Sobre el paralelismo anidado de tareas en la factorización LU de Matrices Jerárquicas

Rocío Carratalá-Sáez[1] y Enrique S. Quintana-Ortí[2]

Resumen— **En este artículo se presenta una versión paralela de la factorización LU de Matrices Jerárquicas ($\mathcal{H}$-matrices) provenientes de Métodos de Elementos de Contorno (BEM). Estas matrices contienen estructuras internas cuya dimensión varía durante la ejecución de operaciones sobre las mismas, por lo que es necesario desligar las estructuras de datos de aquellas utilizadas para representar las dependencias en las tareas en las que se basa la implementación paralelizada. Utilizamos el modelo de programación OmpSs-2 y su runtime para determinar el flujo de datos intrínseco al paralelismo en tiempo de ejecución, así como para aprovechar las dependencias débiles de tareas y la "liberación temprana" (*early release*) de dependencias. Gracias a estas funcionalidades, puede acelerarse la ejecución de la versión paralela de la $\mathcal{H}$-LU y mejorarse el rendimiento.**

Palabras clave— **Matrices jerárquicas, factorización LU, paralelismo basado en tareas, procesadores multinúcleo, BEM.**

I. Introducción

LAS matrices jerárquicas (abreviadas como $\mathcal{H}$-matrices), combinadas con la $\mathcal{H}$-aritmética, ofrecen una abstracción matemática eficiente para lidiar con problemas provenientes de métodos de elementos de contorno, operadores elípticos parcialmente diferenciables o ecuaciones integrales [8]. Estas matrices permiten comprimir una matriz original de dimensión $n \times n$ utilizando solamente $O(n\ c\ log\ n)$ elementos, así como realizar operaciones de álgebra lineal, tales como la factorización LU, con un coste de $O(n\ c^2\ log^2\ n)$ operaciones en coma flotante (flops), siendo c es un parámetro que puede ajustarse para controlar la precisión de la aproximación [6], [7].

A lo largo de los últimos años, se han desarrollado bibliotecas para realizar operaciones de álgebra lineal con $\mathcal{H}$-matrices. Algunos de los trabajos más relevantes han dado lugar a los paquetes HLib[1], H2Lib[2] y HLibPro[3], que en su mayoría utilizan los núcleos computacionales definidos en *Basic Linear Algebra Subprograms* (BLAS) [4] para realizar los cálculos de las operaciones básicas del álgebra lineal. En principio, las rutinas de estos paquetes software pueden (o pueden ser fácilmente modificadas para) ejecutarse en paralelo en arquitecturas multicore enlazando una implementación de BLAS, tal como Intel MKL. No obstante, el grado de eficiencia paralela alcanzable de este modo está limitado, principalmente debido a que las $\mathcal{H}$-matrices están *dominadas* por bloques de rango bajo, lo cual hace necesario explotar el paralelismo en un nivel superior.

Aunque algunas de las bibliotecas actuales presentan algoritmos paralelos multihilo, estos se basan en OpenMP o Intel TBB (*thread building blocks*). El paralelismo de tareas se ha aplicado recientemente en la resolución de sistemas de ecuaciones lineales tanto densos [3] como dispersos [1], alcanzándose en ambos campos buenos resultados a nivel de rendimiento en arquitecturas multinúcleo. Dado que las $\mathcal{H}$-matrices residen en un escenario intermedio entre las matrices densas y las dispersas, parece natual aplicar técnicas similares en operaciones de $\mathcal{H}$-aritmética. En [2] presentamos un prototipo de la implementación de la factorización $\mathcal{H}$-LU para sistemas lineales densos utilizando OmpSs. En dicho trabajo se prueba la viabilidad de optar por una estrategia paralela basada en tareas, si bien hay limitaciones; por un lado, el algoritmo empleado es un prototipo que asume bloques densos o nulos (no se incluyó soporte para bloques de rango bajo); y, por otro, la versión del modelo de programación OmpSs utilizada es la anterior a OmpSs-2, la cual carecía de algunas funcionalidades que son clave en el presente artículo.

En este trabajo presentamos una versión paralela de la $\mathcal{H}$-LU utilizando el modelo de programación OmpSs-2 para lograr una implementación paralela de la $\mathcal{H}$-LU disponible en H2Lib, una biblioteca Open Source desarrollada en el Scientific Computing Group de la Universität zu Kiel.

La $\mathcal{H}$-LU requiere que el *runtime* detecte en tiempo de ejecución las tareas de manera dinámica y los núcleos computacionales pueden realizarse mediante una ejecución secuencial de BLAS (para el caso denso), pero hay algunos aspectos de esta operación que la diferencian de los trabajos sobre matrices densas y/o dispersas previamente referenciados. En primer lugar, la "naturaleza recursiva" de las $\mathcal{H}$-matrices, dificulta la correcta detección y planificación de las tareas. A esto se suma que los bloques de rango bajo pueden variar (disminuyendo o aumentando su tamaño) en tiempo de ejecución. Gracias a las nuevas funcionalidades incorporadas en el modelo de programación OmpSs-2, es posible lidiar con dichas particularidades y alcanzar una buena eficiencia paralela.

