización *AVX-512*).

	Opt-1	Opt-2	Opt-3	Opt-4	Opt-5	Opt-6	Opt-7	Opt-8
Core i5	-	1.42x	2.62x	-	1.06x	1.30x	1.43x	1.0005x
Xeon Platinum	-	1.32x	2.61x	1.15x	1.21x	1.18x	1.09x	1.13x
Xeon Phi KNL	1.03x	1.57x	2.19x	2.10x	1.05x	2.63x	1.40x	1.0012x

Tabla 1: Mejora incremental en cada equipo para N=8192

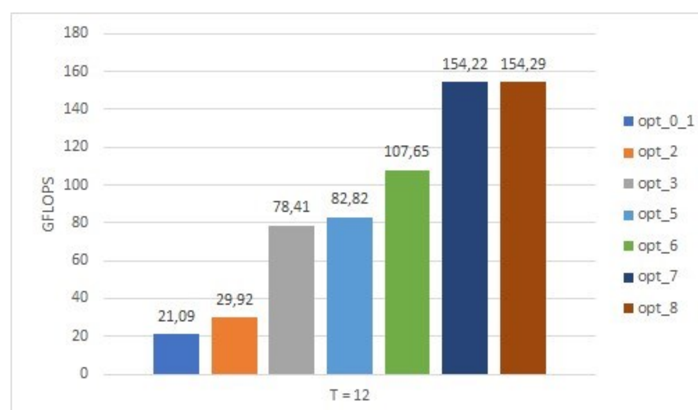

Figura 5: GFLOPS en Core i5 para tipo *float*, configuración óptima

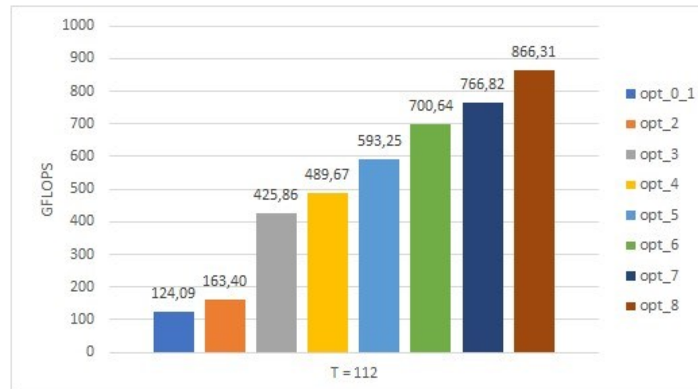

Figura 6: GFLOPS en Xeon Platinum para tipo *float*, configuración óptima

La siguiente mejora destacable ocurre en la predicción de saltos de *Opt-6*, alcanzándose un **1.30x** para Core i5, y **1.18x** para Xeon Platinum. Su optimización posterior *Opt-7*, que consiste en el desenrollado de bucles, brinda una aceleración similar, siendo **1.43x** y **1.09x** en los equipos mencionados. Para la versión *Opt-8* se probaron las 6 combinaciones de granularidad (core, fine) y afinidad (balanced, compact y scatter) [5]. Se encontró que la mejor configuración es **fine-balanced**, por leve diferencia sobre *scatter*, como se observa en la Fig. 7 usando la mitad de hilos disponibles y tipo float. Cuando se dispone de pocos núcleos, en la última optimización se puede medir una diferencia de GFLOPS mínima en comparación a *Opt-7*, lo cual se ve reflejado en el equipo comercial con una mejora de **1.0005x**.

