UNIVERSITAT JAUME I

ESCOLA DE DOCTORAT

Analysis of Parallelization Strategies in the context of Hierarchical Matrix Factorizations

Ph.D. in Computer Science

February 2021

Presented by:	Rocío Carratalá Sáez
Supervised by:	Enrique S. Quintana Ortí
	José Ignacio Aliaga Estellés

Programa de Doctorat en Informàtica

Escola de Doctorat de la Universitat Jaume I

Analysis of Parallelization Strategies in the context of Hierarchical Matrix Factorizations

Memòria presentada per Rocío Carratalá Sáez
per optar al grau de doctora per la Universitat Jaume I

Rocío Carratalá Sáez

Enrique S. Quintana Ortí
(Director)

José Ignacio Aliaga Estellés
(Codirector)

ROCIO|CARRATALA|SAEZ
Firmado digitalmente por ROCIO|CARRATALA|SAEZ
Fecha: 2021.02.24 16:54:25 +01'00'

ENRIQUE SALVADOR|QUINTANA|ORTI
Firmado digitalmente por ENRIQUE SALVADOR|QUINTANA|ORTI
Fecha: 2021.02.24 16:57:44 +01'00'

JOSE IGNACIO|ALIAGA|ESTELLES
Firmado digitalmente por JOSE IGNACIO|ALIAGA|ESTELLES
Fecha: 2021.02.24 17:16:24 +01'00'

Castelló de la Plana, Febrer de 2021

Contents

List of Figures

List of Tables

List of Acronyms

$\mathcal{H}$-Matrix Hierarchical Matrix. ix–xi, 3, 6, 21, 22, 40–50, 54, 62, 63, 66, 67, 71, 74, 75, 82, 83, 85–87, 92, 93, 95–97, 102, 105–107, 110

ACR Accelerated Cyclic Reduction. 5

AHMED Another software library on Hierarchical Matrices for Elliptic Differential equations. 5

API Application Programming Interface. 33

BCT Block Cluster Tree. 20–22, 39, 54

BDL Block Data Layout. 42, 48, 60, 75, 105, 115

BEM Boundary Element Methods. 2, 3, 5, 79, 80, 82, 85, 89, 92, 107, 109, 110, 112, 117, 118, 120, 122

BLAS Basic Linear Algebra Subprograms. 27, 28, 42, 45, 46, 48, 52, 54, 56, 60, 72, 77, 80, 81, 93, 96, 104, 105, 108, 109, 114, 116, 119

BLR Block Low-Rank. 3–5, 92, 126

BRL Block Right-Looking. ix, x, 44–50, 56, 60, 66, 67

BS Block Separable. xvi, 3, 4

BSC Barcelona Supercomputing Center. 34, 107, 109, 117, 120

CAP Computación de Altas Prestaciones (del inglés *High Performance Computing*). 121

CMO Column Major Order. ix, 27, 29, 41, 42, 44, 48, 60, 105, 115

CPU Central Processing Unit. 34

CT Cluster Tree. 20, 21, 62, 63, 94–96

DAG Directed Acyclic Graph. x, 51, 52, 72, 78

DoFs Degrees of Freedom. 62, 63

FEM Finite Element Methods. 2, 3, 5

flop Floating point operation (plural: *flops*). 37

FLOPS flops per second. 36, 37

flops Floating point operations (plural of *flop*). xiii, xvi, 4, 37, 38, 54, 56, 62, 80, 82, 85, 101

GNU GNU's Not Unix. 34

GPU Graphics Processing Unit. 5, 34, 112, 123

H2Pack High Performance $\mathcal{H}^2$-Matrix Package. 6

HBS Hierarchically Block Separable (BS). 4

HiCMA Hierarchical Computations on Manycore Architectures. 5

hm-toolbox Hierarchical matrix toolbox. 5

HODLR Hierarchical Off-Diagonal Low-Rank. 4, 5

HPC High Performance Computing. 110, 111, 121

HSS Hierarchically Semiseparable. 4, 5

IEEE Institute of Electrical and Electronics Engineers. 28, 54, 79, 108, 110, 111, 118, 120, 121, 125, 127, 129, 131

IMU Inertial Measurement Unit. ix, 1, 2

LAPACK Linear Algebra PACKage. 27, 28, 42, 45, 46, 48, 52, 60, 67, 68, 75, 77

LWS Locality Work Stealing. 101

MBLR Multilevel BLR. 4

MKL Math Kernel Library. x, 7, 27, 28, 32, 37, 52, 54, 56, 58, 59, 80, 81, 83, 97, 105, 116

MPI Message Passing Interface. 5, 35

PGNCS primary guidance, navigation, and control system. 1

RMO Row Major Order. 27, 29, 42

SPD Symmetric Positive Definite. 60

SPU Synergistic Processing Unit. 34

STF Sequential Task Flow. 34

STRUMPACK STRUctured Matrix PACKage. 6

SVD Singular Value Decomposition. 15, 17, 38, 46, 65, 68, 79, 110, 121

TBB Thread Building Blocks. 5, 7, 71

TSMC Taiwan Semiconductor Manufacturing Company. 32

ViTE Visual Trace Explorer. 36

WS Work Stealing. 101

“Aprendí que el coraje no era la ausencia
de miedo, sino el triunfo sobre él.
El valiente no es quien no siente miedo,
sino aquel que conquista ese miedo.”

Nelson Mandela

Para ti, abuelo,
por tu abrazo y tu “todo” a cambio de mi “nada”.

Y para mi *peque*, Pelusa,
por darlo todo hasta el final.

Summary

$\mathcal{H}$-Matrices were born as a powerful numerical tool to tackle applications whose data generates structures that end laying in between dense and sparse scenarios. The key benefit that makes $\mathcal{H}$-Matrices valuable is the savings that offer both in terms of storage and computations, in such a way that they are reduced to log-linear costs.

The key behind the success of $\mathcal{H}$-Matrices is the *controllable compression* they offer: by choosing the appropriate admissibility condition to discern important versus dispensable data and designing good partitioning algorithms, one can choose the accuracy loss that wants to assume in exchange for computations acceleration and memory consumption reduction. This is the reason why $\mathcal{H}$-Matrices are specially suitable for boundary element and finite element methods where the pursued result does not need to be totally accurate, but it is important to have it ready as fast as possible, as it can determine, for example, whether an engineering design is ready to be produced or needs to be improved.

On their side, task-parallelism has proved sufficiently its benefits when being employed to optimize the parallel execution when solving linear systems of equations. Particularly, tiled or block algorithms combined with this parallelism strategy have widely been (and are still) employed to provide the scientific community with powerful and efficient parallel solutions for multicore architectures.

The main objective of this thesis is designing, implementing and evaluating parallel algorithms to operate efficiently with $\mathcal{H}$-Matrices in multicore architectures. To this end, the first contribution we describe is a study in which we prove that task-parallelism is suitable for operating with $\mathcal{H}$-Matrices, by simplifying as much as possible the $\mathcal{H}$-Arithmetic scenario. Next, we describe in detail the difficulties that need to be addressed when parallelizing the complex implementations that operate with this type of matrices. Afterwards, we explain how the new features included in OmpSs-2 programming model helped us avoiding the majority of the described issues and thus we were able to attain a fair efficiency when executing a task-parallel $\mathcal{H}$-LU. Lastly, we illustrate how we explored a *regularized* version of $\mathcal{H}$-Matrices, which we call Tile $\mathcal{H}$-Matrices, in which we are able to maintain competitive-with-pure-$\mathcal{H}$-Matrices precision and compression ratios, while leveraging the well known benefits of tile algorithms applied to matrices provided with (regular) tiles (this is, mostly homogeneous block dimensions).

Resumen

Las $\mathcal{H}$-Matrices nacen como una potente herramienta numérica para abordar aplicaciones cuyos datos generan estructuras que se sitúan entre los escenarios densos y dispersos. El beneficio clave por el que las $\mathcal{H}$-Matrices son valiosas es el ahorro tanto en términos de almacenamiento como de cómputo, de un modo tal que llegan a reducirse hasta un coste logarítmico-lineal.

La clave del éxito de las $\mathcal{H}$-Matrices reside en la *compresión controlable* que ofrecen: eligiendo la condición de admisibilidad adecuada para separar los datos importantes de los prescindibles, así como un buen algoritmo de particionado, puede controlarse la pérdida de precisión que se quiere asumir, a cambio de acelerar los cálculos y reducir el consumo de memoria. Esta es la razón por la que las $\mathcal{H}$-Matrices son especialmente apropiadas para métodos de elementos de contorno y elementos finitos, donde el resultado que se persigue no es necesario que sea totalmente preciso, pero sí importa disponer del mismo cuanto antes, dado que puede determinar, por ejemplo, si un diseño de ingeniería está listo para ser producido o necesita ser mejorado.

Por su parte, el paralelismo de tareas ha demostrado con creces sus beneficios cuando se utiliza para optimizar las ejecuciones paralelas al resolver sistemas de ecuaciones lineales. Particularmente, los algoritmos por bloques o *tiles*, combinados con esta estrategia de paralelismo, han sido ampliamente utilizados (y lo siguen siendo) para proveer a la comunidad científica de soluciones paralelas potentes y eficientes en arquitecturas multinúcleo.

El objetivo principal de esta tesis es diseñar, implementar y evaluar algoritmos paralelos para operar de un modo eficiente con $\mathcal{H}$-Matrices en arquitecturas multinúcleo. Con este objetivo en mente, la primera contribución que describimos es un estudio en el que demostramos que el paralelismo de tareas es apropiado para operar con $\mathcal{H}$-Matrices, simplificando para ello, tanto como nos es posible, la Aritmética con $\mathcal{H}$-Matrices. A continuación, describimos en detalle las dificultades a las que se debe hacer frente cuando se paralelizan las complejas implementaciones que sirven para operar con estas matrices. Tras esto, explicamos cómo nos ayudaron las nuevas funcionalidades del modelo de programación OmpSs-2 a sortear la mayoría de dichas cuestiones, para así llegar a alcanzar una buena eficiencia al ejecutar nuestra implementación de la $\mathcal{H}$-LU basada en paralelismo de tareas. Finalmente, explicamos cómo hemos explorado el diseño e implementación de una versión *regularizada* de las $\mathcal{H}$-Matrices, a la que denominamos Tile $\mathcal{H}$-Matrices, en la que somos capaces de mantener un ratio de precisión y compresión competitivo con el proporcionado por $\mathcal{H}$-Matrices puras, a la vez que aprovechamos los beneficios de los algoritmos por bloques, ampliamente

conocidos y utilizados en matrices particionadas en bloques (regulares), es decir, que presentan tamaños de bloque bastante homogéneos.

Agradecimientos

Si volviera atrás, tomaría la misma decisión: hacer esta tesis. Sabiendo lo que sé evitaría algunos errores, pero, como dijo Einstein, "*Si supiéramos lo que estábamos haciendo, no se llamaría investigación, ¿verdad?*". Lo que no cambiaría bajo ningún concepto sois vosotros, mis compañeros de viaje, a los que os dedico estas líneas cargadas de mis más sinceros agradecimientos. A todos (incluidos quienes haya olvidado mencionar), gracias por acompañarme, dejarme formar parte de vuestras vidas y permitirme aburriros con mis $\mathcal{H}$-Matrices.

En lo profesional, en primer lugar, gracias a los profesores y compañeros del grupo de investigación HPC&A de la UJI: José Manuel Badía, Sergio Barrachina, Asun Castaño, Maribel Castillo, Juan Carlos Fernández, Germán León, Merche Marqués, Rafa Mayo y Andrés Tomás. Gracias por vuestros consejos, por el apoyo en la docencia, por las sonrisas por los pasillos y por ayudarme cuando lo he necesitado. A David, Íker, Manel, Mar y Nacho, las "nuevas incorporaciones", gracias por llegar cuando los otros *canallas* me estaban dejando sola en los almuerzos, comidas y frustraciones; ha sido un placer recorrer el final del camino contando con vosotros. Como codirector, gracias José Aliaga por tu ayuda, especialmente cuando apretaban los plazos, la burocracia y la tesis en general. Un paso más allá de lo profesional, como amigos, gracias a Adrián, Maria, Sandra y Sergio.

Adrián i Sergio: esteu com una cabra. No he conegut mai ningú com vosaltres: no m'heu deixat parar, ni parar de riure. Heu resolt tots els dubtes que vos he plantejat durant el doctorat, m'heu fet part dels projectes que teníeu al cap, i em féreu molt fàcil des del principi ser un membre més del grup. Tant de bo col·laborem sempre al terreny professional i seguim compartint bons moments al terreny personal. Maria, vas ser sempre la calma que el despatx i jo necessitàvem, el suport tranquil i amable, a qui confiar-li els problemes amb les implementacions i aquells que no estaven tan relacionats amb elles; quin plaer compartir espai amb tu! Ostrava no va ser el millor viatge, ni el meu pronòstic inicial sobre Bordeus el més encertat, però eixos i tots els altres moments compartits, els tornaria a viure amb tu amb els ulls tancats. Sandra, "Big-Little" ja és per sempre, ara ja no t'alliberes de mi! Gràcies per les confessions al despatx, els consells, els debats intercanviats sobre les meues $\mathcal{H}$-Matrius i els teus algoritmes a blocs, les col·laboracions que ja hem fet realitat i les que espere que arriben i per gaudir la màgia d'Orlando amb mi. Potser per què ens assemblem en moltes coses, o potser precisament per les que ens diferencien, vosaltres quatre sou una de les millors conseqüències de la meua tesi.

Still in the professional environment, Steffen Börm and Sven Christophersen: thanks for your support and assistance while I visited you in Kiel; you helped me understand H2Lib and the $\mathcal{H}$-Arithmetic.

També al camp professional, gràcies a l'equip del BSC que em va acollir durant un mes, i molt especialment a J. M., Vicenç Beltrán, i Xavi Teruel pel temps invertit a ajudar-me a entendre OmpSs-2 i com utilitzar-lo.

Once more between the professional and the friendship environment, thanks to everyone I met at Inria - Bordeaux. HiePACS team always made me feel at home, so thanks to Abdou, Alena, Esragul, Giles, Lionel, Luc, Mathieu V., Nic, Olivier, Pierre, Romain, and Tony. Emmanuel, thanks for saying "yes" to my first visit, and for always having your smile ready. Guillaume, understanding Hmat/Hmat-oss would not have been possible without you. Grégoire, your help, support and friendship was crucial to make $\mathcal{H}$-Chameleon a reality. Mathieu, I think of you as my third co-advisor, and there is no big enough "thank you" I can write to express my gratitude for your patience, assistance with StarPU, Chameleon, and every extra hour you spent with me; I learned a lot thanks to you. HiePACS: there is a little portion of me staying at Bordeaux, living inside a "bouteille de vin", staying at an "Apéro sur la plage". Merci beaucoup!

A Vicent Palmer: gràcies per acceptar ser qui em va introduir en el món de la investigació i per la teua generositat i sinceritat quan em plantejava fer aquest doctorat en informàtica.

Ja en el terreny purament personal, mil gràcies als meus amics: el vostre suport també ha sigut importantíssim per fer aquesta tesi realitat. Sé que m'he perdut moltes coses, i que no vos he fet partícips d'altres meues; mil gràcies per entendre que, a voltes, les Matrius Jeràrquiques foren les úniques per qui tenia temps.

Als de sempre, Ángel, David, Judith, Katia, Mario, Mirella, Pau i Raquel: prompte tornaran les pizzes compartides i els records damunt la taula plena amb tots nosaltres. Això sí, amb un nou membre molt especial a qui tots desitgem conéixer ja: la xicoteta Laia! A mi ja em cau la bava i encara no la conec.

A Alba i Adri, que no sou de sempre, però espere sí per sempre, també vos dec el temps que vos he llevat d'estar junts; tornaran les nits de vins i xerrar sense mirar el rellotge. Hem viscut moments molt importants durant aquesta tesi, espere que ens queden molts per compartir!

Amanda y David, vosotros tampoco sois de siempre, pero espero que hayáis llegado para quedaros. Las noches de Dungeons & Dragons nos han salvado la cuarentena y vuestra experiencia previa en esto de hacer una tesis ha puesto cordura donde, por momentos, solamente había caos.

A mi familia (la de siempre y la política), gracias por comprender lo que no siempre he sabido explicar y por arroparme, cada uno a vuestra manera. Os debo también parte de la cordura que me ayudasteis a mantener y tardes, comidas y cenas que no hemos podido planificar. A las nuevas incorporaciones: Merlín y Koba, bienvenidos y gracias por vuestro cariño y por entender tan rápido mi estilo de vida. A Víctor y María Ángeles: gracias por hacerme sentir como en casa y por hacer de Sergio el chico que me enamoró. A Lucía, gracias por acogerme como una hermana; prepárate, cuando no haya restricciones vamos a ir a que nos enseñes todo lo que conoces de Albacete. A tu iaia, gràcies per què amb noranta-quatre anys segueixes al peu del canó i em poses contra les cordes quan em demanes explicar-te què estic investigant. A Pepe: el tuyo fue el primer ordenador que toqué, así que si a eso le unimos todas las tardes juntos desde pequeña y tu cariño incondicional; nadie puede discutir que tienes parte de culpa en esta tesis y a nadie (ni siquiera a ti) le voy a dejar negar que seguiremos ganando batallas. A ti, abuela, gracias por los chocolates a media tarde, mis cenas y comidas favoritas, tus caricias en forma de golpecillos, la manera con la que me cogías la mano y cómo me entendías desde tu calma y paciencia. También, gracias por dejarle brillar estando siempre ahí y por habernos protegido a ambos, aunque quizá no siempre supimos darnos cuenta. Qué lujo haberte tenido como abuela.