El resto del artículo se estructura como sigue. En la Sección II detallamos algunos fundamentos matemáticos de las $\mathcal{H}$-matrices y cómo se obtienen y representan estas en la biblioteca H2Lib. En la Sección III presentamos las nuevas funcionalidades ofrecidas por el modelo de programación OmpSs-2 y, en la Sección IV, describimos el algoritmo para la $\mathcal{H}$-LU y cómo aprovechar dichas funcionalidades en

[1]Dpto. de Ingeniería y Ciencia de los Computadores, Univ. Jaume I, e-mail: rcarrata@uji.es.

[2]Dpto. de Informática de Sistemas y Computadores, Univ. Politècnica de València, e-mail: quintana@disca.upv.es.

[1]http://www.hlib.org/

[2]http://www.h2lib.org/

[3]https://www.hlibpro.com/

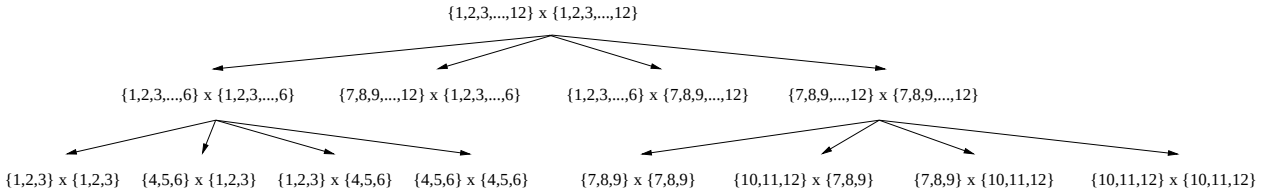

Fig. 1. *Block cluster tree* obtenido del conjunto de índices $I = \{1, 2, 3, ..., 12\}$.

una versión paralelizada del mismo. Finalmente, en la Sección V evaluamos el rendimiento de diferentes implementaciones paralelas y en la Sección VI resumimos las conclusiones obtenidas.

II. $\mathcal{H}$-matrices en H2Lib

A. Fundamentos de las $\mathcal{H}$-matrices

Para comprender cómo están representadas internamente en la biblioteca H2Lib las $\mathcal{H}$-matrices, presentamos brevemente algunos fundamentos matemáticos de las mismas. Para más detalles, véanse [7], [10].

El concepto abstracto de $\mathcal{H}$-matriz puede verse como una representación dispersa de los datos de una matriz densa que logra que los costes de almacenamiento se reduzcan en la medida en la que se explota la representación de los datos mediante bloques de rango bajo en forma factorizada, manteniendo el resto en bloques densos o de rango casi completo, dado que no hay beneficio al factorizarlos para su representación. Para obtener la jerarquía de bloques que define la estructura de la $\mathcal{H}$-matriz, debe utilizarse una condición de *admisibilidad*, la cual determina que un cierto bloque, que es admisible, puede aproximarse con una factorización de rango bajo, hasta una cierta precisión. Veamos esto.

Empecemos con la definición de *cluster tree* para un conjunto de índices dado.

Definición 1: Sea I un conjunto de índices con cardinalidad $n = 1$. El grafo $T_I = (E, V)$, donde V son los vértices y E las aristas, es un *cluster tree* sobre I, si I es la raíz de T_I y, $\forall\ v \in V$, o bien v es una hoja de T_I, o bien $v = \dot{\cup}_{v' \in S(v)} v'$, donde $S(v)$ representa el conjunto de descendientes directos de v.

Las $\mathcal{H}$-matrices representan un particionado jerárquico de un conjunto de índices en forma de *cluster tree*. Cuando se particiona el conjunto de índices producto $I \times I$, utilizando los subconjuntos de I definidos en T_I, obtenemos "*cluster trees* de *cluster trees*".

Definición 2: Sea T_I un *cluster tree* sobre I y considérese el nodo $b = p \times q$. El *block cluster tree* $T_{I\times I}$ sobre T_I puede definirse recursivamente para el nodo b, empezando por la raíz $I \times I$, como sigue:

$$S(b) = \begin{cases} \emptyset \text{ si } b \text{ es admisible o } S(p) = \emptyset \text{ o } S(q) = \emptyset, \\ S' \text{ en cualquier otro caso}, \end{cases}$$

donde $S' := \{p' \times q' : p' \in S(p), q' \in S(q)\}$.

La Figura 1 muestra un ejemplo de un *block cluster tree* cuyas hojas que conforman una partición $I \times I$, con $I = \{1, 2, 3, ..., 12\}$ utilizando un criterio de admisibilidad débil; véase [9] para más detalles.