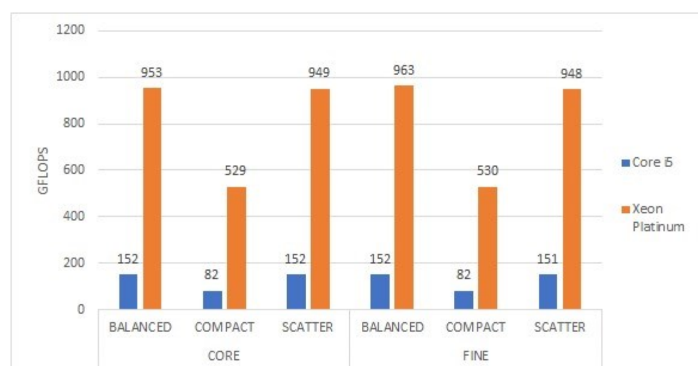

Figura 7: GFLOPS para N=16384 según KMP AFFINITY

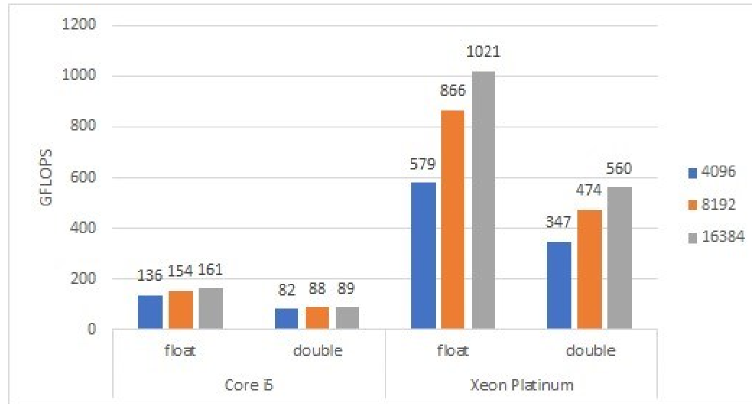

Figura 8: GFLOPS en Opt-8 para tipo float y double

En resumen, si se considera el desempeño integral de las versiones tomando como base a *Opt-0*, al sumar las mejoras obtenidas desde *Opt-2* hasta *Opt8*, se tiene un acumulado de **7.31x** y **6.98x** para Core i5 y Xeon Platinum, respectivamente. Adicionalmente, en la comparación con el Xeon Phi KNL, se puede notar que todas las optimizaciones resultaron beneficiosas, aunque no todas impactaron de la misma manera.

Por último, en la Fig. 8 se muestran los GFLOPS alcanzados con la mejor versión para cada equipo, tipo de dato y tamaño de entrada testeado, empleando la configuración óptima de *T* y *TB* a cada caso. En primer lugar, resulta claro que se obtienen mayores GFLOPS en el equipo Xeon que en el Core i5, considerando su potencia de cómputo. En segundo lugar, se observa que el rendimiento se incrementa a medida que aumenta *N*, dado a la mayor proporción de cómputo frente a sincronización. En particular, la magnitud de esta diferencia es más notoria cuando se dispone de más hilos (caso Xeon Platinum). En tercer lugar, el uso de un tipo de dato de mayor precisión como *double* puede llevar a un resultado más fiable; sin embargo, se debe tener en cuenta que tendrá un costo en el tiempo de respuesta, ya que el rendimiento decae hasta un 45%.

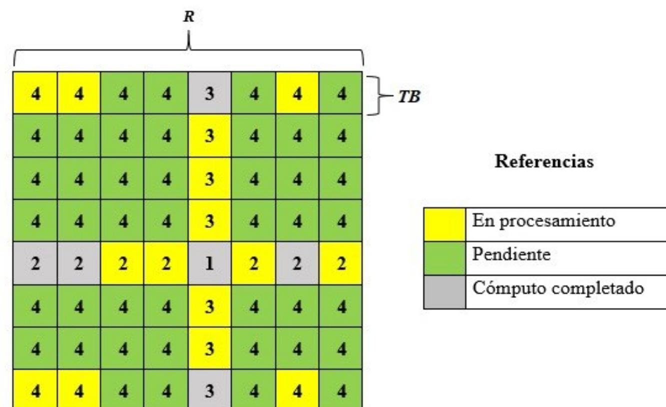

Figura9: Ejemplo de oportunidad de optimización para $R = 8$ en la ronda $k = 4$

5. Trabajo futuro

A partir del trabajo desarrollado y de los resultados encontrados, se mencionan algunos de los próximos pasos a seguir:

- Desarrollar nuevas versiones incrementales y realizar las pruebas correspondientes para determinar si se pueden obtener nuevas mejoras. En el algoritmo FWB presentado, se da comienzo a la fase 4 de una ronda sólo una vez que fueron completadas las fases 2 y 3. Una idea propuesta es reducir esta espera ociosa ya que se requiere satisfacer únicamente las dependencias directas de cada bloque de fase 4 en particular, mientras se continúa el cómputo de las fases 2 y 3. La Fig. 9 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). Esta posible optimización toma mayor relevancia cuando *T* es *grande* y *TB* es *chico*, y su implementación probablemente requiera una sincronización de grano más fino que la que las directivas de OpenMP pueden proveer, siendo necesario trabajar a nivel de Pthreads.
- Realizar modificaciones al código con el fin de posibilitar su compilación mediante el nuevo compilador **ICX** de Intel, el cual incorpora a LLVM [3]como *backend*, 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.

6. Conclusiones

En este artículo, se tomó el trabajo realizado por Costi*et al.* [1] y se adaptó su código para que pueda ejecutarse en procesadores multicore Intel x86, abandonando la especificidad del Xeon Phi KNL. Para ello, se verificó una a una las optimizaciones propuestas por Costi *et al.* [1], realizando ajustes al código de base donde fue necesario, y se analizó su rendimiento en dos servidores de referencia ante diferentes escenarios de prueba. A partir de los resultados obtenidos y su posterior análisis, se pueden mencionar las siguientes conclusiones:

- Entre las adaptaciones realizadas para Xeon Platinum 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 omisión de *Opt-1* debido a la inexistencia de la memoria adicional MCDRAM.
- En el equipo con Intel Core i5, se debió suprimir la optimización *Opt-4* además de la mencionada *Opt-1*, ya que la máxima vectorización disponible es AVX2.

[3] The LLVM Compiler Infrastructure Project https://llvm.org/

- Todas las optimizaciones aplicadas originalmente al Xeon Phi KNL también resultaron beneficiosas en los dos equipos x86 testeados. Las mayores ganancias se obtienen en las optimizaciones que añaden la vectorización de instrucciones.

Habiendo realizado la adaptación del código original, se ha planteado la idea de una nueva posible optimización para FW y su compatibilización con el nuevo compilador de Intel. Posteriormente, se planea el desarrollo de una librería que facilite el uso de estos algoritmos en aplicaciones externas. 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/cacic-2023-fw

Referencias

1. Costi, U.: Aceleración del Algoritmo Floyd-Warshall sobre Intel Xeon Phi KNL. Tesina de Licenciatura en Informática, Universidad Nacional de La Plata (2020)
2. Floyd, R.W.: Algorithm 97: Shortest path. Commun. ACM **5**(6), 345– (Jun 1962). https://doi.org/10.1145/367766.368168
3. 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
4. York, NY, USA (2006). https://doi.org/10.1145/1152154.1152189
5. Han, S., Kang, S.: Optimizing all-pairs shortest-path algorithm using vector instructions (2006)
6. Intel Corporation: Thread Affinity Interface, 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
7. Jalali, S., Noroozi, M.: Determination of the optimal escape routes of underground mine networks in emergency cases. Safety Science **47**(8), 1077 – 1082 (2009).
8. 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. Nakaya, A., Goto, S., Kanehisa, M.: Extraction of correlated gene clusters by multiple graph comparison. Genome Informatics **12**, 44–53 (2001)
11. 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)
12. Reinders, J., Jeffers, J., Sodani, A.: Intel Xeon Phi Processor High Performance Programming Knights Landing Edition. Morgan Kaufmann Publishers Inc., Boston, MA, USA (2016)
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., Buluc¸, 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