Y para quienes venís ahora son mis agradecimientos más especiales, porque esta tesis nunca se habría hecho realidad sin vosotros como pilares donde apoyarme cuando había un traspiés.

A ti Enrique, mi director. En mi entorno suelo referirme a tí como mi *gurú* y, ahora que no hay ya riesgo de *peloteo*, puedo hablarte con sinceridad. Me has permitido conocer a muchos otros profesores e investigadores en las innumerables veces que he recorrido ciudades por todo el mundo para ir contando mis avances con las $\mathcal{H}$-Matrices, y no me he econtrado a nadie con quien prefiriera haber estado para guiarme en esta tesis, ni con quien me gustaría colaborar más que contigo durante mi vida investigadora futura. Siento que a veces te he decepcionado, y dejo por escrito mis disculpas y la promesa de que de esos errores aprenderé y lo haré mejor en el futuro. Ojalá como investigadora llegue algún día a la altura de tus talones; eso querrá decir que lo estaré haciendo bien. Si algún día lidero un proyecto, un equipo o cualquier otra cosa, me conformaría con tener la mitad de tu capacidad de empatizar, tu visión y tu productividad. Gracias por estar siempre disponible, por tus explicaciones, tu apoyo y tu comprensión.

A ti, Sergio, que te colaste en mi vida hace ya casi diez años, aunque lo nuestro empezara hace unos siete. Embarcarnos en la aventura de la tesis a la vez y escribirla casi al mismo tiempo junto con una cuarentena que nadie hubiésemos esperado ha sido, cuanto menos *interesante*, incluso a veces *divertido.* Gracias por compartir tu pasión conmigo y por tu exceso de comprensión: sé que no te lo pongo fácil. Gracias por ser un remanso de paz y calma en todo el caos que yo suelo representar y gracias por, como dijiste una vez, "vivir un sueño conmigo". Te quiero.

A vosotros, papá y mamá, gracias por ser mi garantía. Contar con vosotros significa no perder el rumbo, encontrar el suelo si los pies se despegan de él, o echar a volar cuando la gravedad aprieta fuerte. Siempre he sentido que mis sueños contaban con vuestro apoyo y, lo más importante, con vuestra cordura. Ojalá cuando yo ocupe vuestro lugar consiga transmitir la seguridad, cariño y estabilidad que vosotros me habéis hecho sentir. Papà, ningú estabilitza el meu caos com tu quan em calmes si em guanya el desassossec, t'estime; mamá, nadie junta mis piececillas rotas como lo haces tú cuando me abrazas, te quiero. Gràcies per tot, perdó pels meus pitjors moments.

A Pelusa, *la meua "peque"*, gràcies per donar-ho tot fins al final. Et van faltar dos mesos per veure'm posar el punt final a aquesta tesi. Però, allò que no et vas perdre va ser cap de les nits anteriors al nostre comiat, aquelles on jo escrivia mentre tu, tot i l'epilèpsia, el dolor de cap i l'esgotament, et gitaves silenciosa al meu costat, cuidant que la nit no vencera les meues ganes. Vam viatjar per quatre països diferents, vam estudiar juntes des de primer de Batxillerat, ho vam fer tot fent-nos costat. Va arribar un dia en què et vaig haver de prometre que, com diu Txarango, podries comptar amb mi a l'últim sospir de la nit, que jo t'espantaria els malsons fins que se't curaren les ales. Fins sempre, ha sigut un plaer, espere haver-te fet tan feliç com tu a mi.

Y por último, por supuesto, por derecho y porque sobran los motivos, a ti, abuelo. Como ya he dicho, me diste tu *todo* a cambio de mi *nada.* Recuerdo ver en tus ojos el miedo cuando te contaba cada plan de futuro, mientras me sonreías y animabas a conseguir mis sueños; recuerdo la fe que depositabas en cada paso que yo daba y el empujón que siempre tenías listo cuando estaba a punto de caer; recuerdo los cuentos propios y los clásicos, que algunas noches aún vuelvo a protagonizar transportándome debajo de aquellas sábanas infantiles; recuerdo tu voz, tus comentarios cascarrabias y que para mí nunca hubo una réplica, ni siquiera cuando me echabas de menos más que nadie. Y recuerdo, no sabes cúanto, tu abrazo. Gracias por haber sido mi abuelo. Si en mi vida logro ser la mitad de lo que eras tú, entonces todo habrá tenido sentido.

Chapter 1

Introduction

Contents of the chapter

1.1 Motivation

"*Okay, Houston, we've had a problem here.*"

(Apollo 13 Command Module Pilot John L. "Jack" Swigert)

That one is possibly one of the most known, reused, and translated quotes from last century. Swigert pronounced those words [16] when he noticed warning lights alerting a drop in voltage of one of the electrical buses supplying power to the Command and Service Module. Thankfully, what could have been a disaster had a happy ending.

However, that is not the only case in which the Apollo expeditions suffered emergency situations. A year earlier, in 1969, Apollo 11 [15] was launched into space; the first expedition that landed humans on the Moon. During the trip, a Gimbal Lock occured [14, 65]. The Apollo spacecraft was provided with an inertial guidance system named Apollo primary guidance, navigation, and control system (PGNCS), and one of its components was an Inertial Measurement Unit (IMU) gimbaled on three axes. The Gimbal Lock phenomena is caused by the superposition of two of the three axes that are employed in a gyroscope system (such as the IMU) to represent the orientation of an object as the combination three axial rotations with Euler Angles (see Figure 1.1). For example, if the axis associated to rotations with respect to the z-axis and the x-axis overlap, then the object *is told* to move in opposite ways along the same direction, and thus behaves unpredictably, causing it to lurch.

The Apollo 11 Gimbal Lock situation forced the pilot to manually control the whole movement of the aircraft, yet fortunately he did it successfully. In any case, it was definitely a hard situation that could have been avoided (and was in fact solved for posterior Apollo expeditions). The solution

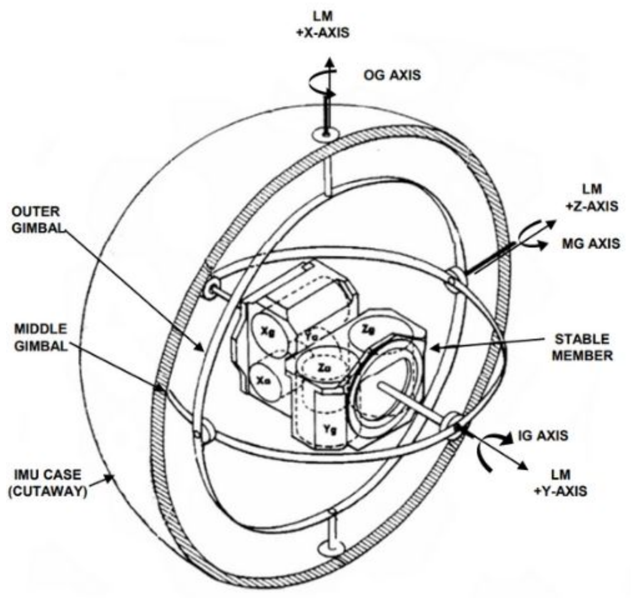

Figure 1.1: Apollo IMU. Source: `https://apollo11space.com/apollo-and-gimbal-lock/`.

to avoid this issue is simple: by adding a fourth gimbal, the redundancy that derives serves to prevent the system from entering in a Gimbal Lock situation. Although the engineers in charge of the Apollo 11 design knew the existence of this phenomena, they considered that there was no need to include redundancy, as mathematically it is enough to employ three gimbals to cover the three degrees of freedom that define the domain in which the object develops its movements, and they *underestimated* the probabilities of suffering a Gimbal Lock.

The lessons learnt from these and other human mistakes, miscalculations, and real-life situations underestimations justify by far the need of simulating as much as possible every situation and phenomena one can think of before building a true system. As a result, computer simulations, together with the numerical methods that support them, have become essential in most engineering applications.

This thesis puts the focus on offering competitive open source algorithms to operate with $\mathcal{H}$-Matrices, where the relevance of this special type of matrices comes from two numerical methods in which they are employed: Boundary Element Methods (BEM) [72] and Finite Element Methods (FEM) [23]. In brief, BEM and FEM serve to solve linear partial differential equations formulated as integral equations. Particularly, FEM is employed when modeling the impact of physical elements, such as temperature or pressure, and thus utilized in fields such as structural analysis, as well as heat transfer and fluid flow, among others. BEM is employed in a wide range of Physics-related fields, such as fluid mechanics, acoustics, and electromagnetism, for example. Usually, BEM and FEM results are employed to *stress* the systems in the simulations, this is, to take different components to the limit and see how they behave, with the aim of performing all the necessary adjustments and tests before building the actual system.

As it will be properly explained in later chapters, $\mathcal{H}$-Matrices allow to speed up computations, as well as save memory, in exchange for a controlled loss of accuracy. Computationally, this means that the results from the operations are obtained earlier, at a smaller memory cost, while guaranteeing a certain (acceptable) fixed precision. This is the reason why, although *mathematically young* (they were first defined in late 90s by Prof. Wolfgang Hackbusch), they are nowadays widely employed in BEM and FEM related computations. There is a large variety of applications in which $\mathcal{H}$-Matrices are employed to simulate different physical phenomena: elasticity [24, 119], geodesy [107], geostatistics [108], wave propagation [67, 82, 83], neural networks [44], acoustics [85], electronics and electromagnetics [73, 77, 106, 118], earthquake studies [97], etc. In addition, $\mathcal{H}$-Matrices are also leveraged to fasten the computation of certain mathematical operations, such as likelihood functions [47, 84], preconditioning [25, 26, 70], or randomization and stochastic problems [8, 18, 19, 75]. Moreover, some companies posses their own proprietary packages to operate with $\mathcal{H}$-Matrices, such as Airbus does with Hmat/Hmat-oss [64, 86], a software which will be later described in this document.

Prior to this thesis, there existed packages (see Section 1.2) that offered some degree of parallelism, but either their efficiency was limited, or they were proprietary and thus they could not be employed by the general scientific community. $\mathcal{H}$-Matrices present particular characteristics, such as a nested structure of blocks and the combination of dense and low-rank blocks. These features (and additional ones that will be properly detailed later in the document) pose certain programming difficulties, such as special data layout needs and data size variations during the solution of certain operations. Precisely, these difficulties limit the degree of parallelism that can be extracted by employing *classic* parallelism strategies and programming models.

In this PhD dissertation we study how to overcome the mentioned difficulties analysing two different perspectives: 1) the usage of alternative programming models that include special features (particularly OmpSs-2, which provides the option of defining *weak* task dependencies as well as an early release of tasks); and 2) a re-definition of the Hierarchical Matrix ($\mathcal{H}$-Matrix) classic structure to build Tile $\mathcal{H}$-Matrices, by means of which we leverage both the benefits of regular tiling benefits when employing classic parallel strategies, and the storage/computations savings that characterize $\mathcal{H}$-Matrices.

1.2 State-of-the-Art

In Chapter 2 we present several matrix classifications according to their dimension or data distribution, as well as all the necessary linear algebra background. However, to analyse the state-of-the-art for $\mathcal{H}$-Matrices, in this section we first provide a revision of the different matrix formats that can be used to store the data in a compressed form, in addition to $\mathcal{H}$-Matrices. Moreover, we also offer a summary of the main linear algebra software leveraging these matrix representations.

All the linear algebra terms employed in this section (LU/Cholesky Decomposition, block of a matrix, admissibility, etc.) are defined in Chapter 2 and can be consulted there if necessary.

1.2.1 Compressed representation of matrices

The main types of matrices which represent the data in a compressed form can be classified into three main groups, according to the cost of computing an LU (or Cholesky) decomposition [49, 56, 59, 89]:

- Quadratic cost: Block Low-Rank (BLR) and Block Separable (BS) matrices.

- Log-linear cost: Hierarchical Matrices ($\mathcal{H}$-Matrices) and Hierarchical Off-Diagonal Low-Rank (HODLR) matrices.
- Linear cost: $\mathcal{H}^2$-Matrices, Hierarchically Semiseparable (HSS) matrices, and Hierarchically BS (HBS) matrices.

The definition 2.34 in Chapter 2 introduces the term *low-rank matrix.* The BLR format [10] can be viewed as a simplified version of $\mathcal{H}$-Matrices, in the sense that a BLR matrix is partitioned into regular blocks (that is, blocks of the same size) which can be either low-rank or full-rank, according to a fixed admissibility condition. The BS format [42, 49, 50] can be viewed as a particular case of BLR in which a weak admissibility condition (see Definition 2.49 in Chapter 2) is chosen, and thus all the off-diagonal blocks are compressed as low-rank blocks. Matrix factorizations such as the LU or Cholesky decompositions can be performed over BLR/BS matrices with a complexity of $O(n^2)$ (quadratic) floating point operations (flops), instead of $O(n^3)$ (cubic) flops as is the case for dense matrices.

As it will be exposed in definition 2.52 in Chapter 2, $\mathcal{H}$-Matrices [28, 53, 55, 56] present a considerably more complex structure than BLR matrices. If a strong admissibility condition (see Definition 2.48 in Chapter 2) is utilized, the recursive partition of the blocks that conform $\mathcal{H}$-Matrices is kept until the refinement is sufficiently small to compress the data (forming low-rank blocks) or the inadmissible data can be kept in the original format (this is, constituting a dense block). When a weak-admissibility condition is employed, equivalent to BS formats, the off-diagonal blocks are directly compressed and the derived matrix follows the HODLR format [12], which can be regarded as the counterpart of $\mathcal{H}$-Matrices. In these scenarios, the LU or Cholesky decompositions are computed with a near-linear complexity, concretely $O(n\, k \log^q n)$ flops, where q is a small integer which depends on the particular factorization that is computed, and k can be tuned to configure the local rank of $\mathcal{H}$-Matrices blocks and thus control the accuracy of the approximation.

$\mathcal{H}^2$-Matrices are more complex structures and remain out of the scope for this thesis. Their purpose is to remove the logarithmic factor in the complexity costs of $\mathcal{H}$-Matrices and HODLR formats by using a nested-basis structure. Again, the type of admissibility condition determines which structure is obtained: a strong admissibility criteria yields $\mathcal{H}^2$-Matrices, while the weak admissibility conditions generate HSS [39, 115] or HBS [50] matrices. By exploiting nested basis structures, the factorizations can be performed with a linear complexity, this is $O(n)$ flops.

Newer structures, such as Multilevel BLR (MBLR) [11] or Lattice $\mathcal{H}$-Matrices [68, 116], leverage the best features of BLR and $\mathcal{H}$-Matrices formats to achieve a close-to-linear complexity (as $\mathcal{H}$-Matrices do), while simplifying the matrix structure by employing a more regular format and less levels in the hierarchy, as the one presented with BLR. However, there are still no software packages that provide efficient implementations of these formats.

1.2.2 Linear algebra software for compressed matrices

A wide variety of linear algebra software is available to compute operations over the compressed formats introduced in the previous section, as described in this section.

The following are some of the most relevant packages available to solve linear algebra operations involving BLR matrices:

- *CHOLMOD* [41] offers rank-structured algorithms to solve positive definite linear systems arising from discretized partial differential equations. Particularly, it is a supernodal solver which uses randomization with fixed rank.

- *MUMPS-BLR* [89] employs low-rank approximations to reduce the cost of sparse direct multifrontal solvers. The implementations included in *MUMPS-BLR* can be also efficiently executed in shared and distributed memory systems.

- PaStiX *BLR* [102, 103] provides BLR compression techniques to compute the sparse supernodal solver PaStiX, either as a direct solver operating at a lower precision or as a preconditioner. The implementations included in PaStiX *BLR* can be executed in shared memory parallel platforms.

The following list includes the most important packages targeting linear algebra operations involving $\mathcal{H}$-Matrices:

- *H2Lib Package* [54] (and the former one *HLib Package [63]*) provides a whole set of routines written in `C` for $\mathcal{H}$-Matrices and $\mathcal{H}^2$-matrices operations, including application modules for FEM and BEM in 2D and 3D. This package is based on OpenMP and can exploit a limited degree of shared memory (i.e. multithreaded) parallelism.

- *$\mathcal{H}$-Lib^{pro}* [91] is a `C++` software library that implements algorithms for $\mathcal{H}$-Matrices operations. The binaries are available for free only for academic purposes. It leverages Intel Thread Building Blocks (TBB) and Message Passing Interface (MPI) to exploit multithreaded servers and distributed memory parallelism.

- *HACApK* [69] is a software mainly written in Fortran that comprises routines to solve $\mathcal{H}$-Matrices operations, especially optimized for Graphics Processing Units (GPUs).

- *HMAT* and *HMAT-oss* [64, 86] correspond to the Airbus proprietary `C/C++` library for $\mathcal{H}$-Matrices, and the subset of its sequential routines, respectively. The proprietary version of this software can efficiently compute $\mathcal{H}$-Matrices operations in multithreaded servers and distributed memory platforms.

- *Hierarchical Computations on Manycore Architectures (HiCMA)* [60] focuses on the underlying tile low-rank data format of $\mathcal{H}$-Matrices to provide efficient implementations of the matrix product and Cholesly Factoriztion of $\mathcal{H}$-Matrices, as well as the LU factorization in tile low-rank format, both in multithreaded servers and distributed memory environments using StarPU and MPI.

- *Accelerated Cyclic Reduction (ACR)* [40] is a software to realise a distributed-memory fast solver for rank-compressible block tridiagonal systems arising from the discretization of elliptic operators. It combines cyclic reduction with hierarchical matrices.

- *Another software library on Hierarchical Matrices for Elliptic Differential equations (AHMED)* [13] is a C++ software library that implements $\mathcal{H}$-Matrices for the efficient treatment of discrete solution operators for elliptic boundary value problems.