El conjunto de matrices de rango bajo con un rango máximo k es un conjunto de $\mathcal{H}$-matrices.

Definición 3: El conjunto de $\mathcal{H}$-matrices para un *block cluster tree* $T_{I\times I}$ sobre un *cluster tree* T_I se define como:

$$\begin{aligned} \mathcal{H}(T_{I\times I}, k) := \{ & M \in \mathbb{R}^{I\times I} \mid \forall\ p \times q \in \mathcal{L}(T_{I\times I}) : \\ & rango(M\mid_{p\times q}) \leq k \vee \{p, q\} \\ & \cap\, \mathcal{L}(T_I) \neq \emptyset \} \end{aligned}$$

donde $\mathcal{L}(\mathcal{T}_\mathcal{I})$ es el conjunto de hojas de T_I y $k \in \mathbb{N}$.

Siguiendo con el ejemplo de la Figura 1, el *block cluster tree* definido en el ejemplo define el particionado de una matriz de tamaño 12×12 en una $\mathcal{H}$-matriz como la ilustrada en la Figura 2.

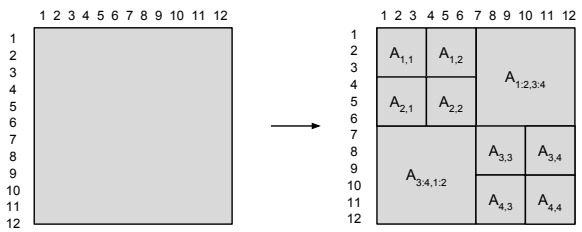

Fig. 2.$\mathcal{H}$-matriz obtenida aplicando el particionado definido por el *block cluster tree* de la Figura 1 sobre una matriz de tamaño 12×12.

Las $\mathcal{H}$-matrices utilizan la $\mathcal{H}$-aritmética para calcular la suma de matrices, su inversión, producto y la factorización LU con un coste computacional logarítmico [6]. Para esto, la suma de matrices de rango bajo se trunca para que tenga un rango c fijado (o, preferiblemente, con respecto a una precisión ϵ dada) mediante su descomposición en valores singulares (SVD) [5]. Dicho de otro modo, los resultados de las operaciones sobre $\mathcal{H}$-matrices (excepto el producto matriz-vector) son aproximados para poder garantizar una complejidad aritmética logarítmica.

B. Representación de $\mathcal{H}$-matrices en H2Lib

La biblioteca H2Lib ofrece rutinas secuenciales implementadas en C para el manejo de $\mathcal{H}$-matrices. Para poder determinar la partición que da lugar a una $\mathcal{H}$-matriz, es necesario clasificar en distintos conjuntos los grados de libertad (en adelante, DoFs) que definen el problema de origen, de manera que se pueda formar eficientemente un *cluster tree*. Asumiendo que conocemos la extensión geométrica de cada DoF que aparece en nuestra aplicación, esto se reduce frecuentemente al soporte de funciones básicas de elementos finitos. Gracias a esta información, podemos

configurar un cuadro delimitador de ejes paralelos $\mathcal{B}_t$ (o *bounding box*) que contenga la unión de todas las extensiones correspondientes al clúster t. Este cuadro, a su vez, se divide en dos partes a lo largo de alguna dimensión geométrica, lo que da lugar a dos cuadros separados $\mathcal{B}_{t_1}$, $\mathcal{B}_{t_2}$. A continuación, ambos cuadros se procesan de forma recursiva hasta que el número de DoFs ubicados en un cuadro es inferior a una constante prefijada, denotada por *tamaño de hoja* (C_{lf}) o *leafsize* (C_{lf}). Con el fin de manejar todos estos cuadros de manera eficiente, se organizan en una estructura de árbol (esto es, un *block cluster tree* anteriormente definido) expresada por `sonst` $= \{t_1, t_2\}$. Este proceso se resume en el Algoritmo 1.

Algoritmo 1 Construcción del *clustertree*

Entrada: Información geométrica sobre los DoFs almacenada en el vector `dofs`, de tamaño `size`.
Salida: Partición jerárquica de los DoFs vía el *clustertree* `t`.

```
procedure SETUP_CLUSTERTREE(dofs,size)
si size > C_lf entonces
  d ← FIND_SPLITTING_DIMENSION(dofs, size)
  sons ← 2
  t ← NEW_CLUSTER(dofs, size, sons)
  {dofs1, dofs2, size1, size2} ← SORT_DOFS(dofs, size, d)
  t_1 ← SETUP_CLUSTERTREE(dofs1, size1)
  t_2 ← SETUP_CLUSTERTREE(dofs2, size2)
  SONS(t) ← {t_1, t_2}
si no
  t ← NEW_LEAF_CLUSTER(dofs, size)
fin si
devolver t
```

En H2Lib, los distintos clústeres que conforman las particiones jerárquicas se representan con la estructura de datos (en C) que sigue:

```
struct cluster {
    uint      size;
    uint      *dofs
    uint      *bbox_min;
    uint      *bbox_max;
    cluster   *son;
    uint      sons;
}
```

Aquí, `size` es el total de alementos asociados al clúster y `dofs` es el vector que contiene los DoFs. El cuadro delimitador $\mathcal{B}_t$ para un clúster t se almacena entre los vectores `bbox_min` y `bbox_max`, respectivamente. Por su parte, `son` y `sons` representan el árbol de clústers.