Even though they are not directly related to the work presented in this thesis, there are also other software packages to operate over matrices in HODLR and HSS formats, such as *Hierarchical* *matrix toolbox (hm-toolbox)* [90] in MATLAB, *STRUctured Matrix PACKage (STRUMPACK)* [48], and *HODLRlib* [9]. Moreover, the main software packages to treat $\mathcal{H}^2$-Matrices are *H2Lib*, *High Performance $\mathcal{H}^2$-Matrix Package (H2Pack)* [61], and *LoRaSp* [87].

1.3 Objectives

The main goal set for this thesis was the design, implementation and evaluation of realizations of $\mathcal{H}$-Arithmetic operations capable of being efficiently executed in multithreaded architectures by leveraging task-based parallelism strategies.

These are the main (sub-)objectives pursued to achieve the main objective:

- Study and analyze $\mathcal{H}$-Matrices and the associated $\mathcal{H}$-Arithmetic. $\mathcal{H}$-Matrices lay in between the dense and sparse scenarii, due to their combination of dense and low-rank blocks. For this reason, we will study them, as well as the main $\mathcal{H}$-Arithmetic operations and existing implementations, with the purpose of identifying the features that condition the parallel strategies that can be employed.
- Test the suitability of task-parallelism for the $\mathcal{H}$-LU. Task-based parallelism is gaining importance and recognition, as it usually helps to decouple the mathematical details of the algorithms from the effective collections of (sub-)operations that conform larger ones with their own identities. Moreover, task-parallelism strategies allow to easily specify/capture data dependencies between tasks in such a way that this job can be off-loaded to a runtime, which performs it transparently for the programmer. One can imagine that task-parallelism is also suitable for operations with $\mathcal{H}$-Matrices, but that assumption needs to be evaluated and proved, and we need to demonstrate this by defining prototype $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky operations and parallelizing them using OpenMP/OmpSs tasks.
- Design, develop, and test an efficient task-parallel $\mathcal{H}$-LU. Once we can ensure that task-parallelism is suitable in this context, we will parallelize the $\mathcal{H}$-LU in H2Lib to validate the task-parallelism benefits for this operation.
- Design, develop and test an efficient task-parallel open source $\mathcal{H}$-Package. The epitome of this thesis will be the assembly of an open source package capable of (parallel) efficiently operating with $\mathcal{H}$-Matrices. To this end, we will present $\mathcal{H}$-Chameleon, which is based on the Chameleon [2, 3, 38] and Hmat-oss[64, 86] existing packages, as well as the StarPU runtime system [17, 110], together with a special data structure named Tile $\mathcal{H}$-Matrix that combines the benefits from $\mathcal{H}$-Matrices and also from regular tiles partitionings.

1.4 Related work

In this section, we provide a brief summary of the main works related to the objectives of this thesis. The connection of each work with the two data structures mostly employed in this dissertation ($\mathcal{H}$-Matrices or Tile $\mathcal{H}$-Matrices) is exposed, as well as the similarities and/or differences in terms of the parallelization strategy.

Along this dissertation, different parallelization strategies for $\mathcal{H}$-Matrix factorizations are presented. The packages whose factorizations have been parallelised are H2Lib [54] and $\mathcal{H}$-Chameleon.

The first one is an open-source library that implements the original $\mathcal{H}$-Arithmetic defined in [55]. The later one is a combination of two existing packages: Chameleon [2, 3, 38] and Hmat-oss [64, 86]. Moreover, the programming models utilized for the task-based parallelization are OpenMP, OmpSs/OmpSs-2, and StarPU. Intel Math Kernel Library (MKL) is also used in this dissertation, but mainly for some efficiency comparisons that always offer higher performance when using task-parallelism based approaches.

Prior to this dissertation, the H2Lib package already provided some *limited* degree of parallelism, thanks to the use of OpenMP simple parallel structures, such as parallel loops. With the work presented in this dissertation, the parallel performance achieved by the originally parallelised $\mathcal{H}$-LU in H2Lib is overcome, thanks to leveraging a task-based parallel implementation that utilizes special features from the OmpSs-2 programming model to expose a higher degree of concurrency.

Chameleon [2, 3, 38] is originally a dense linear algebra package, and this dissertation enhances it with an extension that allows operating with hierarchical structures (particularly, Tile $\mathcal{H}$-Matrices), thanks to including low-rank arithmetic from the Hmat-oss package. This package utilizes StarPU [17, 110] to offer task-based parallelism based on the Sequential-Task-Flow (STF) model, which decouples the task submission step from the parallel task execution [109]. Hmat-oss is the collection of open-source sequential routines to operate with low-rank structures and $\mathcal{H}$-Matrices that conform the Hmat package [36, 37, 86]. This is a proprietary library from Airbus that is also based in the original $\mathcal{H}$-Arithmetic defined in [55]. The closed-source Hmat package offers a fair task-based parallel efficiency, leveraging a fine grain dependency management on top of StarPU. The $\mathcal{H}$-Chameleon package proposed in this thesis combines Chameleon and Hmat-oss kernels, and utilizes StarPU for the parallelism. Thus, as Chameleon originally also follows a parallelization strategy similar to that in Hmat, the comparison between $\mathcal{H}$-Chameleon and Hmat performance is fair. As shown in Chapter 6, with $\mathcal{H}$-Chameleon we manage to offer a simpler (and open-source) implementation of the $\mathcal{H}$-Arithmetic that is competitive with Hmat.

Prior to the development of this thesis, there already existed parallel implementations of some of the operations in the $\mathcal{H}$-Arithmetic. Kriemann proposed some parallel $\mathcal{H}$-Matrices arithmetics on shared memory systems [78] via POSIX-threads [32]. This included the $\mathcal{H}$-Matrix construction, $\mathcal{H}$-Matrix-Vector product, $\mathcal{H}$-Matrix-$\mathcal{H}$-Matrix product, and the $\mathcal{H}$-Matrix inverse. Some time later, Kriemann describes in [79] a task-based parallel implementation of the $\mathcal{H}$-LU based on Intel TBB [105], which is used in $\mathcal{H}$-Lib^{pro} to accomodate an efficient scheduling on parallel systems. The tasks descriptions in this dissertation when parallelising the $\mathcal{H}$-LU in H2Lib are similar to those presented by Kriemann. The main difference is that the OmpSs-2 programming model is equipped with special features that allow us to anticipate the execution of tasks belonging to different levels in the $\mathcal{H}$-Matrix hierarchy. As a result, a fair parallel efficiency is reached. Moreover, H2Lib is fully open-source, while $\mathcal{H}$-Lib^{pro} is not.

More recent works [21] rely on alternative programming models (particularly, Cilk [45] and Tascell [62]). In case those programming models are employed to parallelize the $\mathcal{H}$-LU, it would be worth it to compare its efficiency with the one achieved by the parallel $\mathcal{H}$-LU from H2Lib or $\mathcal{H}$-Chameleon proposed in this dissertation.

Some of the authors of the work mentioned in the previous paragraph have already explored developing a hybrid MPI+OpenMP version to improve the parallel scalability of the construction and the $\mathcal{H}$-Matrix-vector product [69] in the HACApK package. A hybrid approach with MPI+OmpSs-2 and H2Lib is interesting and is in fact one of the open research lines of this dissertation. In case HACApK and H2Lib offers an MPI-based implementation of the $\mathcal{H}$-LU at some point, it could be interesting to analyze their differences and similarities.

As part of the future work with $\mathcal{H}$-Chameleon (already in process), there exists the plan to offer a distributed-memory parallelization based on MPI. In [68, 116] the authors present their own version of the lattice $\mathcal{H}$-Matrices, which are conceptually equivalent to our Tile $\mathcal{H}$-Matrix approach, though the data structures and kernels that conform their approach are different to those in $\mathcal{H}$-Chameleon. The simplicity in terms of leveraging existing packages is the main difference between $\mathcal{H}$-Chameleon and the mentioned works. Moreover, the high efficiency reflected by the evaluation of $\mathcal{H}$-Chameleon converts it into the first open-source package to offer Tile $\mathcal{H}$-Matrices based parallel implementations for shared memory systems. Once the MPI-based implementation of $\mathcal{H}$-Chameleon is ready, it will be interesting to compare its distributed parallel performance not only with Hmat, but also with the implementations of the mentioned works.

1.5 Structure of the document

This thesis is structured in seven chapters. We provide next a brief description of the six that follow the present introductory chapter:

- In Chapter 2 we provide a summary of the linear algebra terms and propositions that are necessary to establish the mathematical basis for $\mathcal{H}$-Matrices.
- In Chapter 3 we present an overview of the matrix computations and parallelization tools and strategies which constitute the fundamentals for the algorithms that perform $\mathcal{H}$-Arithmetic operations.
- In Chapter 4 we describe the first contribution developed in the context of this thesis: a limited implementation of the $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky algorithms (thus, prototypes) with the only purpose of testing task-parallelism in the context of $\mathcal{H}$-Matrices.
- In Chapter 5 we review the efforts made to parallelize the $\mathcal{H}$-LU in the H2Lib package, emphasizing the difficulties of the process and how the OmpSs-2 programming model helped us to overcome them.
- In Chapter 6 we present this thesis' last contribution: $\mathcal{H}$-Chameleon. This is an extension of the Chameleon package that operates with Tile $\mathcal{H}$-Matrices leveraging Hmat-oss package kernels.
- Lastly, in Chapter 7 we summarize the conclusions extracted from the PhD dissertation, as well as the (in)directly related publications. We conclude by describing some open research lines. Chapter 8 is equivalent to Chapter 7 but written in Spanish.

Chapter 2

Linear algebra background

Contents of the chapter

A matrix is a functional representation of the data, traditionally organized along two dimensions in computer science. By employing dense and sparse structures, a wide range of the data that needs to be represented with two dimensions is covered. However, recent advances in engineering simulations and computations require a gray perspective in what used to be regarded as black and white. This is the reason for the introduction of new matrix structures, as well as matrices with three or more dimensions. $\mathcal{H}$-Matrices lie in that “gray scale of approaches”, providing a powerful numerical tool to tackle scenarios in between the dense and sparse data configurations, offering a trade-off between accuracy and performance/storage gains.

A background in general linear algebra is required in order to construct and parallelize algorithms that perform operations over $\mathcal{H}$-Matrices. For this reason, in this chapter we introduce several general linear algebra definitions, properties and operations, before presenting the mathematical foundations of $\mathcal{H}$-Matrices. Mathematical proofs will not be detailed, but can be consulted in the referenced literature [31, 43, 56].

2.1 General vectors and matrices definitions and properties

This work follows the computer science convention to view vectors and matrices as sets of elements organized along one or more dimensions, respectively. The mathematical definitions underlying that idea are presented next.

Definition 2.1. (Vector) Let $\mathbb{R}$ be the finite set of ordered real numbers, and $\mathbb{R}^n$ the vector space of real n-vectors. Then, $x = (x_1, x_2, \ldots, x_n)$ defines a *vector* in $\mathbb{R}^n$ if $x_i \in \mathbb{R}$, *with* $1 \leq i \leq n$, $i \in \mathbb{N}$.

Definition 2.2. (Matrix) The structure $A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n}$ is a *matrix* if $a_{ij} \in \mathbb{R}$, *with* $1 \leq i \leq m$, $1 \leq j \leq n$, i,$j \in \mathbb{N}$. When indexing a matrix element a_{ij}, i and j will respectively index the row and column in which the specific element can be found.

Notation 2.1. Each of the values of a vector or a matrix is called an *entry*, *coefficient* , or *element.*

Notation 2.2. Let $x \in \mathbb{R}^n$, and $A \in \mathbb{R}^{m \times n}$. Then, the *i-th* component of x can be denoted as x_i, but also as $x(i)$ or $x[i]$; and the *(i,j)-th* entry $a_{ij} \in A$ can also be indexed as $A(i,j)$ or $A[i,j]$. Due to the computer science perspective adopted in this work, when indexing vector or matrix elements in later chapters, the first element will have index 0; for example, for a vector x, the top entry is x_0, and the top-left entry of a matrix A is a_{00}.

To be able to measure vectors and matrices, there exists the term *dimension.*

Definition 2.3. (Dimension) The term *dimension* refers to the topological size of the covering properties of an object, that is, the number of coordinates required to determine a point in the object.

The dimensions of vectors and matrices in the context of this work will be equivalent to the number of elements that form the specific structure. This means that, given a vector $v \in \mathbb{R}^n$, then n (amount of elements that form the vector) will specify the dimension of the vector v. Furthermore, given a matrix $A \in \mathbb{R}^{m \times n}$, its dimension will be $m \times n$ (number of rows times number of columns).

Notation 2.3. The dimension of a vector or a matrix will also be referred to as its *size* in this document. Moreover, the dimension of a vector is equivalent to its *length* and the dimension of a matrix can also be referred to as its *order.*

Remark 2.1. A vector can also be viewed as a matrix with either $m = 1$ or $n = 1$, respectively corresponding to a *row vector* or a *column vector.*

A concept that will gain importance along this document is the *rank.*

Definition 2.4. (Full vs. rank-deficient matrices) Let $A \in \mathbb{R}^{m \times n}$. Then its *rank*, denoted as $rank(A)$ or $rk(A)$, is defined as 1) the maximum number of linearly independent column vectors in the matrix if $m \geq n$; or 2) the maximum number of linearly independent row vectors in the matrix if $n \geq m$ (both definitions are equivalent if $m = n$). A matrix whose rank is equal to the largest possible according to its dimension (i.e., the minimum between its number of rows and columns) is said to be *full rank*; otherwise, it is called *rank-deficient* or *low-rank.*

Definition 2.5. (Rank deficiency) Let $A \in \mathbb{R}^{m \times n}$ be a low-rank matrix, then its *rank deficiency* is equivalent to $min(m, n) - rank(A)$.

Lastly, it is important to define the concept of (matrix) *norm* as it will be employed, among other purposes, to verify certain matrix computations.

Definition 2.6. (Norm) Let $A \in \mathbb{R}^{m\times n}$. Then, a *norm* of that matrix is provided by any function $|| \cdot || : \mathbb{R}^{m\times n} \rightarrow \mathbb{R}$ which verifies:

1) $||A|| \geq 0$ with $||A|| = 0$ if and only if $A = 0$ (*positivity*);

2) $\forall\ \alpha \in \mathbb{R}$, $||\alpha A|| = |\alpha| ||A||$ (*homogeneity* or *scaling*); and

3) $\forall\ B \in \mathbb{R}^{m\times n}$, $||A + B|| \leq ||A|| + ||B||$, and $||A|| - ||B|| \leq ||A - B||$ (*triangle inequality*).

2.2 Classification of matrices

There exist plenty of different schemes to classify matrices. For example, classifications can be made taking into account the matrix dimension, the existence and location of null entries, the partitioned or structured form they present, etc. The most utilized matrix types in the context of this thesis are defined in the following subsections.

2.2.1 Matrix types according to their dimension

Regarding their dimension, there exist rectangular and square matrices. These concepts are presented in this subsection, together with the concept of submatrix, and a few additional terms related to square matrices: determinant, eigenvalues, eigenvectors, singular values, and singular vectors.

Definition 2.7. (Rectangular and square matrices) Let $A \in \mathbb{R}^{m\times n}$. If $m < n$ or $n < m$, then A is said to be a *rectangular* matrix, particularly a *wide* or a *tall* matrix, respectively; if $m = n$, then A is *square*.

Remark 2.2. As stated before, a rectangular matrix where $m = 1 < n$ or $n = 1 < m$ is equivalent to a row vector or a column vector, respectively.

It is important to note that this document focuses on bidimensional matrices. Higher dimensioned matrices such as, for example, tensors, are out of the scope of this thesis.

A particularly important definition for future chapters, presented next, introduces *submatrix*. Its value comes from the fact that partitioning a matrix into submatrices (or blocks) is essential to achieve high parallel efficiency when executing algorithms that perform matrix computations. In computer science, there are many different ways of performing this partition, and some of them will be exposed in future chapters.

Definition 2.8. (Block) Let $A \in \mathbb{R}^{m\times n}$, and let I, J be the non-empty finite sets of indices that respectively correspond to the row and column indices of A. Then, a *submatrix* or *block* $B \subset A$ is determined by two subsets or clusters $\tau \in I$, $\sigma \in J$, expressed by $\tau \times \sigma$, whose elements are $B = (a_{ij})_{i\in\tau, j\in\sigma}$. The size of a block is determined by the number of elements that form each of its indices sets; this is, $size(B) = length(I) \times length(J)$.

The terms *determinant*, *eigenvalues*, *eigenvectors*, *singular values*, and *singular vectors* will be employed in this work when constructing matrices or evaluating matrix computations to verify that the desired results have been attained.

Definition 2.9. (Determinant) Let $A \in \mathbb{R}^{n \times n}$. Then, its *determinant*, denoted as $det(A)$ or $|A|$, is given by $det(A) = |A| = \sum_{i=1}^{n} a_{ij} a^{ij}$, with $1 \leq i,j \leq n$, $i,j \in \mathbb{N}$, where a^{ij} is the *cofactor* of a_{ij} defined by $a^{ij} = (-1)^{i+j} M_{ij}$. In this last expression, M_{ij} denotes the (i,j) minor of the matrix, that is, the determinant of the submatrix of A obtained by eliminating the *i-th* row and *j-th* column of the matrix.

Notation 2.4. In case $det(A) = 0$, the matrix is said to be *singular*; if $det(A) = 1$, then the matrix is said to be *unimodular*.

The eigenvectors, eigenvalues, singular values, and singular vectors are crucial to evaluate the importance of certain entries of a matrix with respect to others, and are frequently exploited when compressing the matrix information.

Definition 2.10. (Eigenvalues and eigenvectors) Let $A \in \mathbb{R}^{n \times n}$. Then $\lambda \in \mathbb{R}$ is an *eigenvalue* of A if and only if $Av = \lambda v$ for some nonzero column vector $v \in \mathbb{R}^n$. Each vector v that satisfies this equation is an *eigenvector* associated with the eigenvalue λ.

Definition 2.11. (Singular values and singular vectors) Let $A \in \mathbb{R}^{m \times n}$. Then, the nonnegative scalar σ is a *singular value* of A if $A \cdot v = \sigma \cdot u$ and $A^T \cdot u = \sigma \cdot v$, for a certain pair of real vectors $u \in \mathbb{R}^m$, $v \in \mathbb{R}^n$, which are referred to as *left* and *right singular vectors*, respectively.

Proposition 2.1. Let $A \in \mathbb{R}^{m \times n}$. Then, the number of nonzero singular values of A is equal to $rank(A)$.

2.2.2 Matrix types according to their data distribution

Another interesting criterion to classify matrices is based on their data distribution, this is, the way their entries are spread along the structure, or the particularities of some (of all) of their values. In this subsection, the terms *dense* matrix, *sparse* matrix, *diagonal* matrix, *identity* matrix, *band* matrix, *triangular* matrix, and *symmetric* matrix are defined.

Definition 2.12. (Dense matrix) Let $A \in \mathbb{R}^{m \times n}$. Then, if a representative amount of its elements are not zero, the matrix is considered to be *dense*.

In certain fields, for example due to the geometry of the problem where the data arises from, the values represented by matrices are repeated as if they were *reflected in a mirror*, constituting what is known as *symmetric* matrices.

Definition 2.13. (Symmetric matrix) Let $A \in \mathbb{R}^{n \times n}$. Then, A is *symmetric* if $a_{ij} = a_{ji}, with\ 1 \leq i,j \leq n$, $i,j \in \mathbb{N}$.

In contrast to dense matrices, sparse matrices are mostly populated with zero elements. The distribution of the non-zero entries along the matrix gives room to special cases that are detailed next.

Definition 2.14. (Sparse matrix) Let $A \in \mathbb{R}^{m \times n}$. Then, if the majority of its entries are zero, it is called a *sparse* matrix. The measure of zero elements with respect to non-zero ones is referred to as the matrix *sparsity*.

The extreme case of a sparse matrix is the null matrix, defined next.

Definition 2.15. (Null matrix) Let $A \in \mathbb{R}^{m\times n}$. Then, A is a *null* matrix if and only if $a_{ij} = 0,\ with\ 1 \leq i \leq m, 1 \leq j \leq n,\ i{,}j \in \mathbb{N}$; this is, all the matrix entries are zero.

Besides containing a remarkable amount of null entries, if the non-zero elements of a sparse matrix are located following a regular pattern, or contain certain values, the matrix can be classified as *diagonal*, *identity*, *null*, *band*, or *triangular*.

Definition 2.16. (Diagonal matrix) Let $A \in \mathbb{R}^{n\times n}$. Then, A is a *diagonal* matrix if and only if $a_{ij} = 0$, with $1 \leq i{,}j \leq n$, $i \neq j$, $i{,}j \in \mathbb{N}$; this is, all the non-zero entries of A are placed on its diagonal.

Definition 2.17. (Identity matrix) Let $A \in \mathbb{R}^{n\times n}$. Then, A is the *identity* matrix if and only if $a_{ii} = 1$ with $1 \leq i \leq n$, $i \in \mathbb{N}$; this is, all its diagonal values equal one, and all the other elements are zero.

Notation 2.5. The identity matrices of a certain size n will be denoted as I_n.

In the previous section, the eigenvalues of a square matrix were defined. The set of eigenvalues can be expressed as stated in the following proposition, by referencing the identity matrix.

Proposition 2.2. For every square matrix $A \in \mathbb{R}^{n\times n}$, the set of its eigenvalues is denoted by $\lambda(A) = \{\lambda : det(A - \lambda I_n) = 0\}$. Concretely, $det(A - \lambda I_n) = 0$ is called the *characteristic equation* or *characteristic polynomial* of A, and because this equation does not have more than n solutions, A does not have more than n eigenvalues.

If the non-zero entries of the matrix are not limited to the diagonal, but stay close to it in what can be seen as *parallel diagonals*, we obtain a band matrix, defined next.

Definition 2.18. (Upper/lower diagonal) Let $A \in \mathbb{R}^{m\times n}$. Then, the *k-th upper diagonal* of A is composed by all $a_{ij} \in A : i = j - k$, with $1 \leq i \leq m - k, 1 \leq k < j \leq n$, $i{,}j \in \mathbb{N}$, and accordingly the *k-th lower diagonal* of A is composed by all $a_{ij} \in A : j = i - k$, with $1 \leq j \leq n - k, 1 \leq k < i \leq m$, $i{,}j \in \mathbb{N}$.

Definition 2.19. (Band matrix) Let $A \in \mathbb{R}^{m\times n}$. Then, A is a *band* matrix with distances p, $q \in \mathbb{N}$ if and only if $\forall\ a_{ij} > 0$, $a_{ij} \in A$, that entry either belongs to the diagonal, any of the *k-th* lower diagonals with $k \leq p$, or the *k-th* upper diagonals with $k \leq q$, and $a_{ij} = 0$ for all entries outside that (sub)diagonals.

Lastly, triangular matrices are presented. As it will be justified later in this document, they constitute the clue for the efficient solution of systems of linear equations.

Definition 2.20. (Triangular matrix) Let $A \in \mathbb{R}^{m\times n}$. Then A is an *upper triangular* matrix if 1) $\forall\ a_{ij} \in A : a_{ij} \neq 0 \rightarrow i \leq j,\ and\ \forall\ a_{ij} \in A : i > j \rightarrow a_{ij} = 0$, with $1 \leq i \leq m$, $1 \leq j \leq n$, $i{,}j \in \mathbb{N}$, or 2) *lower triangular* matrix if $\forall a_{ij} \neq 0 \rightarrow i \geq j,\ and\ \forall\ a_{ij} \in A : i < j \rightarrow a_{ij} = 0$, with $1 \leq i \leq m$, $1 \leq j \leq n$, $i{,}j \in \mathbb{N}$.

2.3 Main vector and matrix operations

The algorithms presented in future chapters, which constitute the core of this dissertation, will address several matrix operations. These operations will mainly be applied to the resolution of systems of linear equations. This subsection provides a brief linear algebra background on these operations, as it constitutes the basis for the mentioned algorithms.

2.3.1 Basic vectors and matrix operations

First, basic operations involving addition and/or product of vectors and/or matrices need to be defined, prior to more complex ones.

Definition 2.21. **(Vector-vector addition)** Let v,$w \in \mathbb{R}^m$. Then, their *addition*, denoted by $v+w$, gives a vector $u \in \mathbb{R}^m$, defined by $u_i = v_i + w_i \in \mathbb{R}$, $\forall u_i \in u$, $v_i \in v$, $w_i \in w$, $1 \leq i \leq m$, $i \in \mathbb{N}$.

Definition 2.22. **(Matrix-matrix addition)** Let A,$B \in \mathbb{R}^{m\times n}$. Then, their *addition*, denoted by $A + B$, results into a matrix $C \in \mathbb{R}^{m\times n}$ whose entries are defined by $c_{ij} = a_{ij} + b_{ij}$, $\forall\ c_{ij} \in C$, $a_{ij} \in A, b_{ij} \in B, 1 \leq i \leq m, 1 \leq j \leq n$, i,$j \in \mathbb{N}$.

Remark 2.3. To be able to perform the addition of two vectors, it is mandatory that they have the same length. In the case of adding two matrices, both need to have the same dimensions.

Definition 2.23. **(Scalar-vector product)** Let $v \in \mathbb{R}^m$ and $\alpha \in \mathbb{R}$. Then, their *scalar-vector product*, denoted by $\alpha \cdot v$, results in a vector $w \in \mathbb{R}^m$, defined by $w_i = \alpha \cdot v_i \in \mathbb{R}^m,\ \textit{with}\ w_i \in w,\ v_i \in v,\ \ 1 \leq i \leq m.$

Definition 2.24. **(Scalar-matrix product)** Let $A \in \mathbb{R}^{m\times n}$ and $\alpha \in \mathbb{R}$. Then, their *scalar-matrix product*, denoted by $\alpha \cdot A$, results in a matrix $B \in \mathbb{R}^{m\times n}$, defined by $b_{ij} = \alpha \cdot a_{ij}$, *with* $b_{ij} \in B$, $a_{ij} \in A$, $1 \leq i \leq m, 1 \leq j \leq n$, i,$j \in \mathbb{N}$.

Definition 2.25. **(Dot product)** Let v,$w \in \mathbb{R}^m$. Then, their *dot product*, denoted by $v \cdot w$, results into a scalar value $\alpha \in \mathbb{R}$ defined by $\alpha = \sum_{i=1}^{m} a_i \cdot b_i$, with $a_i \in a$, $b_i \in b$, $1 \leq i \leq m$, $i \in \mathbb{N}$.

Definition 2.26. **(Vector-matrix product)** Let $A \in \mathbb{R}^{m\times n}$ and $v \in \mathbb{R}^m$ be a row vector. Then, their *vector-matrix product*, denoted by $v \cdot A$, results in a row vector $w \in \mathbb{R}^n$, and its entries are defined as follows: $w_j = \sum_{i=1}^{m} v_i \cdot a_{ij}$, $\forall w_j \in w$, $a_{ij} \in A, v_j \in v, 1 \leq i \leq m, 1 \leq j \leq n$, i,$j \in \mathbb{N}$. Equivalently, if $u \in \mathbb{R}^n$ is a column vector, then the *matrix-vector product* $A \cdot u$ gives a column vector $w \in \mathbb{R}^m$, defined by $w_i = \sum_{j=1}^{n} a_{ij} \cdot u_j \forall w_i \in w$, $a_{ij} \in A, u_j \in u, 1 \leq i \leq m, 1 \leq j \leq n$, i,$j \in \mathbb{N}$.

Definition 2.27. **(Matrix-matrix product)** Let $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$. Then, their *matrix-matrix product*, denoted by $A \cdot B$, results into a matrix $C \in \mathbb{R}^{m\times p}$, where its entries are defined as follows: $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}, \forall\ c_{ij} \in C$, $a_{ik} \in A, b_{kj} \in B, 1 \leq i \leq m, 1 \leq j$,$k \leq p$, $\ i$,j,$k \in \mathbb{N}$.

Remark 2.4. To be able to perform the product of two matrices, it is necessary that the number of columns of first matrix and the number of rows of the second one are the same.

2.3.2 Matrix decompositions

Now that the basic operations have been defined, it is the moment to present the main decompositions that will be employed along this document: Singular Value Decomposition (SVD), LU, and Cholesky.

Prior to the definition of the SVD, it is necessary to define the terms *transposed*, *inverse*, and *orthogonal* matrix, as the SVD will generates orthogonal matrices though a process where the other two are utilized.

Definition 2.28. (Transposed matrix) Let $A \in \mathbb{R}^{m \times n}$. Then the *transposed* matrix of A is denoted by A^T with each of the entries following that $A(i,j) = A^T(j,i)$, *with* $1 \leq i \leq m$, $1 \leq j \leq n$, $i,j \in \mathbb{N}$; this is, the *i-th* row and *j-th* column of A become the *j-th* row and *i-th* column entry of A^T.

Definition 2.29. (Inverse matrix) Let $A \in \mathbb{R}^{n \times n}$. Then A is *invertible* if there exists the *inverse* matrix of A, denoted by A^{-1}, which verifies that $A \cdot A^{-1} = A^{-1} \cdot A = I_n$.

Notation 2.6. An invertible matrix is also called *nonsingular* or *nondegenerate*.

Definition 2.30. (Orthogonal matrix) Let $A \in \mathbb{R}^{n \times n}$. Then, A is an *orthogonal* matrix if and only if $A \cdot A^T = I_n$. Particularly, $A^{-1} = A^T$ for orthogonal matrices, with the entries verifying $\hat{a}_{ij} = a_{ji}$, $\forall \hat{a}_{ij} \in A^{-1}$, $a_{ji} \in A^T$, $1 \leq i,j \leq n$, $i,j \in \mathbb{N}$.

The following theorem defines the Singular Value Decomposition. It is crucial for building and operating with $\mathcal{H}$-Matrices because, as it will be properly exposed later, it is the key tool to isolate the less representative values in a structure, which will be removed to allow compressing them.

Theorem 2.3.1. (Singular Value Decomposition - SVD) Let $A \in \mathbb{R}^{m \times n}$. Then there exist orthogonal matrices $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$, such that $U^T \cdot A \cdot V = \Sigma$, with $\Sigma \in \mathbb{R}^{m \times n}$ being a diagonal matrix with the (scalar) singular values $\sigma_1, \sigma_2 \ldots \sigma_n$ on its diagonal, ordered from the largest (first entry) to the smallest (last entry) value, such that $0 < \Sigma_{ii} = \sigma_i \leq \sigma_{i+1} = \Sigma_{i+1,i+1}$, $1 \leq i \leq n$, $i \in \mathbb{N}$.

Remark 2.5. As stated in a previous proposition, the rank of a matrix is equal to the number of its nonzero singular values. Thus, the SVD exposes the rank of a matrix.

If the SVD offers a powerful numerical tool to attain a proper compression of the information, the LU factorization is needed to efficiently solve systems of equations, thanks to the reduction of the problem to the solution of triangular systems of equations that result from employing this decomposition. (Note that an SVD could also be employed to solve linear systems, but it would be much more costly than the LU in terms of computations.)

Notation 2.7. Let $A \in \mathbb{R}^{m \times n}$, then the subset of entries of A located from the row r_a to the row r_b, and from the column c_a to the column c_b is denoted by $A_{r_a:r_b,c_a:c_b} = A[r_a : r_b, c_a : c_b] = A(r_a : r_b, c_a : c_b)$. Equivalently, let $v \in \mathbb{R}^n$ be a real vector, then the subset of elements of v from the *a-th* element to the *b-th* one is denoted by $v_{a:b} = v[a : b] = v(a : b)$.

Theorem 2.3.2. (LU decomposition) Let $A \in \mathbb{R}^{n \times n}$, and consider $det(A[1:k,1:k]) \neq 0$, with $1 \leq k \leq n,\ k \in \mathbb{N}$. Then there exist a unit lower triangular matrix $L \in \mathbb{R}^{n \times n}$ and an upper triangular matrix $U \in \mathbb{R}^{n \times n}$, such that $A = L \cdot U$. Moreover, the LU factorization is unique and $det(A) = u_{11} \cdot u_{22} \dots u_{nn}$.

An additional important decomposition is the Cholesky factorization. This matrix operation is particularly efficient for certain types of systems of equations, as explained next.

Definition 2.31. (Positive definite matrix) Let $A \in \mathbb{R}^{n \times n}$. Then A is a *positive definite* matrix if $x^T Ax > 0$ for any real vector $x \in \mathbb{R}^n$ that is not null. Equivalently, A is a *positive definite* matrix if all the eigenvalues of A are positive.

Theorem 2.3.3. (Cholesky decomposition) Let $A \in \mathbb{R}^{n \times n}$ be a symmetric positive definite matrix. Then, there exists a unique real square lower triangular matrix $G \in \mathbb{R}^{n \times n}$, with positive diagonal entries, such that $A = GG^T$.

2.3.3 Systems of equations

A considerable number of scientific applications involve systems of linear equations. Thus, the computer science community has put a considerable effort on optimizing the calculus related to this problem. This section presents the associated basic mathematical concepts definitions.

Definition 2.32. (Equation) An *equation* is a statement that equals two algebraic expressions that have the same value. The variables $x, y, z, \dots$ that conform the equation are named *unknowns*, the scalars that multiply the unknowns are named *coefficients*, and the scalar values added to the (multiplied) unknowns are named *constants*.

Notation 2.8. An equation is *linear* if both sides of it are a sum of (constant) multiples of x,y,z... plus an optional constant.

Definition 2.33. (System of linear equations) A *system of linear equations* is a collection of at least two equations with unknowns x,y,z,... The set of values (each of them associated to each of the unknowns) that make all the equations true simultaneously conform the system *solution*. If there exists such a solution, then the system is *consistent*; otherwise, it is *inconsistent*.

Solving systems of linear equations is commonly done by converting the problem into a triangular system that has the same solution as the original one. Future chapters will describe forward and back substitution methods (employed to this end), together with different computational strategies to optimize the process of solving them.

2.4 Compressed matrices

As it was highlighted earlier in this document, the interest on utilizing $\mathcal{H}$-Matrices in computer science is the storage savings and performance gains that can be attained in exchange for a certain accuracy loss. This accuracy reduction is due to the compression performed on the original data, which dismisses the less representative elements to build a compressed structure.

This subsection will introduce $\mathcal{H}$-Matrices. To this end, the basis of $\mathcal{H}$-Matrices are presented, including different low-rank related concepts, partitioning criteria and rules, and different compressed structures.

2.4.1 The basics: low-rank matrices, admissibility and partitions

A rough definition of $\mathcal{H}$-Matrices can describe them as matrices that have been recursively partitioned into submatrices which are either subdivided into more submatrices, or converted into a low-rank format (unless that certain submatrix cannot be compressed and, therefore, is kept in a dense format).

In the process of building $\mathcal{H}$-Matrices, a procedure to perform the partition into submatrices is needed, together with a criterion to determine whether a certain submatrix or block can be compressed (admissible) or not (inadmissible). For this reason, this introductory part defines the terms low-rank matrix, admissibility, and partition, as well as other related concepts necessary to finally construct $\mathcal{H}$-Matrices.