La condición de admisibilidad empleada en H2Lib es:

$$\max\{\mathrm{diam}(\mathcal{B}_t), \mathrm{diam}(\mathcal{B}_s)\} \leq \eta \, \mathrm{dist}(\mathcal{B}_t, \mathcal{B}_s),$$

donde *diam* y *dist* hacen referencia al diámetro Euclideano del cuadro delimitador y la distancia Euclideana entre dos de ellos y $\eta \in \mathbb{R}_{>0}$ es el parámetro elegido para controlar el canje entre la cantidad de bloques admisibles en la matriz y la exactitud de la aproximación.

Finalmente, el particionado de la $\mathcal{H}$-matriz se hace de forma recursiva aplicando el Algoritmo 2 (una explicación detallada del mismo puede encontrarse en [7], [6]). En él, en un determinado nivel de la recursión, pueden producirse tres tipos de estructuras: un bloque de rango bajo (i.e., un nuevo bloque admisible), una nueva partición recursiva (vía una llamada al mismo algoritmo) o un bloque denso convencional (i.e., un bloque no admisible).

Algoritmo 2 Construcción del *block cluster tree*

Entrada: Clúster fila `t`, clúster columna `s`.
Salida: *blocktree* `b` para el par de clústers `(t,s)`.

```
procedure SETUP_BLOCKTREE(t,s)
si ADMISSIBLE(t, s) entonces
  b ← NEW_ADMISSIBLE_BLOCK(t, s)
si no
  si SONS(t) ≠ ∅ ∧ SONS(s) ≠ ∅ entonces
    b ← NEW_PARTITIONED_BLOCK (t, s)
    para todo t' ∈ SONS(t), s' ∈ SONS(s) hacer
      b[t'][s'] ← SETUP_BLOCKTREE(t', s')
    fin para
    devolver b
  si no
    b ← NEW_INADMISSIBLE_BLOCK(t, s)
  fin si
fin si
devolver b
```

La estructura con la que se representa una $\mathcal{H}$-matriz proveniente del proceso recursivo descrito en H2Lib (en C) tiene esta forma:

```
struct hmatrix {
    cluster  rc, cc;
    rkmatrix r;
    amatrix  f;
    hmatrix  *son;
    uint     rsons, csons;
}
```

En esta estructura, `rc` y `cc` son, respectivamente, los clústers de filas y columnas del bloque en concreto; `rsons` y `csons` representan la cantidad de hijos por fila y columna que respectivamente presenta dicho bloque (nótese que ambos serán 0 si el bloque es una hoja en el *block cluster tree*). Los bloques de rango bajo se almacenan en una `rkmatrix`, mientras que los densos en una `amatrix` y los bloques sobre los que se aplica de nuevo un particionado de manera recursiva se representan con un vector de punteros que referencian a los bloques hijos.

Por su parte, la estructura diseñada en H2Lib (en C) para almacenar un bloque de rango bajo es:

```
typedef struct rkmatrix {
        uint k;         /* Maximal rank */
        amatrix A;      /* Left factor A */
        amatrix B;      /* Right factor B */
}
```

III. El modelo OmpSs-2

El modelo de programación OpenMP 4.5, con el que OmpSs presenta ciertas similitudes, ofrece soporte para anidación y también para definir depen-

dencias entre tareas. No obstante, cuando se definen dependencias anidadas, es necesario aplicar medidas correctoras para asegurar la correcta coordinación de las dependencias entre los distintos niveles. Por ejemplo, es necesario añadir un punto de sincronización (mediante `taskwait`) al final de cada una de las tareas que está particionada en subtareas. Esto limita la eficiencia máxima que se puede lograr.

Por su parte, OmpSs-2 presenta dos características nuevas, que no están disponibles en OpenMP, muy útiles para abordar el problema en el que se basa el trabajo que se describe en este artículo: dependencias *débiles* y "liberación temprana" (*early release*) de dependencias.

Las dependencias respecto a operandos de una determinada tarea anotadas como *débiles* son ignoradas por el *runtime* cuando se determina si una tarea está lista para ejecutarse o no. Esto es posible porque dichos operandos solo son leídos o escritos por tareas secundarias y no por la tarea en la que se anotan dependencias respecto a ellos de forma débil. La ventaja de utilizar este tipo de dependencias es que las subtareas se pueden instanciar antes y en paralelo.