Definition 2.34. (Low-rank matrix) Let $A \in \mathbb{R}^{m\times n}$ with $r = rank(A) = min(m,n)$. Also, let the SVD be given by $A = U_{m\times r} \cdot \Sigma_{r\times r} \cdot V^T_{r\times n}$, which can be expressed:

$$A = [u_1 u_2 \cdots u_r] \begin{bmatrix} \sigma_1 & & & 0 \\ & \sigma_2 & & \\ & & \ddots & \\ 0 & & & \sigma_r \end{bmatrix} \begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_r^T \end{bmatrix}$$

with $u_1,u_1,\ldots,u_r$ denoting the columns of $U_{m\times r}$ and $v_1,v_2,\ldots,v_r$ the rows of $V_{n\times r}$. This SVD can be employed to convert A into a *low-rank matrix*, with rank $k < r \in \mathbb{N}$, by annihilating the $r-k$ trailing singular values of A; this is $A_k = U_{m\times k} \cdot (\Sigma_k)_{k\times k} \cdot V^T_{k\times n} = \sum_{i=1}^{k} \sigma_i \cdot u_i \cdot v_i^T$, where $\Sigma_k = diag(\sigma_1,\sigma_2,\ldots \sigma_k)$, $U_{m\times k} = [u_1,u_1,\ldots,u_k]$, and $V_{k\times n} = [v_1,v_2,\cdots,v_k]$.

When reducing the information originally contained in the matrices to generate a compressed structure, it is crucial to set a criteria to isolate the less representative elements that will be removed. Concretely, the process of building $\mathcal{H}$-Matrices will determine whether certain submatrices can be converted into low-rank submatrices (or blocks). This decision of whether keeping/removing elements is now explained in detail, from admissibility conditions to partitioning processes, being the former ones the criteria to refine or stop the division performed by the later ones.

Definition 2.35. (η-admissibility) Let $A \in \mathbb{R}^{m\times n}$, with $m = |I|$, $n = |J|$, I,J non-empty finite sets representing the rows and columns indices of A[1]. Let b be a block of A such that $b = \tau \times \sigma \subseteq A$, with $\tau \subseteq I$, $\sigma \subseteq J$, and let $0 < \eta \in \mathbb{R}$. Then, the block b is η-admissible if

$$min\{diam(\tau),diam(\sigma)\} \leq \eta\ dist(\tau,\sigma) \tag{2.1}$$

with *diam* and *dist* defined via the Euclidean norm, as:

$$diam(\tau) := max\{||x_1 - x_2|| : x_1,x_2 \in \mathcal{X}_\tau\},$$

[1]Hereafter, this will be denoted by $A \in \mathbb{R}^{I\times J}$.

$$diam(\sigma) := max\{||x_1 - x_2|| : x_1{,}x_2 \in \mathcal{X}_\tau\},$$
$$dist(\tau{,}\sigma) := min\{||x_1 - x_2|| : x_1 \in \mathcal{X}_\tau{,}x_2 \in \mathcal{X}_\sigma\},$$

where $\mathcal{X}_\tau$ and $\mathcal{X}_\sigma$ correspond to the subsets of τ and σ where the function that generates the data in that block is not zero (namely, the *supports* of τ and σ).

Proposition 2.4.1. The η-admissibility condition ensures that

$$min\{diam(\mathcal{X}_\tau), diam(\mathcal{X}_\sigma)\} \leq \eta \; dist(\mathcal{X}_\tau, \mathcal{X}_\sigma).$$

While this allows that the matrix data are properly evaluated, it is important to remark that this classical admissibility definition can be very costly to calculate. Consequently, the η-admissibility it is not usually calculated with respect to supports, but to bounding boxes.

Definition 2.36. (Bounding box) Let $A \in \mathbb{R}^{I \times J}$, and assume $\tau \subseteq I$ is a cluster whose support is defined as $\mathcal{X}_\tau$. Then, the he smallest axis-parallel cuboid $\mathcal{B}$ containing τ, denoted by $\mathcal{B}_\tau$, is called *bounding box.*

Figure 2.1 illustrates graphically what the supports and bounding boxes represent, as well as the *diam* and *dist* in definition 2.36.

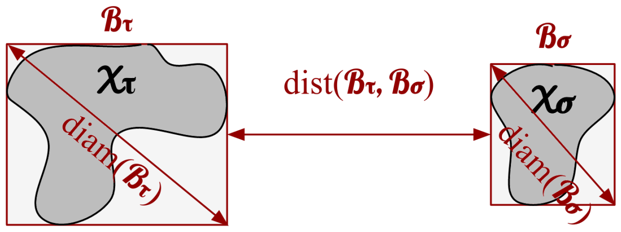

Figure 2.1: Sample of supports ($\mathcal{X}_\tau$, $\mathcal{X}_\sigma$) and bounding boxes ($\mathcal{B}_\tau$, $\mathcal{B}_\sigma$) for the domains τ and σ.

Definition 2.37. (η-admissibility with respect to bounding boxes) Let $A \in \mathbb{R}^{I \times J}$, assume b is a block of A such that $b = \tau \times \sigma \subseteq A$, with $\tau \subseteq I$, $\sigma \subseteq J$. Let $0 < \eta \in \mathbb{R}$, and let $\mathcal{B}_\tau$ and $\mathcal{B}_\sigma$ be the bounding boxes of the supports $\mathcal{X}_\tau$ and $\mathcal{X}_\sigma$, respectively. Then, the η-admissibility condition expressed in (2.35) is redefined by

$$min\{diam(\mathcal{B}_\tau){,}diam(\mathcal{B}_\sigma)\} \leq \eta \; dist(\mathcal{B}_\tau, \mathcal{B}_\sigma). \tag{2.2}$$

In the context of this work, the admissibility criteria will be employed to determine whether a certain matrix block is admissible and, consequently, it can be compressed and stored in a low-rank format or, contrarily, it should be divided into smaller blocks (which will be re-evaluated to check if the admissibility condition is then fulfilled). In case the subdivision of an inadmissible block yields a block of dimension that is too small, it is kept in a dense format.

The process of generating a partition of a matrix implicitly involves the creation of a tree whose hierarchy of nodes dictates the subdivisions applied to the matrix blocks. For this reason, basic definitions and properties of trees are presented next, before detailing the partitioning methods.

Definition 2.38. (Graph) Let V, E be non-empty, finite sets respectively named *vertex set* and *edges set* . Then, the pair set (V,E) is called a *graph* with vertices $v \in V$ and edges $e \in E$, if it fulfills the property $E \subset V \times V$, with each pair $e = (v,w)$ denoting an edge from v to w, with $v,w \in V$.

Notation 2.9. In case all the edges of a graph are directed from one vertex to another, it is a *directed graph.*

Notation 2.10. A graph G is said to be *connected* if there is a path between any pair of vertices; otherwise, it is called *disconnected.*

Definition 2.39. (Graph cycle) Let $G = (V,E)$ be a graph. Then a *graph cycle* is a subset of the edge set $E' \subset E$ that forms a path such that the last node corresponds to the first node.

In a graph, certain relationships between specific vertices are highlighted, and the *son-father* relation is widely employed.

Notation 2.11. A graph G is said to be *acyclic* if graph cycles are not possible.

Definition 2.40. (Son mapping) Let G be a graph with a non-empty, finite vertex set V of size $n \in \mathbb{N}$. Then, $S : V \to \mathcal{P}(V)$ is a mapping from V into the power set $\mathcal{P}(V)$ (with *power set* meaning the set of all index subsets) called *son mapping* verifying the following statements:

1. Given two vertices $v,w \in V$, then w is a *son* of v if $w \in S(V)$, and accordingly v is the *father* of w.
2. Any sequence of vertices $(v_1,v_2 \dots v_k) \in V$ with $1 \leq k \leq n$, $k \in \mathbb{N}$ is called a *path* if $v_{i+1} \in S(v_i)$, *with* $1 \leq i < k$, $k \in \mathbb{N}$, and k is equivalent to the *path length.*
3. Given two vertices $v,w \in V$, if there exists a path from v to w, then w is called *successor* of v and w is the *predecessor* of v.

Definition 2.41. (Matrix graph) Let A be a matrix whose index sets are respectively given by I (rows indices) and J (columns indices); this is, $A \in \mathbb{R}^{I \times J}$. Then, the *matrix graph* $G(A)$ associated with A is defined by $V = I \cup J$, $E = \{(i,j) \in I \times J : A_{ij} \neq 0\}$.

Trees are specific types of graphs which constitute, as stated earlier, one of the $\mathcal{H}$-Matrices foundations since they allow to form the partitioning of the structure.

Definition 2.42. (Tree) Let G be a graph with a non-empty, finite vertex set V of size $n \in \mathbb{N}$, and let $S(V)$ denote its son mapping. Then, the structure formed by the vertex set together with the son mapping is called a *tree*, denoted by $\mathcal{T} = (V,S)$, if it is a connected and acyclic graph, and the following properties are fulfilled:

1. $\mathcal{T}$ has exactly $n - 1$ edges. This is equivalent to state that there is exactly one vertex $r \in V$ such that $\cup_{v \in V} S(v) = V \backslash \{r\}$, which means that r is not a son of any other vertex. The vertex r is called the tree *root*, denoted by *root*(V). Moreover, all $v \in V$ are successors of $r = root(V)$.

2. Any two vertices in $\mathcal{T}$ are connected exclusively by one path, and this implies that any $v \in V \backslash \{r\}$ has exactly one father.

3. For any new edge, $\mathcal{T}$ contains exactly one cycle.

4. Every edge is a cut-edge; this is, if that edge is deleted, then the graph becomes disconnected.

Definition 2.43. (Tree leaf) Let $\mathcal{T}$ be a tree with a vertex set V and a son mapping S. Each vertex $v \in V$ such that $S(v) = \emptyset$ (i.e., does not have any son) is called a *leaf*, and the set of leaves of $\mathcal{T}$ is denoted by $\mathcal{L}(T)$.

Definition 2.44. (Vertex level) Let $\mathcal{T}$ be a tree with a vertex set V and a son mapping S. Then, the *level of a vertex* $v \in V$ is defined as $level(v) = length(path\ r(\mathcal{T}) - v)$, this is, the amount of vertices forming that path. Accordingly, the *depth* of the tree is defined as $depth(\mathcal{T} := max\{level(v) : v \in V\}$, in other words, the maximum depth of its vertices.

Definition 2.45. (Vertex degree) Let $\mathcal{T}$ be a tree with a vertex set V and a son mapping S. Then, the *degree of a vertex* $v \in V$ is defined as $degree(v) = \#S(v)$, this is, the amount of sons and successors that v has. Moreover, the degree of the tree corresponds to degree$(\mathcal{T}) = max_{v \in V} degree(v)$.

A special type of trees named Cluster Tree (CT) is presented next.

Definition 2.46. (Cluster Tree - CT) Let I be a set of indices. Then, $\mathcal{T}_I$ is a *Cluster Tree (CT)* formed with the elements of I as vertices and $S(V) = S(I)$ (the son mapping), if all these conditions are verified:

1. I is the root of $\mathcal{T}_I$.

2. $\cup_{\sigma \in S(\tau)} \sigma = \tau \ \forall \ \tau \in \mathcal{T}_I \backslash \mathcal{L}(\mathcal{T}(I))$, this is, each node which is not a leaf is equivalent to the disjoint union of its sons.

3. $T_I \subset \mathcal{P}(I) \setminus \{\emptyset\}$, this is, each node of $\mathcal{T}_I$ is a subset of the index set I.

In a CT, the term "cluster" derives from the fact that each of its vertices is constituted by a subset of an index set (which is the root of the tree), instead of being formed by only an individual element. Consequently, it is logical to think of employing a CT to define a block partition of a vector, with the set of indices of the CT being the indices that correspond to the vector entries. Thus, as we pursue a block partition of a matrix, it seems natural to extend this concept from one dimension (vector) to two dimensions (matrix) which, in turn, seems possible by combining two CTs, formed by the indices associated either to the rows or the columns of the matrix. This is how the concept Block Cluster Tree (BCT) becomes tangible.

Definition 2.47. (Block Cluster Tree - BCT) Let I, J be sets of indices, and let $\mathcal{T}_I$ and $\mathcal{T}_J$ be the CTs respectively generated from them. Then, $\mathcal{T}_{I \times J}$ is a *Block Cluster Tree (BCT)* generated from $\mathcal{T}_I$ and $\mathcal{T}_J$ with $S = S_{\mathcal{T}_{I \times J}}$ defining the set of sons, fulfilling the following statements:

1. $I \times J$ is the root of the tree $\mathcal{T}_{I \times J}$.

2. For each possible block in the BCT (this is, for each of its vertices) $b \in \mathcal{T}_{I\times J}$, $b = \cup_{b'\in S(b)} b' \forall\, b \in \mathcal{T}_{I\times J} \backslash \mathcal{L}(\mathcal{T}(I \times J))$. In other words, each node which is not a leaf is equivalent to the disjoint union of its sons.

3. Each block (or vertex) of the CT $b \in \mathcal{T}_{I\times J}$ can be denoted by $b = \tau \times \sigma$ with $\tau \in \mathcal{T}_I$, $\sigma \in \mathcal{T}_J$.

4. Each sub-block of a certain block in the BCT $b' = \tau' \times \sigma' \in S(b) \in \mathcal{T}_{I\times J}$, with $b = \tau \times \sigma \in \mathcal{T}_{I\times J} \backslash \mathcal{L}(\mathcal{T}(I \times J))$ verifies that $\tau' = \tau$ *or* $\tau' \in S_{\mathcal{T}_I}(\tau)$, and $\sigma' = \sigma$ *or* $\sigma' \in S_{\mathcal{T}_J}(\sigma)$; this is, the sub-block is formed by either a subset of indices (or the subset itself) or each of the indices subsets defining b.

Prior to explaining the process of building an $\mathcal{H}$-Matrix, it is necessary to extend the definition of the term *η-admissibility* provided in definition 2.35. That definition provides a general view of the standard admissibility criteria; however, sometimes it is still too costly to compute, even when utilizing bounding boxes instead of *raw* data directly. This is the reason to employ *weak admissibility* conditions [58] in contrast to *strong admissibility* criteria (based on the standard admissibility described in Definition 2.35). Both conditions are next presented to clarify the differences.

Definition 2.48. (Strong admissibility) Let $A \in \mathbb{R}^{I\times J}$, with I, J non-empty, finite sets representing its row and column indices, let b be a block of A such that $b = \tau \times \sigma \subseteq A$, with $\tau \subseteq I$, $\sigma \subseteq J$; and let $0 < \eta \in \mathbb{R}$. Assume η-admissibility (as defined in definition 2.35) is the criteria utilized. Then, the *strong admissibility condition* is as follows:

$$b \text{ is admissible } \iff$$
$$1)\ b \text{ is a leaf of the BCT that defines the partition of } A\text{; or}$$
$$2)\ b \text{ is } \eta\text{-admissible, this is, } min\{diam(\sigma),\, diam(\tau)\} \leq \eta\ dist(\sigma,\tau)$$

Definition 2.49. (Weak admissibility) Let $A \in \mathbb{R}^{I\times J}$, with I,J non-empty, finite sets representing its row and column indices, let b be a block of A such that $b = \tau \times \sigma \subseteq A$, with $\tau \subseteq I$, $\sigma \subseteq J$, and let $0 < \eta \in \mathbb{R}$. Then, the *weak admissibility* condition is as follows:

$$b \text{ is admissible } \iff$$
$$1)\ b \text{ is a leaf of the BCT that defines the partition of } A\text{; or}$$
$$2)\ \sigma \neq \tau$$

Having defined the weak/strong admissibility criterion and the BCT, it is the moment to define a matrix partition.

Definition 2.50. (Matrix partition) Let $A \in \mathbb{R}^{m\times n}$ be a real matrix whose row and column indices are respectively defined by the sets I and J, and let $\mathcal{T}(I \times J)$ be a BCT. Then, $\mathcal{P}$ is called a *partition* of $I \times J$ if the following statements are true:

1. $\mathcal{P} \subset \mathcal{T}(I \times J)$ (consistency with respect to $\mathcal{T}(I \times J)$).

2. $b,b' \in \mathcal{P} \rightarrow (b = b'$ *or* $b \cap b' = \emptyset)$ (disjointedness).

3. $\dot{\cup}_{b\in\mathcal{P}} b = I \times J$ (disjoint covering property).

At this point, all the necessary terms which conform the basis or foundations of $\mathcal{H}$-Matrices have been introduced. It only remains to put them all together to form an *admissible matrix partition*, which will actually define the hierarchy of nested blocks that conforms an $\mathcal{H}$-Matrix.

Definition 2.51. (Admissible matrix partition) Let $A \in \mathbb{R}^{I \times J}$ be a real matrix whose row and column indices are respectively defined by the sets I and J, and let $\mathcal{P} \subset \mathcal{T}(I \times J)$ be a partition of $I \times J$. Let also $adm : b \in \mathcal{P} \to \{admissible, inadmissible\}$ be the function that determines whether a given block b of the partition is admissible or not. Then, $\mathcal{P}$ is an *admissible matrix partition* if either $adm(b) = admissible$ or $b \in \mathcal{L}(\mathcal{T}(I \times J))$ for all $b \in \mathcal{P}$.

2.4.2 $\mathcal{H}$-Matrices

After having defined low-rank matrices, admissibility conditions and partitioning through BCTs, it is the moment to define the $\mathcal{H}$-Matrices, followed by some interesting properties that characterize them.