La liberación temprana de dependencias permite proporcionar dependencias de grano más fino entre tareas "hermanas". Esto se consigue liberando inmediatamente las dependencias de una tarea, en cuanto finaliza, siempre que no estén siendo utilizadas en ese momento por ninguna de sus subtareas. Además, tan pronto como las subtareas terminan, se liberan las dependencias que no están siendo utilizadas en ese momento por ninguna de sus tareas "hermanas".

En la Sección IV se detalla cómo afectan estas dos funcionalidades al paralelismo de tareas aplicado en la $\mathcal{H}$-LU.

IV. Factorización LU de $\mathcal{H}$-matrices

A. Algoritmo secuencial de la $\mathcal{H}$-LU

El algoritmo para la factorización LU de $\mathcal{H}$-matrices puede entenderse como una generalización del algoritmo a *Blocked Right Looking LU* que, en este caso, tiene en cuenta una estructura jerárquica [5]. Consideremos la $\mathcal{H}$-matriz $A \in \mathbb{R}^{n\times n}$ particionada del mismo modo que en la Figura 1; la secuencia de operaciones que calcula la $\mathcal{H}$-LU en ese caso es:

O1.1 :	$A_{1,1}$	:=	$L_{1,1}U_{1,1}$,
O1.2 :	$U_{1,2}$	:=	$L_{1,1}^{-1}A_{1,2}$,
O1.3 :	$L_{2,1}$	:=	$A_{2,1}U_{1,1}^{-1}$,
O1.4 :	$A_{2,2}$	:=	$A_{2,2} - L_{2,1}, \cdot U_{1,2}$,
O1.5 :	$A_{2,2}$	:=	$L_{2,2}U_{2,2}$,
O2 :	$U_{1:2,3:4}$	:=	$L_{1:2,1:2}^{-1}A_{1:2,3:4}$,
O3 :	$L_{3:4,1:2}$	:=	$A_{3:4,1:2}U_{1:2,1:2}^{-1}$,
O4 :	$A_{3:4,3:4}$	:=	$A_{3:4,3:4} - L_{3:4,1:2} \cdot U_{1:2,3:4}$,
O5.1 :	$A_{3,3}$	:=	$L_{3,3}U_{3,3}$,
O5.2 :	$U_{3,4}$	:=	$L_{3,3}^{-1}A_{3,4}$,
O5.3 :	$L_{4,3}$	:=	$A_{4,3}U_{3,3}^{-1}$,
O5.4 :	$A_{4,4}$	:=	$A_{4,4} - L_{4,3}, \cdot U_{3,4}$,
O5.5 :	$A_{4,4}$	:=	$L_{4,4}U_{4,4}$.

En concreto, se resuelven las siguientes operaciones básicas cuando se opera con bloques densos:

- Factorización LU (en el caso del ejemplo, en O1.1, O1.5, O5.1 y O5.5);
- Resolución de un sistema de ecuaciones lineal triangular (en el caso del ejemplo, con un factor triangular inferior unitario en O1.2, O2 y O5.2, o con un factor triangular superior en O1.3, O3 y O5.3);
- Producto matriz-matriz (en el caso del ejemplo, en O1.4, O4 y O5.4).

En el caso de haber bloques de rango bajo involucrados, que estarán representados en forma factorizada, esto es, $X = AB^*$ donde A, B tienen solamente k columnas (rango de X), las operaciones básicas para su manejo son:

- Producto matriz-vector;
- Resolución de un sistema de ecuaciones lineal triangular aplicando sustitución progresiva o regresiva a las k columnas de A o k filas de B, respectivamente;
- Producto matriz-matriz.

Adicionamente debe tenerse en cuenta que, cuando se suman dos bloques de rango bajo, se vuelve a comprimir el resultado utilizando la descomposición en valores singulares para descartar los valores más pequeños.

B. Dependencias entre las operaciones de la $\mathcal{H}$-LU

La Figura 3 representa las dependencias existentes entre las operaciones que componen la factorización $\mathcal{H}$-LU descrita en la Sección III para la matriz de ejemplo A. En concreto, puede observarse que la factorización se descompone inicialmente en 5 tareas (O1 - O5), y algunas de ellas, a su vez, se descomponen en subtareas (por ejemplo, la tarea O1 se descompone en 5 tareas: O1.1 - O1.5).

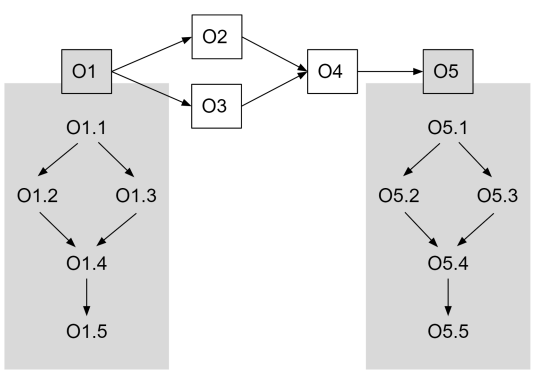

Fig. 3.Dependencias de datos entre las operaciones de la factorización $\mathcal{H}$-LU para el algoritmo descrito en la Sección III para la matriz de ejemplo A.