Definition 2.52. (Hierarchical matrix, $\mathcal{H}$-Matrix) Let $A \in \mathbb{R}^{I \times J}$ be a real matrix whose row and column indices are respectively defined by the sets I and J, and let $\mathcal{P} \subset \mathcal{T}(I \times J)$ be an admissible matrix partition of $I \times J$. Let $r : \mathcal{P} \to \mathbb{N}_0$ be the local rank distribution function. Then, all matrices $M \in \mathbb{R}^{I \times J}$ with $rank(M|_b) \leq r(b)$ for all $b \in \mathcal{P}$ conform the set $\mathcal{H}(r,\mathcal{P}) \subset \mathbb{R}^{I \times J}$, with each of them being *hierarchical matrices* ($\mathcal{H}$-Matrices).

Definition 2.53. (Restriction) Let $\mathcal{H}(r,\mathcal{P}) \in \mathbb{R}^{I \times J}$ be the set of $\mathcal{H}$-Matrices with respect to the partition $\mathcal{P} \subset \mathcal{T}(I \times J)$ and the rank distribution r. Let also $I' \times J' \subset I \times J$. Then, the partition $\mathcal{P}|_{I' \times J'} := \{b \cap (I' \times J') : b \in \mathcal{P}\} \backslash \{\emptyset\}$ of $I' \times J'$ is called *restriction* of $\mathcal{P}$ to $\mathcal{T}(I' \times J')$, and the following properties are verified:

1. The partition $\mathcal{P}|_{I' \times J'}$ of $I' \times J'$ is admissible if $\mathcal{P}$ is an admissible partition.
2. The restriction of an $\mathcal{H}$-Matrix $M \in \mathcal{H}(r,\mathcal{P})$ leads to a hierarchical submatrix $M|_{I' \times J'} \in \mathcal{H}(r,\mathcal{P}|_{I' \times J'})$.
3. If $I' \times J' \in \mathcal{T}(I \times J,\mathcal{P})$, then the restricted partition $\mathcal{P}|_{I' \times J'}$ is a subset of $\mathcal{P}$.

Proposition 2.3. (Diagonal invariance) Let $\mathcal{H}(r,\mathcal{P}) \in \mathbb{R}^{I \times J}$ be the set of $\mathcal{H}$-Matrices with respect to the partition $\mathcal{P} \subset \mathcal{T}(I \times J)$ and the rank distribution r, and $M \in \mathcal{H}(r,\mathcal{P}) \in \mathbb{R}^{I \times J}$. Then, for all diagonal matrices $D_1 \in \mathbb{R}^{I \times I}$ and $D_2 \in \mathbb{R}^{J \times J}$, the products $D_1 \cdot \mathcal{M}$, $\mathcal{M} \cdot D_2$, and $D_1 \cdot \mathcal{M} \cdot D_2$ are also $\mathcal{H}$-Matrices which belong to $\mathcal{H}(r,\mathcal{P}) \in \mathbb{R}^{I \times J}$.

Proposition 2.4. (Invariance with respect to transposition) Let $\mathcal{H}(r,\mathcal{P}) \in \mathbb{R}^{I \times J}$ be the set of $\mathcal{H}$-Matrices with respect to the partition $\mathcal{P} \subset \mathcal{T}(I \times J)$ and the rank distribution r, and $M \in \mathcal{H}(r,\mathcal{P})$. Assume that, for all the subsets $\tau \in I$, $\sigma \in J$, $adm(\tau \times \sigma) \leftrightarrow adm(\sigma \times \tau)$ and also that $(\tau \times \sigma) \in \mathcal{L}(\mathcal{T}(I \times J)) \leftrightarrow (\sigma \times \tau) \in \mathcal{L}(\mathcal{T}(I \times J))$. Then, $M^T \in \mathcal{H}(r',\mathcal{P}')$ with the rank distribution r' defined by $r'(\sigma \times \tau) := r(\tau \times \sigma)$.

Prior to closing this section, it is important to remark a negative property which impedes employing pivoting techniques in $\mathcal{H}$-Arithmetic.

Proposition 2.5. (Prevention from pivoting) Let $\mathcal{H}(r,\mathcal{P}) \in \mathbb{R}^{I \times J}$ be the set of $\mathcal{H}$-Matrices with respect to the partition $\mathcal{P} \subset \mathcal{T}(I \times J)$ and the rank distribution r, and $M \in \mathcal{H}(r,\mathcal{P})$. The application of any general permutation matrices $\prod_I$ and $\prod_J$ destroys the hierarchical structure of M, which means that $M \cdot \prod_J$, $\prod_I^T \cdot M$ or $\prod_I^T \cdot M \cdot \prod_J$ do not belong to $\mathcal{H}(r,\mathcal{P})$. The only exception is applying block-diagonal permutations, which would modify the ordering inside the blocks of $\mathcal{P}$ but keep the resulting matrix in $\mathcal{H}(r,\mathcal{P})$, as the block partition is given by $\mathcal{L}(\mathcal{T}(I))$ for $\prod_I$, and by $\mathcal{L}(\mathcal{T}(J))$ for $\prod_J$.

Chapter 3

Matrix Computations and Parallelism

Contents of the chapter

"*A series of mathematical steps, especially in a computer program, which will give you the answer to a particular kind of problem or question*". That is the definition given by Collins dictionary[1] to the term "algorithm".

In computer science, and particularly in the context of this thesis, algorithms are equivalent to sets of instructions that, performed by the computer, provide an answer to the target problem. However, providing an answer is not enough, and different algorithms could reach the proper answer, whereas not all of them are *interesting* or *appropriate* from a computational point of view. Efficiency is the key to figure out how valuable is a certain algorithm and evaluate its suitability.

In this chapter, we briefly review the basics to achieve efficient algorithms and evaluate them, followed by an introduction to the block-based algorithms for the operations shown in the Chapter 2. Afterwards, we present some tools and keys to reach good parallel efficiency.

More detailed descriptions of linear algebra algorithms, and particularly matrix computations algorithms, can be found in [51, 76].

[1]https://www.collinsdictionary.com/

3.1 The basics to implement efficient algorithms

When considering different approaches to implement specific algorithms to perform linear algebra operations, some of the most important factors which determine their efficiency are:

1. Data layout: storage and access.
2. Optimized formulation of the algorithms.
3. Precision and accuracy.

These items are described in detail in the following sections.

3.1.1 Data layout: storage and access

One of the most important aspects when designing a code pursuing high efficiency is the way data is stored and accessed. On the one hand, the original features of the data should be taken into account to design the layout of the data in memory. For instance, in the case of a symmetric matrix, it is not necessary to store all the elements, but only about half of them (concretely, the main diagonal plus either the entries in the strictly upper or lower triangular part of the matrix); other specialized layouts are possible for band or triangular matrices. Sparse matrices are usually stored in such a way that only the non-zero elements are stored.

Table 3.1 summarizes the memory cost (in terms of the amount of elements that should be stored) when considering the different matrix types in Chapter 2.

Matrix type	**Dimension**	**#Elements to store**
Dense	$m \times n$	$m \times n$
Symmetric	$n \times n$	$\frac{n(n+1)}{2}$
Diagonal	$n \times n$	n
Band	$n \times n$, with bandwidth p,q	$n + \frac{p(2n-p-1)}{2} + \frac{q(2n-q-1)}{2}$
Triangular	$n \times n$	$\frac{n(n+1)}{2}$
Sparse	$m \times n$, with k nonzero entries	For example, in CSR format[2]: 3 vectors R, C, V to respectively store the row and column indices of the nonzero entries, and the associated values. The total number of elements that are stored is $2k + n + 1$.
Low-rank matrix	$m \times n$, rank k	2 rectangular matrices $U_{m\times k}$, $V_{k\times n}$: $m \times k + k \times n$

Table 3.1: Amount of elements that should be stored according to the matrix type.

When storing the elements that form a vector or a matrix, the common strategy is to maintain all the elements in memory in a “contiguous” (linearised) array vectorized form. This implies that a matrix of dimension $m \times n$ will be stored as if it was a long vector of length equal to $m \times n$.

[2]Sparse matrices are out of the scope of this thesis. There exist many alternative representations to the Compressed Sparse Row (CSR) format.

In addition to maintaining only the necessary information according to the specific matrix structure, when choosing how to “arrange” the linearised array, it is important to properly choose the storage ordering. To this end, the Column Major Order (CMO) and the Row Major Order (RMO) are the two common options. When mapping the entries of a matrix into consecutive addresses of the memory, the former one implies that the elements will be stored by columns, and the later one by rows, as defined next.

Definition 3.1. (CMO and RMO) Let A be a matrix of dimension $m \times n$, and consider a vector v of length $l = m \cdot n$ representing the linearised storage of A. Then:

1. A is stored in CMO if:
   $A(i,j) = v(i + j \cdot m)$, $\forall\ 0 \leq i < m$, $0 \leq j < n$.
2. A is stored in RMO if:
   $A(i,j) = v(i \cdot n + j)$, $\forall\ 0 \leq i < m$, $0 \leq j < n$.

It is also important to note that accessing the data in memory should be done consistently with the way it is stored to reduce the cache misses that slow down performance. Let’s discuss this in some detail.

The memory is organized hierarchically. On the bottom there is the disk; followed bottom-up by the main memory; and the cache memory between this and the functional units (where the arithmetic is performed). Taking into account that the cache is a high-speed but relatively small memory, and also that moving data between two levels of the hierarchy introduces a certain overhead, memory movements should be thought carefully and minimized as much as possible, so that the number of times the needed information is not found in cache memory is reduced (and, therefore, the number of cache misses is minimized).

The best way to access data while reducing cache misses is via utilizing *logical* (this is, not existent in the actual structure) blocks when implementing data accesses in the code. By *logical blocks* we mean that the matrix data is accessed as if it was partitioned into (and stored by) blocks (see definition 2.8). Deciding the optimal block size is complex and depends on the structure size and typology, problem to be solved, transfer rates between different memory levels, and CMO vs. RMO storage, among others. However, the unconditional principle that governs the block size selection is the goal to maximize the amount of arithmetic operations that can be done without needing to perform a new memory transfer; that is, maximizing the re-utilization of the data present in cache by introducing regular memory access patterns through *blocking* the algorithms.

Details about block-based algorithms will be provided in Section 3.2.

3.1.2 Libraries for mathematical computations

Highly optimized implementations of the algorithms for fundamental linear algebra operations are provided by instances of the Basic Linear Algebra Subprograms (BLAS) [29, 81] and the Linear Algebra PACKage (LAPACK) [80] such as that in Intel’s MKL [93]. Instead of *reinventing the wheel*, it is convenient to use these existing libraries in order to ensure fair performance.

BLAS is a software package/interface that provides routines for basic vector and matrix operations. The BLAS definition is split into three levels: the Level 1 BLAS is focused on vector and vector-vector operations; the Level 2 BLAS is dedicated to matrix-vector operations; and the Level 3

BLAS comprises matrix-matrix operations. LAPACK is a software package which offers routines for solving systems of linear equations, least-squares solutions of problems, eigenvalue problems, and singular value problems; as well as perform matrix decompositions and related computations.

MKL provides a whole set of implementations of the routines specified in the BLAS and LAPACK optimized for Intel processors. It is important to note that MKL adheres to the interfaces defined by BLAS and LAPACK, so consequently no code changes are needed.

3.1.3 Precision and accuracy

The Institute of Electrical and Electronics Engineers (IEEE) defines computer floating point arithmetic in the 754 standard [71]; concretely, the IEEE 754-2019 [1] is the current active version. This standard covers, among other terms, three floating point precision types:

- Half precision (half), which employs 16 bits per value.
- Single precision (float), which employs 32 bits per value.
- Double precision (double), which employs 64 bits per value.

These precisions are respectively named as `binary16`, `binary32` and `binary64` in the standard.

The range of values that each floating point precision can cover is very different and increases when using more bits per value. The same applies to the precision, and thus the accuracy of the values. However, obviously, numbers represented with larger amounts of bits also consume more memory space and require more computational time when involved in arithmetic operations. The selected precision should be chosen carefully, according to the specific application requirements, and it has a far from despicable effect on performance.

3.2 Basic block algorithms

We next present a key tool to attain high performance, in the form of block-based algorithms for the basic operations described in Chapter 2.

There are two basic (non-blocked) algorithms: SAXPY and GAXPY, that need to be defined, as they will be recurrently employed when constructing block-based algorithms involving vectors and matrix operations.

Algorithm 3.1. (**SAXPY Algorithm**) Let x,$y \in \mathbb{R}^n$ be two vectors and $\alpha \in \mathbb{R}$ a scalar value. The procedure of updating x with αy, this is $x := x + \alpha y$, is called *SAXPY* and the corresponding algorithm is as follows:

```
SAXPY algorithm: $x := x + \alpha y$
for $0 \leq i < n$ do
    $x(i) = x(i) + \alpha y(i)$
end for
```

Algorithm 3.2. (**GAXPY Algorithm**) Let $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$ be two vectors, and $A \in \mathbb{R}^{m\times n}$ a real matrix. Then, the procedure of updating x with Ay, this is $x := x + Ay$, is called *GAXPY*, and can be expressed as a sequence of SAXPYs, as detailed in the following algorithm:

```
**GAXPY algorithm (row-oriented):** $x := x + Ay$
for $0 \leq i < m$ do
    for $0 \leq j < n$ do
        $x(i) = x(i) + A(i,j)y(j)$
    end for
end for
```

Note that this algorithm can be reformulated to proceed by columns instead of rows:

```
**GAXPY algorithm (column-oriented):** $x := x + Ay$
for $0 \leq j < n$ do
    for $0 \leq i < m$ do
        $x(i) = x(i) + A(i,j)y(j)$
    end for
end for
```

Depending on how the data is stored (CMO or RMO), it is better to choose a row-oriented or a column-oriented approach, in order to minimize cache misses and avoid harming performance with memory transfer overhead.

Besides, this GAXPY algorithm can be readjusted when A presents a special structure, such as being a symmetric matrix.

Algorithm 3.3. (**GAXPY Algorithm with a symmetric matrix**) Let $x,y \in \mathbb{R}^n$ be two vectors, and $A \in \mathbb{R}^{n \times n}$ a symmetric matrix. Then, the GAXPY procedure for updating x with Ay, this is $x := x + Ay$, can be reformulated as:

```
**GAXPY algorithm (with $A$ symmetric matrix):** $x := x + Ay$
for $0 \leq j < n$ do
    for $0 \leq i < j - 1$ do
        $x(i) = x(i) + A(in - (i+1)i/2 + j)y(j)$
    end for
    for $j \leq i < n$ do
        $x(i) = x(i) + A(jn - (j+1)j/2 + i)y(j)$
    end for
end for
```

Analogous to the vector updates, the matrix updates can also be expressed by employing SAXPY/GAXPY operations.

Algorithm 3.4. (**SAXPY and GAXPY based Matrix-Matrix Product Algorithm**) Let $A \in \mathbb{R}^{m \times r}$, $B \in \mathbb{R}^{r \times n}$, and $C \in \mathbb{R}^{m \times n}$ be three real matrices. Then, the procedure for updating C with the result of multiplying $A \cdot B$, this is $C := C + A \cdot B$, based on a sequence of SAXPY operations, is called *SAXPY Matrix-Matrix Product*, and the corresponding algorithm is as follows:

```
**SAXPY Matrix-Matrix Product algorithm:** $C := C + A \cdot B$
for $0 \leq j < n$ do
    for $0 \leq k < r$ do
        $C(:,j) = C(:,j) + A(:,k) \cdot B(k,j)$
    end for
end for
```

This operation can also be cast in terms of GAXPY operations, as:

```
**GAXPY Matrix-Matrix Product algorithm:** $C := C + A \cdot B$
for $0 \leq j < n$ do
    $C(:,j) = C(:,j) + A \cdot B(:,j)$
end for
```

Now that these two basic computational algorithms have been defined as a composition of some basic linear algebra operations involving vectors and matrices, it is the moment to introduce block-based algorithms.

Notation 3.1. (**Matrix Block Notation**) Let $A \in \mathbb{R}^{m\times n}$ be a real matrix. Then, the block or submatrix which comprises from the elements in row i_p to row i_q, and column $j_{p'}$ to column $j_{q'}$ is represented as $A(i_p : i_q, j_{p'} : j_{q'})$, and equivalently as $A(\alpha,\beta)$ with the matrix index subsets $\alpha = (i_p : i_q)$, $\beta = (j_{p'} : j_{q'})$, where $0 \leq i_p, i_q < m$, $0 \leq j_{p'}, j_{q'} < n$.

Theorem 3.2.1. (**Matrix Block Product Consistency**) Let $A \in \mathbb{R}^{m\times r}$, $B \in \mathbb{R}^{r\times n}$, $C \in \mathbb{R}^{m\times n}$ be three real matrices. Then, if two blocks $A(\alpha, \beta_A)$ and $B(\beta_B, \gamma)$ are taken (note that β_A and β_B should present the same size, let's say k), then it is verified that $C_{\alpha,\gamma} = \sum_{i=1}^{k} A(\alpha,\beta_{A_i})B(\beta_{B_i},\gamma)$.

Thanks to this consistency, it is possible to define the blocked GAXPY algorithm that is now presented.