Aparentemente, el grado de paralelismo está limitado, dado que solamente podrían ejecutarse en paralelo O1.2 con O1.3, O2 con O3 y O5.2 con O5.3. No obstante, esto se debe al exceso de simplicidad de la matriz tomada como ejemplo. Las $\mathcal{H}$-matrices presentan particionados más complejos (es-

to es, anidándose varios particionados) y que no solamente afectan a los bloques diagonales. La Figura 4 presenta un ejemplo (todavía simple) de una $\mathcal{H}$-matriz alternativa a la presentada anteriormente.

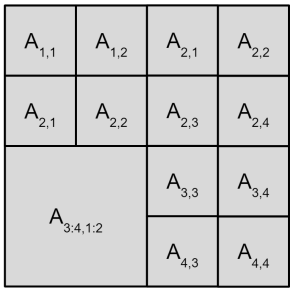

Fig. 4.$\mathcal{H}$-matriz alternativa con un particionado simple de $2x2$, dimensión 12.

En este nuevo ejemplo, la secuencia de operaciones que componen el algoritmo para la $\mathcal{H}$-LU descrito anteriormente es ahora más compleja. Concretamente, la operación

$$\text{O2}: \quad U_{1:2,3:4} \quad := \quad L^{-1}_{1:2,1:2} A_{1:2,3:4}$$

puede ahora subdividirse en las siguientes operaciones:

$$\begin{aligned}
\text{O2.1}: \quad U_{1,3} &:= L^{-1}_{1,1} A_{1,3}, \\
\text{O2.2}: \quad U_{1,4} &:= L^{-1}_{1,1} A_{1,4}, \\
\text{O2.3}: \quad A_{2,3} &:= A_{2,3} - L_{2,1} \cdot U_{1,3}, \\
\text{O2.4}: \quad A_{2,4} &:= A_{2,4} - L_{2,1} \cdot U_{1,4}, \\
\text{O2.5}: \quad U_{2,3} &:= L^{-1}_{2,2} A_{2,3}, \quad \text{y} \\
\text{O2.6}: \quad U_{2,4} &:= L^{-1}_{2,2} A_{2,4}.
\end{aligned}$$

Si bien O1 y O2 siguen presentando la dependencia de datos que ya se indicó en la Figura 3, sus respectivas subtareas tienen, entre ellas, parte de dichas dependencias (aunque no su totalidad), tal como se representa en la Figura 5.

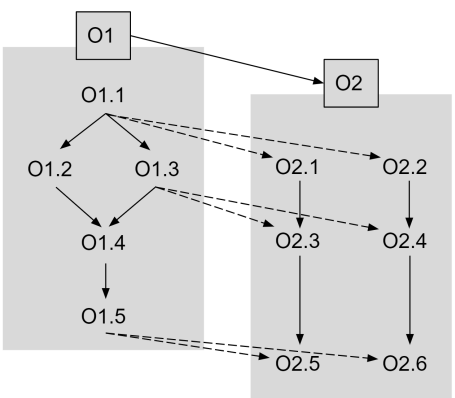

Fig. 5.Dependencias de datos entre las operaciones O1 y O2 de la factorización $\mathcal{H}$-LU para el algoritmo descrito en la Sección III para la nueva matriz de ejemplo (Figura 4). Con líneas discontinuas se indican las dependencias entre tareas "no hermanas" (no son subtareas de una misma tarea con granularidad inmediatamente superior).

Así pues, si en este escenario se aplica paralelismo anidado, tras la operación O1.1 pueden no solo realizarse O1.2 y O1.3, sino también O2.1 y O2.2.

C. *OmpSs-2 aplicado a la $\mathcal{H}$-LU*

Con los modelos de programación OmpSs (en su primera versión) y OpenMP 4.5, es necesario incluir un `taskwait` al final de cada tarea que se encuentra subdividida en subtareas para garantizar una correcta ejecución si se aplica paralelismo anidado. La ventaja de aplicar, en su lugar, el modelo de programación OmpSs-2 es que pueden anidarse las dependencias de tareas hasta alcanzar aquellas de grano más fino, anotándose como dependencias *débiles* todas aquellas relativas a tareas de grano grueso (en el ejemplo, las presentes entre O1 y O2, entre otras) y solamente como dependencias *fuertes* las existentes entre las tareas de grano más fino (en el ejemplo, O1.1 y O2.1, entre otras). De este modo, pueden solaparse ejecuciones pertenecientes a distintos niveles de la anidación de tareas y anticiparse parte de las ejecuciones en cuanto las dependencias estrictamente necesarias se satisfacen (y no todas las indicadas en las tareas de grano más grueso de las que son subtareas las que ya están listas para ejecutarse) gracias a la liberación temprana de dependencias.

V. Experimentos

En esta sección se describen los experimentos realizados para comprobar la eficiencia de la versión paralelizada de la $\mathcal{H}$-LU presente en H2Lib explotando paralelismo de tareas. Las $\mathcal{H}$-matrices utilizadas en los test pertenecen al contexto de BEM y, en particular, de la resolución de la ecuación de Laplace en $d \in \{2, 3\}$ dimensiones, con los núcleos computacionales de la forma

$$g : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}\,,$$

$$g(x,y) = \begin{cases} -\frac{1}{2\pi} \log \|x-y\|_2 & : d = 2, \\ \frac{1}{4\pi} \|x-y\|_2^{-1} & : d = 3. \end{cases}$$

Se ha utilizado aritmética de doble precisión IEEE 754 en un nodo del supercomputador MareNostrum 4, en el Barcelona Supercomputing Center, que contiene dos sockets Intel Xeon Platinum 8160, con 24 núcleos por socket y 96 Gbytes de memoria DDR4 RAM. Además, se han empleado `gcc` 4.8.5, Intel MKL 2017.4 (con las instrucciones AVX2 habilitadas), y OmpSs-2 (`mcxx` 2.1.0).

El criterio de admisibilidad utilizado es:

$$\max\{\operatorname{diam}(\mathcal{B}_t), \operatorname{diam}(\mathcal{B}_s)\} \leq \eta \operatorname{dist}(\mathcal{B}_t, \mathcal{B}_s)\,,$$

donde $\eta \in \mathbb{R}_{>0}$.

Nuestros experimentos demuestran los beneficios de utilizar las nuevas funcionalidades de OmpSs-2 (dependencias *débiles* y liberación temprana de dependencias) en el paralelismo de tareas anidado aplicado a la resolución de la $\mathcal{H}$-LU. Para esto, utilizaremos dos configuraciones de $\mathcal{H}$-matrices: una de dimensión 30K proveniente de un problema de BEM

con $d = 2$, y otra de dimensión 42K donde el problema de origen tiene $d = 3$. En ambas, el grado de *dispersión* está controlado con el parámetro η, que para el primer caso tomará los valores 0,25, 0,5 y 1, y, para el segundo, 0,5, 1, 2. La Figura 6 presenta las seis $\mathcal{H}$-matrices utilizadas.

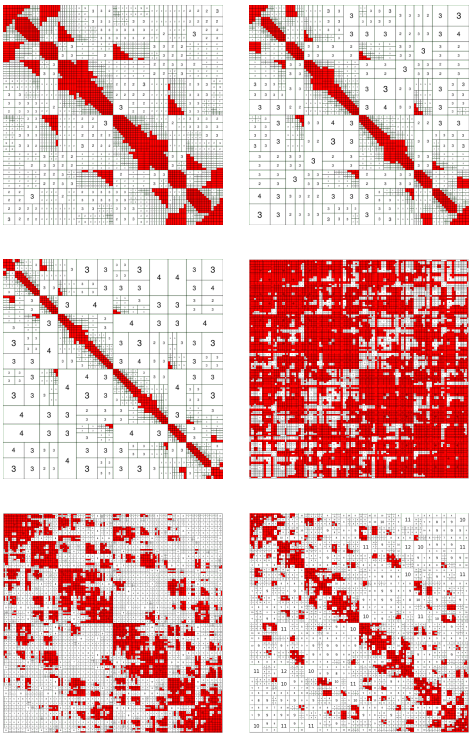

Fig. 6.$\mathcal{H}$-matrices utilizadas para los experimentos. De arriba a abajo y de izquierda a derecha, las tres primeras tienen dimensión 30K y los valores de η son 0,25, 0,5 y 1; las tres restantes tienen dimensión 42K y sus valores de η son 0,5, 1 y 2. Los bloques rellenados con color rojo son densos, el resto son bloques de rango bajo.

En los experimentos realizados con las matrices de dimensión 30K se utilizarán hasta 24 cores y, con las de dimensión 42K, hasta 48 cores. En todos los casos, se comparan los rendimientos paralelos utilizando dependencias *débiles* (WD) y sin utilizarlas. Las Figuras 7 y 8 presentan los *speedup* alcanzados en los experimentos realizados, en relación a la ejecución secuencial, representándose con líneas discontinuas los valroes correspondientes a las implementaciones en las que no se incluyen WD.

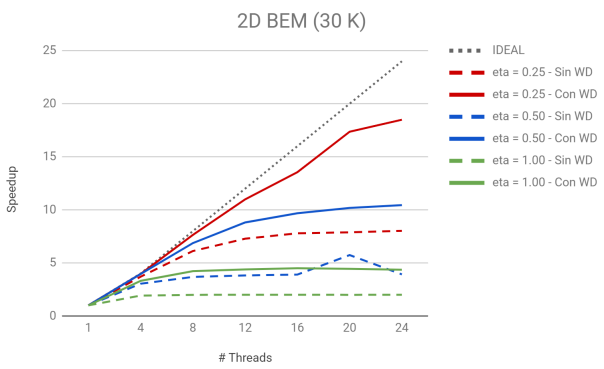

Fig. 7.Speedup de la ejecución paralela de la $\mathcal{H}$-LU con y sin dependencias débiles (WD), para las matrices de dimensión 30K con distintos valores de η y hasta 24 cores.