Algorithm 3.5. (**Blocked GAXPY Algorithm**) Let $A \in \mathbb{R}^{m\times n}$ be a real matrix, and $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$ be two real vectors. Then, the *block-based GAXPY* procedure $x := x + Ay$ can be performed following any of the following two algorithms, according to a row-blocked perspective or a column-blocked perspective, relatively:

```
**Row-Blocked GAXPY Algorithm:** $x := x + Ay$
$\alpha = 0$
for $i = 0 : q-1$ do
    $idx = \alpha : \alpha + m_i - 1$
    $x(idx) = x(idx) + A(idx, :) \cdot y$
    $\alpha = \alpha + m_i$
end for
```

where $x(idx)$ corresponds to $x_i := x_i + A_i y$, with A_i denoting the i-th block row of A, and q corresponding to the number of block rows of A. This is mathematically equivalent to:

$$\begin{pmatrix} x_0 \\ \vdots \\ x_{q-1} \end{pmatrix} = \begin{pmatrix} x_0 \\ \vdots \\ x_{q-1} \end{pmatrix} + \begin{pmatrix} A_0 \\ \vdots \\ A_{q-1} \end{pmatrix} y, \text{ with the block rows } A_i = \begin{pmatrix} A_{\alpha,0} & \cdots & A_{\alpha,n-1} \\ \vdots & \ddots & \vdots \\ A_{\alpha+m_i-1,0} & \cdots & A_{\alpha+m_i-1,n-1} \end{pmatrix},$$

where $\alpha = \sum_{k=0}^{i} m_k$, m_i is the row-length of the i-th block row, and $i = 0 : q-1$.

```
**Column-Blocked GAXPY Algorithm:** $x := x + Ay$
$\beta = 0$
**for** $j = 0 : r-1$ **do**
    $jdx = \beta : \beta + n_j - 1$
    $x = x + A(: ,jdx) \cdot y(jdx)$
    $\beta = \beta + n_j$
**end for**
```

where x corresponds to $x := x + A_j y_j$, A_j denotes the j-th block column of A, and r corresponds to the number of block columns of A. This is mathematically equivalent to:

$$x = x + (A_0 \dots A_{r-1}) \begin{pmatrix} y_0 \\ \vdots \\ y_{r-1} \end{pmatrix}, \text{ with the block columns } A_j = \begin{pmatrix} A_{0,\beta} & \dots & A_{0,\beta+n_j-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,\beta} & \dots & A_{m-1,\beta+n_j-1} \end{pmatrix},$$

when $\beta = \sum_{k=0}^{j} n_k$, n_j is the column-length of the j-th block column, and $j = 0 : r-1$.

In the Row-Blocked GAXPY Algorithm, the q GAXPYs that are performed are *thinner*, while in the Column-Blocked GAXPY Algorithm, the r GAXPYs are *thinner*. Depending on the specific application and data distribution, a careful selection must be made.

Algorithm 3.6. (**Blocked Matrix-Matrix Product Algorithm**) Let A,B,$C \in \mathbb{R}^{n \times n}$ be three square real matrices (the procedure is similar for rectangular ones). Then, the procedure for updating C with the result of multiplying $A \cdot B$, this is $C := C + A \cdot B$, can be performed via the block algorithm that follows, where the square algorithmic block size is l:

```
**Block Matrix-Matrix Product Algorithm:** $C := C + AB$
**for** $\alpha = 0 : NB-1$ **do**
    $i = \alpha l : (\alpha+1)l - 1$
    **for** $\beta = 0 : NB-1$ **do**
        $j = \beta l : (\beta+1)l - 1$
        **for** $\gamma = 0 : NB-1$ **do**
            $k = \gamma l : (\gamma+1)l - 1$
            $C(i,j) = C(i,j) + A(i,k) \cdot B(k,j)$
        **end for**
    **end for**
**end for**
```

This can be expressed mathematically as:

$$\begin{pmatrix} C_{0,0} & \dots & C_{0,NB-1} \\ \vdots & \ddots & \vdots \\ C_{NB-1,0} & \dots & C_{NB-1,NB-1} \end{pmatrix} = \begin{pmatrix} C_{0,0} & \dots & C_{0,NB-1} \\ \vdots & \ddots & \vdots \\ C_{NB-1,0} & \dots & C_{NB-1,NB-1} \end{pmatrix} + \begin{pmatrix} A_0 \\ \vdots \\ A_{NB-1} \end{pmatrix} (B_0 \dots B_{NB-1}),$$

with $A_i \in \mathbb{R}^{l \times n}$, $B_j \in \mathbb{R}^{n \times l}$, and NB denoting the amount of blocks, this is, $NB = n/l$.

3.3 Parallelising algorithms: tools and keys

In April 2020 it was the 55th anniversary of the origin of one of the most important assertions in computer science: Moore's Law [94]. According to Moore's statement in [95], which polished the original formulation, this Law states that "*the number of transistors in a dense integrated circuit doubles about every two years*".

This could initially be understood as a synonym for "doubling processing speed every two years"; and in some sense this was true for many years, as transistor size was approximately halved every two years. Consequently a greater amount of transistors fit into a chip, whose computational power increased accordingly. For example, Intel's first microprocessor, which was launched in 1971 under the name *4004*[3], could ran at 740 kHz, contained 2,300 transistors and its size was 10,000 nm (nanometers); in contrast, the *IBM Power10* processor, announced in the 2020 Hot Chips International Conference[4], will be equipped with transistors of 7 nm (this is, less than 1000 times the size of the transistors employed in 1971). Moreover, there exist public announcements regarding 3nm transistors being developed by Taiwan Semiconductor Manufacturing Company (TSMC) as well as by Samsung.

However, as transistors shrinking advanced, the physical limitations and issues gained importance so that it has become extremely costly (both in terms of economy and research) to maintain Moore's Law. This is the reason why, in current systems, the strategies which are applied to attain high efficiency cannot only depend on transistor size shrinking, but in other features. Figure 3.1 reflects an analysis of the microprocessor trend in the past 48 years regarding transistors growth, single thread performance, frequency, power consumption, and number of cores. In the context of this thesis, software tools are analysed and employed, but it is important to remark that most of them are useful thanks to the increment in the number of processor cores (parallelism) to perform the computations.

The following sections present some of the software tools and keys that can be utilized to improve the scalability of parallel codes.

3.3.1 Tools

The software tools employed in this thesis are mainly threaded parallel MKL Library function calls; parallel programming models (and runtime systems) utilized in shared memory parallelizations, such as OpenMP and OmpSs (on top of the Nanos++ runtime system in the case of the second); and StarPU. These tools are next presented in short detail.

3.3.1.1 Threaded Intel MKL

Apart from the sequential version of the Intel MKL, Intel also offers a multithreaded version of this library. When an application is enhanced with MKL calls, it only needs to be properly linked with the multithreaded version of the library, and set the necessary environment parameters to enable the parallel execution of the called functions.

[3]https://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html
[4]https://www.anandtech.com/show/15806/hot-chips-32-2020-schedule-tiger-lake-xe-power10-xbox-series-x-tpuv3-jim-keller

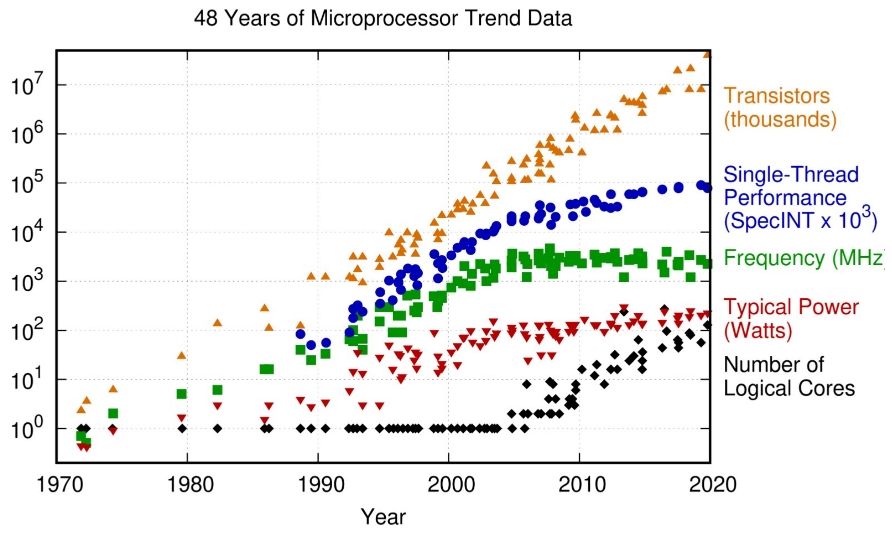

Figure 3.1: Analysis of trends in the past 48 years regarding transistors growth, single thread performance, frequency, power consumption, and number of logical cores. The original data, collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten (until 2010), and H. Rupp (afterwards), and the figure are available at: `https://github.com/karlrupp/microprocessor-trend-data`.

3.3.1.2 OpenMP Application Programming Interface (OpenMP API)

As described in the official specifications, the OpenMP Application Programming Interface (API) [99] is the "*de facto standard for shared-memory on-node programming of systems from embedded systems to largest supercomputers that are equipped with accelerator devices*". This API allows to utilize the cores of a shared memory environment by adding simple directives to codes written in `C`, `C++`, or `Fortran`.

In general, all OpenMP directives begin with `#pragma omp`. These are the most important ones employed or referenced in this thesis, together with their purpose:

- `#pragma omp parallel [clause]` serves to define a parallel region.
- `#pragma omp single` specifies that a certain region will only be executed by one thread.
- `#pragma omp parallel for` annotates that the specific for loop iterations should be spread along the available threads to execute them in parallel.
- `#pragma omp barrier` forces a synchronization point, this is, all the threads should reach that directive before any of them progresses in the execution.
- `#pragma omp task [clause]` specifies a collection of instructions coupled under a unique job to be executed by one thread, and (possibly) in parallel with other tasks.
- `#pragma omp taskwait` is equivalent to a barrier in the context of parallel tasks being executed.
- `#pragma omp atomic` ensures that the access performed to a specific data location is atomic, avoiding simultaneous accesses from diverse threads, which could cause inconsistencies (in case they all updated the data location).

Also, the environment variable `OMP_NUM_THREADS` can be used to fix the amount of threads to be utilized by OpenMP, as with `omp_set_num_threads()`.

Moreover, in order to enable OpenMP, either the flag `-fopenmp` (for GNU compilers) or `-qopenmp` (for Intel compilers) should be included when compiling; otherwise, the compiler will ignore the OpenMP directives, as if they were simple comments.

3.3.1.3 OmpSs

OmpSs [98] is a parallel programming model developed at and maintained by the Barcelona Supercomputing Center (BSC) [22] to exploit parallelism in shared memory environments, including asynchronous parallelism and heterogeneity on devices such as GPUs and FPGAs.

As it will be detailed in a later chapter, in the context of this thesis, the main interest on utilizing OmpSs in contrast to OpenMP is to leverage its novel features (such as weak dependencies and early release), which permit to extract additional task-level concurrency when traversing the $\mathcal{H}$-Matrix hierarchy.

The OmpSs directives are very similar to those in OpenMP, with the exception that they begin with `pragma oss` instead of `pragma omp`. When compiling a code enriched with OmpSs directives (which can be either written in `C` or `C++`), the Mercurium compiler (`mcc`) [92] needs to be employed to utilize the runtime Nanos++ [96], and also the `--ompss-2` flag (otherwise, the compiler will disregard OmpSs directives as if they were mere comments).

3.3.1.4 StarPU

StarPU is an open source library written in C, targeting heterogeneous architectures with Processing Units, such as Central Processing Units (CPUs), GPUs, Synergistic Processing Unit (SPU), etc. Particularly, StarPU views an application as a task graph, and the runtime is in charge of handling its parallel execution following a Sequential Task Flow (STF), based on the application-provided tasks. The runtime schedules the tasks efficiently according to a given scheduling policy (different options can be chosen by the user), and also performs the data transfer between CPUs/GPUs.

In StarPU submitting tasks requires to define 1) specific data structures, named *codelets*, which describe the computational kernel to whom they are associated; and 2) a collection of tasks augmented with a description of the employed codelets, which and how (read/write/both) data are accessed, optionally a callback function to be called when the task is completed (this is necessary because submitting a task to StarPU is a non-blocking operation), and (if necessary) extra information such as priority. Generally, codelets can be viewed as a set of pointers to the alternative implementations of the same theoretical function, and tasks as their scheduled execution on some data handlers.

Different StarPU scheduling policies are employed in the context of this thesis, particularly `prio`, because, as it will be exposed in later chapters, it is the option that offered the best results. Selecting a scheduling policy for StarPU is done trhough the environment variable `STARPU_SCHED`, and the following (non performance modelling) options are available:

- `eager`: the scheduler uses a single central task queue, from which the workers concurrently draw tasks. A task whose priority is different from zero is put at the front of the queue.

- `random`: the scheduler uses a task queue per worker, and assigns the tasks randomly according to the worker's overall performance.
- `ws` (work stealing): the scheduler uses a task queue per worker, and assigns each task to certain workers, by default. Afterwards, a worker that becomes idle steals a tasks from the most loaded worker.
- `lsw` (locality work stealing): this is equivalent to `ws`, but the *task robbery* is performed from a neighbour worker instead, and priorities are taken into account.
- `prio (priorities)`: the scheduler uses a single central task queue, from which the workers concurrently draw tasks, but those are sorted according to their programmer-specified priority (with values between -5 and +5).
- `heteroprio`: the scheduler uses different priorities for the different processing units, so it must be properly configured.

The most important StarPU functions are:

- `starpu_init()` to initialize StarPU.
- `starpu_task_create()` to allocate and fill (but not submit) the task structure with the default settings.
- `starpu_task_submit(struct starpu_task *task)` to submit a specific task to StarPU. It must be noted that, in case a blocking call is desired, either 1) the `synchronous` parameter of the `task` is set to 1, or 2) `starpu_task_wait()` is called after submitting the task, to force a synchronization barrier.
- `starpu_shutdown()` to terminate StarPU.

Moreover, MPI transfers can be easily integrated in a code which uses StarPU by employing the library `libstarpumpi`, which basically converts `MPI_*` functions to StarPU equivalents.

More details about StarPU can be found in its Handbook[5]. It only remains to add that, when compiling codes that employ StarPU directives, the flags `$(pkg-config --cflags starpu-1.3)` and `$(pkg-config --libs starpu-1.3)` are required.

3.3.2 Keys for parallelism success

The keys that need to be properly addressed in order to extract parallelism that derives in high performance (in the context of this thesis) are:

1. Parallel programming model selection.
2. Synchronization.
3. Load balancing.

[5] `https://files.inria.fr/starpu/doc/starpu.pdf`

4. Data motion overheads.

Regarding the parallel programming model and/or implementation selection, it is very important to know the target application. In general, the most standardised model could be thought to be the most efficient one; however, as it happened in the context of this thesis and will be described in later chapters, certain features offered by more specific programming models (such as OmpSs or StarPU) can be more beneficial due to the application particularities, but may not be available (yet) in standards like OpenMP. At the end, the programmer needs to choose carefully between standardised and stable but (possibly) more inflexible models versus less established options enhanced with newer/under development beneficial features.

Synchronization and load balancing are usually two issues that generate significant bottlenecks, overhead, and wasted processor cycles. In many applications there exist difficulties derived from a wide variety of issues such as the existence of data dependencies between different tasks, massive message exchanges between processes, coarse-grain tasks, remarkable differences between the iterations that form a loop that is parallelised, etc. All these scenarios complicate the load balancing when different threads are executed concurrently and need to be analysed in detail, in order to check if a different parallel approach could reach a more equitable workload distribution among processors. Besides, aspects such as the block size (when operating with block-based algorithms), the order in which certain tasks are executed (in task-based parallel implementations), or joining/splitting certain tasks can also contribute to yield higher parallel performance by reducing not only the workload unbalance but also the need to synchronise the threads.

Many parallel programming models provide support for performance tools that allow to visualize traces, which help the user to monitor the (parallel) executions. For instance, StarPU can generate traces using FxT [46], which can be viewed with Visual Trace Explorer (ViTE) [114]; and OmpSs also provides support for Paraver [100], whose purpose is similar. These tools can be very helpful when detecting idle periods due to synchronization issues (which could be redundant or avoidable after a deeper analysis) as well as performance deteriorations due to imbalanced workloads.

Lastly, it is also very important to control the amount of data that is moved between the different processes, and also the dependencies annotated when employing task-based approaches. Unnecessary data movements or data dependencies could generate insurmountable overheads that critically damage performance. Tools to analyse traces can also be very helpful when trying to solve those issues, together with a good understanding of the graph that defines the task dependencies (with the purpose of limiting the edges to the essential ones) in the applicable cases.

3.4 Measuring performance

To evaluate the performance of the algorithm's implementations, one of the common measures to take into account is the total execution time. Another interesting metric is the floating point operations per second (FLOPS). This section introduces some details regarding time- and FLOPS-based performance evaluations.

3.4.1 Execution time and speedup, efficiency, and scalability

If a certain code can be executed faster while keeping the quality of the desired results (such as accuracy), then it is better than any of the slower ones. Thus, the elapsed time between the start and the end of its execution is one of the measures that will be used in this thesis to reflect performance improvements.

Moreover, part of the execution time analysis will be based on comparisons between different versions of the same algorithm implementation, such as, for example, sequential vs. parallel execution, or MKL vs. non-MKL based codes. In this context, the *speedup* will be a metric that is leveraged for time comparisons, as defined next.

Definition 3.2. (Speedup) Let $t_1, t_2 \in \mathbb{R}$ be two time measurements. Then, the *speedup* or ratio of t_1 with respect to t_2 is computed as $S = t_1/t_2$.

Particularly, when measuring the benefits from parallelizing an algorithm, the *efficiency* is particularly valuable to evaluate the benefits of the applied parallelization.

Definition 3.3. (Efficiency) Let $t_s, t_p \in \mathbb{R}$ be two time measurements, respectively corresponding to the sequential and parallel execution times. Also, let $p \in \mathbb{N}$ be the number of processors employed in the parallel execution. Then, the *efficiency* of the parallel algorithm is defined as the fraction of time during which a processor is usefully utilized, this is: $E = \frac{t_s}{pt_p} = S/p$.