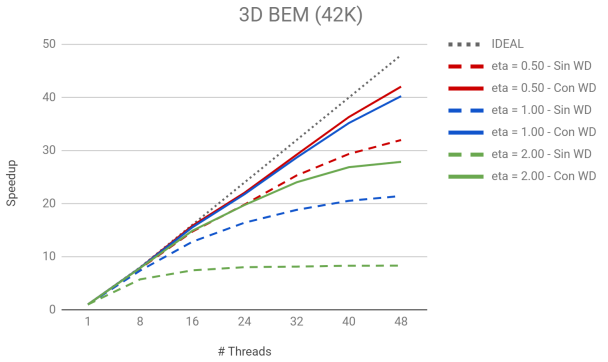

Fig. 8.Speedup de la ejecución paralela de la $\mathcal{H}$-LU con y sin dependencias débiles (WD), para las matrices de dimensión 42K con distintos valores de η y hasta 48 cores.

En general, se observa que el *speedup* aumenta con el ratio de bloques densos, reportándose valores notablemente altos para $d = 3$ (dado que la cantidad de núcleos no es demasiado alta comparada con la dimensión del problema) y menores para el caso $d = 2$. Estas diferencias en términos de eficiencia pueden justificarse analizándose la estructura de las $\mathcal{H}$-matrices. Concretamente, en el caso de $d = 3$, hay una mayor cantidad de bloques, lo cual aumenta el número de tareas a ejecutar y ayuda a exponer un mayor grado de concurrencia en ese caso.

VI. Conclusiones

Hemos mostrado eficiencias paralelas destacables para la factorización $\mathcal{H}$-LU de $\mathcal{H}$-matrices provenientes de BEM en 2D y 3D. Esto ha sido posible, en gran parte, gracias a utilizar las nuevas características presentes en OmpSs-2 (dependencias *débiles* y liberación temporana de dependencias), las cuales permiten explotar el paralelismo de tareas anidado. Como parte del trabajo futuro, queremos investigar esquemas de paralelismo híbridos que combinen paralelismo de tareas con paralelismo multihilo a nivel de núcleos computacionales, así como nuevas estrategias para extraer un mayor grado de paralelismo de tareas.

Agradecimientos

Los investigadores de la Universitat Jaume I (UJI) han sido financiados con los proyectos CICYT TIN2014-53495-R y TIN2017-82972-R del MINECO y FEDER; el proyecto UJI-B2017-46 de la UJI; y el programa FPU del MECD.

Referencias

[1] José I. Aliaga, Rosa M. Badia, María Barreda, Matthias Bollhöfer, and Enrique S. Quintana-Ortí, *Leveraging task-parallelism with OmpSs in ILUPACK's preconditioned cg method*, 26th Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD 2014), pages 262–269, 2014.

[2] José I. Aliaga, Rocío Carratalá-Sáez, Ronald Kriemann, Enrique S. Quintana-Ortí *Task-parallel LU factorization of hierarchical matrices using OmpSs*, 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017, pp. 1148–1157. doi:10.1109/IPDPSW.2017.124.

[3] Rosa M. Badia, Jose R. Herrero, Jesus Labarta, Jose M. Pérez, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. *Parallelizing dense and banded linear algebra libra-*

ries using SMPSs, Concurrency and Computation: Practice and Experience, 21:2438–2456, 2009.

[4] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. *A set of level 3 basic linear algebra subprograms*, ACM Trans. on Mathematical Software, 16(1):1–17, March 1990.

[5] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 3rd edition, 1996

[6] Lars Grasedyck and Wolfgang Hackbusch. *Construction and arithmetics of $\mathcal{H}$-matrices*, Computing, 70(4):295–334, August 2003.

[7] Wolfgang Hackbusch. *A sparse matrix arithmetic based on H -matrices. part i: Introduction to H -matrices*, Computing, 62(2):89–108, May 1999.

[8] Wolfgang Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*, volume 49 of Springer Series in Computational Mathematics. Springer-Verlag Berlin Heidelberg, 2015.

[9] Mohammad Izadi. *Hierarchical matrix techniques on massively parallel computers*, Ph.D. dissertation, Universität Leipzig, 2012.

[10] Ronald Kriemann. *H -LU factorization on many-core systems*, Computing and Visualization in Science, 16(3):105– 117, 2013.

[11] OmpSs project home page. *http://pm.bsc.es/ompss.*

[12] The OpenMP API specification for parallel programming. *http://www.openmp.org/.*