Typically, the efficiency decays as the number of processes involved in the execution is increased. This maintenance or decay of the efficiency is referred to as *scalability.* In other words, the scalability of a parallel implementation is the measure of its capacity to increase the speedup in proportion to the number of processors. The maximum speedup attainable for a given number of processors is presented in the next theorem.

Theorem 3.4.1. (Amdahl's Law) Let $0 \leq f \leq 1$, $f \in \mathbb{R}$ be the ratio of the total execution time which can be parallelized, and $p \in \mathbb{N}$ the number of processors involved in the parallel execution. Then, the maximum speedup attainable is $S_{max} = \frac{1}{(1-f)+\frac{f}{p}}$.

If the efficiency remains constant as the number of processors that take part in a parallel execution is increased, then the implementation is said to be *scalable.* Particularly, the scalability is said to be *linear* if the observed speedup is equal to the maximum value defined by Amdahl's Law; *sublinear* when it is fewer than that; and *superlinear* when it is greater than that.

3.4.2 Floating point operations (flops) and flops per second (FLOPS)

The term FLOPS is the metric that counts the number of effectively executed flops in one second, and it is typically employed to evaluate the performance: more FLOPS implies higher performance.

Table 3.2 comprises the amount of flops for the operations described in Chapter 2, considering matrices of size $m \times n$ (also a second matrix of size $n \times p$ in the matrix-matrix case; $n \times n$ if a square matrix is involved in the operation), and vectors of length n.

It is important to note that, in Table 3.2, floating point operation (flop) counts are based on general cases in which dense matrices are considered. If other types of matrices were employed, the flops may vary. For example, in the case of the matrix-matrix product:

- For a square diagonal matrix, the flops count is n.

- For a square lower triangular matrix, the flops count is $n(n+1)$.
- For a square k-band matrix, the flops count is $2nk$.
- For a sparse matrix, the flops count is $2s$ (worst case: all the non-zero elements are in the same row) or less, with s being the number of non-zero elements.
- For a low-rank matrix of rank k, the flops count is $2k(m+n)$.

Operation	**flops**
Vector-vector addition	n
Scalar-vector product	n
Scalar-matrix product	mn
Dot vector product	$2n$
Matrix-vector product	$2mn$
Matrix-matrix product	$2mnp$
SVD	$O(km^2n + k'n^3)$ with k, k' constants that depend on the SVD algorithm (see [51])
LU Decomposition	$\frac{2}{3}n^3$
Cholesky decomposition	$\frac{1}{3}n^3$

Table 3.2: flops corresponding to the vector and matrix operations described in Chapter 2.

Strictly calculating, flops count would be extended: for instance, matrix-vector product flops would precisely be $(2n-1)m$. However, "$Big - O$" criteria prevails when counting flops, which means that they will only contemplate the biggest order number in the total flops count. These numbers will be the most costly ones and, thus, the ones in which attention is focused on. This principle could be seen as if flops were equivalent to polynomials in which lower-order terms are ignored.

Usually, millions and billions of flops are executed per second and, as a consequence, the prefixes *mega-* (MFLOPS) and *giga-* (GFLOPS) are employed to specify this when reporting actual measurements.

Chapter 4

It all began with prototypes

Contents of the chapter

4.1 Introduction

The first objective we set in the context of this work was determining whether task-based parallel programming models were appropriate for $\mathcal{H}$-Matrices algorithms. In other words, we wanted to test the efficiency of such programming models when dealing with a hierarchically partitioned matrix in which leaf blocks (blocks that constitute leaves in the BCT, so that they are not partitioned into smaller ones) are imbalanced, both in terms of the size and the data distribution (dense versus sparse blocks).

To this end, we developed prototypes of the $\mathcal{H}$-LU [6] and the $\mathcal{H}$-Cholesky [7] factorizations, written in `C` language. Instead of reproducing pure $\mathcal{H}$-Matrices and operating with them, the implementations we designed calculated the LU and Cholesky decomposition of matrices whose structure was actually defined by a hierarchy of nested blocks, with the leaves presenting different block sizes, and being either dense or null-blocks instead of dense or low-rank blocks. For this reason we refer to them as *prototypes*. At that time, we did not intend to *reinvent the wheel*, as there already existed libraries which provided a whole set of implementations to operate over pure $\mathcal{H}$-Matrices, but our purpose was to simulate the data imbalance and hierarchical structure in a simplified scenario. By substituting low-rank blocks by null-blocks, we saved ourselves from implementing all the low-rank algebra operations needed in $\mathcal{H}$-Matrices operations, such as, for example, compression, and were also able to accelerate the testing process.

Thus, our objective of evaluating the efficiency of task-based programming models in this simplified scenario was set with the purpose of determining whether if it was worth it to invest the effort of parallelising the already existing $\mathcal{H}$-Matrices libraries with this approach. Even though the prototypes did not consider low-rank blocks, we could still take into consideration the insights obtained from experimenting with these prototypes, because we were still performing a number of tasks which was similar to those in pure $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky scenarios, as well as reproducing the different workloads associated to tasks involving blocks with different ranks.

In the next sections, we will first describe the prototype structures we designed for $\mathcal{H}$-Matrices, including the storage layout chosen; then the prototype $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky algorithms will be presented, remarking the differences with respect to pure $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky algorithms; after that, we will introduce the parallelization performed over the prototypes; and we will finalize by presenting a performance analysis, and the conclusions obtained from it.

4.2 The prototype $\mathcal{H}$-Matrix structure

One of the most delicate aspects of implementing algorithms to treat with $\mathcal{H}$-Matrices is the storage layout. The structure to represent an $\mathcal{H}$-Matrix needs to be able to cover, at least:

- The hierarchy of the $\mathcal{H}$-Matrix blocks in such a way that it is easy and fast to determine the parent/descendant blocks (if any) of a specific block.
- The blocks which are not leaves in the hierarchy, this is, those that are partitioned into sub-blocks.
- The dense blocks, which require to store the associated dimension (m, n) and the entries.
- The low-rank blocks, which require to store the rank and the entries in the corresponding V and U vectors (see Definition 2.34).

More details about the necessities of the structure to store a pure $\mathcal{H}$-Matrix will be provided in future chapters. Since in this chapter we are only referring to the prototype previously described, the structure to represent the prototype $\mathcal{H}$-Matrix will be simpler, and will only need to cover:

- The hierarchy of the prototype $\mathcal{H}$-Matrix blocks. It is still needed to properly refer to the parent/descendant blocks (if any) of a specific block.
- The blocks which are not leaves in the hierarchy, this is, these that are partitioned into sub-blocks. In contrast to the available software implementations for true $\mathcal{H}$-Matrices, where different partitioning algorithms and admissibility criteria can be employed, and so the hierarchy description needs to cover a wide scope of variations, in our case, when partitioning a block, it will always be split into regular (this is, same dimension) sub-blocks, and we will also utilize the weak admissibility condition (see Definition 2.49).
- The dense blocks, storing the associated dimension (m, n) and entries.

- The null blocks (instead of low-rank), which only require to store the block dimension (m, n).

To store the described structure, we defined the following items:

- An array of `double *values` comprising all the matrix entries stored in CMO by blocks (more details about this will be presented in Section 4.2.1).
- A set of integers and arrays to store the prototype $\mathcal{H}$-Matrix hierarchy information, which include:
  - `int n_levels`: an integer to store the number of levels of the hierarchy.
  - `int total_blocks`: an integer to store the number of blocks in the matrix (all of them, not only the leaves).
  - `int *levels_sizes`: an array of integers to store the different block dimensions, from the smallest to the biggest one.
  - `int *chldtab`: an array of integers to store, for the *i-th* element, the index of the head of the list of the sub-blocks of the *i-th* node.
  - `int *nmchtab`: an array of integers to store, for the *i-th* element, the number of sub-blocks of the *i-th* node.
  - `int *brthtab`: an array of integers to store, for the *i-th* element, the index of the block which is the *left* brother (`null` for the root node), this is, it is either in the same row but in the left column of the *i-th* block, or in the previous row and right column of it (in short, this is the block that is exactly before in the list of sub-blocks of the *i-th* block parent block).
  - `int *rbrthtab`: an array of integers to store, for the *i-th* element, the same as `int *brthtab` but referring to the *right* brother. Note that now the last node stores `null` value.
  - `int *hgthtab`: an array of integers to store, for the *i-th* element, the level it belongs to in the hierarchy (with 1 associated to the root, this is, the highest dimension block, which is equivalent to the whole matrix, and whose size is the last one stored in the `levels_sizes` array).
- An array `struct Mtx_Obj *Matrix_blocks`, composed of all the structures of type `Mtx_Obj` employed to store each block information, which are as follows:

```
struct Mtx_Obj
{
    int struct_vec_index;
    int son_index;
    int i;
    int j;
    int m;
    int n;
    int lda;
    int parent;
```

```
                int isNull;
                int isLeaf;
                union
                {
                        int v_values_index;
                        struct Mtx_Obj *sons;
                };
        };
```

where each of the elements represents, for a sample block `struct Mtx_Obj my_block`:

- `int struct_vec_index`: the index of the specific `Mtx_Obj` in the `Matrix_blocks` array.
- `int son_index`: the index of the specific block (`Mtx_Obj`) in the array of sons of its parent block.
- `int i`, `int j`: the coordinates (i,j) in which the first value of the block is stored in the array `values`.
- `int m`, `int n`: the block dimension ($m \times n$).
- `int lda`: the leading dimension of the specific block, which specifies the distance between two consecutive elements in the same row.
- `int parent`: the index of the parent block in the `Matrix_blocks` array.
- `int isNull`, `int isLeaf`: respectively store value `1` in case the block is null/leaf, or `0` otherwise.
- `union { int v_values_index, struct Mtx_Obj * sons }`: either the index in the array `values` in which this block's first element is stored (in case it is a leaf), or an array of pointers to the sub-blocks (`Mtx_Obj`) of the specific block (if it is not a leaf).

Figure 4.1 shows a simple example of a prototype $\mathcal{H}$-Matrix of dimension 8×8, and the contents of the corresponding structures are represented in Table 4.1. Moreover, Table 4.2 shows a sample of the structures corresponding to four of the thirteen blocks that belong to the example matrix in Figure 4.1.

4.2.1 Storage layout

CMO and RMO (see definition 3.1) are the storage formats that have been traditionally employed for dense matrices. $\mathcal{H}$-Matrices require a special storage layout design, as they lay in between the dense and sparse scenarios. For this reason we present an alternative to CMO/RMO format, that we name *CMO by Blocks* or Block Data Layout (BDL). This alternative approach follows the same ordering as CMO, but instead of directly applying it to the whole matrix entries, it follows the hierarchy block pattern. Figure 4.2 shows the difference between the CMO storage format (left) and BDL (right). We have chosen BDL based on CMO instead of RMO because the computational kernels from BLAS and LAPACK that we will use to perform linear algebra operations adhere to the Fortran convention, which requires matrix operands to be in CMO.

When operating with $\mathcal{H}$-Matrices (also with our prototype), various leaves are sometimes involved in an operation with bigger leaf blocks, and thus there are two ways of proceeding: 1) treat the

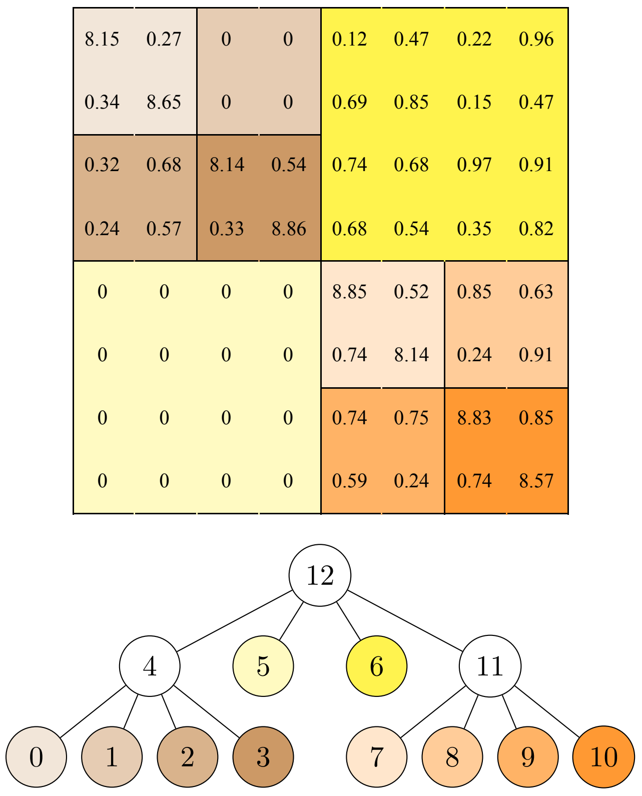

Figure 4.1: Example of a simple 8×8 prototype $\mathcal{H}$-Matrix (above) and its corresponding tree representing the parent/child relationships between all the matrix blocks (below). The leaf blocks are coloured and the non-leaf ones are left white. Each matrix block index is written inside the tree nodes, and it specifies to the index of its associated block structure (of type `Mtx_Obj`) in the `Matrix_blocks` array.

Element	Value
`values`	`{8.15, 0.34, 0.27, 8.65, 0.32, 0.24, 0.68, 0.57, 8.14, 0.33, 0.54, 8.86, 0.12, 0.69, 0.74, 0.68, 0.47, 0.85, 0.68, 0.54, 0.22, 0.15, 0.97, 0.35, 0.96, 0.47, 0.91, 0.82, 8.85, 0.74, 0.52, 8.14, 0.74, 0.59, 0.75, 0.24, 0.85, 0.24, 0.63, 0.91, 8.63, 0.74, 0.85, 8.57}`
`n_levels`	`3`
`total_blocks`	`13`
`levels_sizes`	`{2,4,8}`
`chldtab`	`{-1, -1, -1, -1, 3, -1, -1, -1, -1, -1, -1, 10, 11}`
`nmchtab`	`{0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 4, 4}`
`brthtab`	`{-1, 0, 1, 2, -1, 4, 5, -1, 7, 8, 9, 6, -1}`
`rbrthtab`	`{1, 2, 3, -1, 5, 6, 11, 8, 9, 10, -1, -1, -1}`
`hgthtab`	`{3, 3, 3, 3, 2, 2, 2, 3, 3, 3, 3, 2, 1}`

Table 4.1: Values associated with the matrix in Figure 4.1 as well as all the elements needed to define its hierarchy. The zeros belonging to the null blocks are not included in the `values` array.

Block 0	**Block 4**	**Block 5**	**Block 12**

`struct_vec_index=0`
`son_index=0`
`i=0, j=0`
`m=2, n=2, lda=2`
`parent=4`
`isNull=0, isLeaf=1`
`v_values_index=0`
	`struct_vec_index=4`
	`son_index=0`
	`i=0, j=0`
	`m=4, n=4, lda=8`
	`parent=12`
	`isNull=0, isLeaf=0`
	`v_values_index=0`
		`struct_vec_index=5`
		`son_index=1`
		`i=4, j=0`
		`m=4, n=4, lda=4`
		`parent=12`
		`isNull=1, isLeaf=1`
		`v_values_index=16`
			`struct_vec_index=12`
			`son_index=-1`
			`i=0, j=0`
			`m=8, n=8, lda=8`
			`parent=-1`
			`isNull=0, isLeaf=0`
			`v_values_index=0`

Table 4.2: Sample of the structures (detailing their associated element values) corresponding to four of the thirteen blocks that form the example matrix in Figure 4.1.

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

0	2	8	10	32	36	40	44
1	3	9	11	33	37	41	45
4	6	12	14	34	38	42	46
5	7	13	15	35	39	43	47
16	20	24	28	48	50	56	58
17	21	25	29	49	51	57	59
18	22	26	30	52	54	60	62
19	23	27	31	53	55	61	63

Figure 4.2: Indices representing the ordering in which the elements of the example matrix in Figure 4.1 are stored, according to CMO (left) and CMO by Block/Block Data Layout (right) storage formats.

smaller leaves as if they form a single bigger block of the same size as the bigger leaf block involved in the operation; or 2) perform a partitioning of the bigger leaf block into sub-blocks of size equivalent to the dimension of the smaller leaves size. For example, continuing with the example in Figure 4.1, in case blocks 0 to 3 need to operate with block 5, the mentioned issue will appear. These situations are particularly frequent when operating with $\mathcal{H}$-Matrices, and they force us to choose carefully the storage layout, and also to adapt the algorithms to it. These will be analyzed in Section 4.4.

4.3 The prototype-based algorithms

In $\mathcal{H}$-Arithmetic, the algorithms to solve the LU and Cholesky decomposition essentially follow the same approaches as if the matrices were dense, which implies that the $\mathcal{H}$-Matrix algorithms designed for parallel performance will operate over blocks. However, $\mathcal{H}$-Matrices already include a hierarchy of blocks of different sizes, and this fact will need to be taken into account when implementing these block-based implementations.

Particularly for the $\mathcal{H}$-LU, a description of how the Block Right-Looking (BRL) algorithm (described next in the Subsection 4.3.1) to solve the LU decomposition of dense matrices is adjusted and applied in scenarios involving $\mathcal{H}$-Matrices can be found in the literature [79]. The main adaptation consists in modifying the progress of the operations to follow the hierarchy and consider irregular block sizes. Equivalently, in [27] there is a description of how to perform the $\mathcal{H}$-Cholesky, which follows the same strategy as the one described for the $\mathcal{H}$-LU.

In our case of study, we will employ the same algorithms as if pure $\mathcal{H}$-Matrices were being used, but simplifying the (sub)operations that operate with low-rank blocks, as we will instead have null blocks. This will be exposed in detail in the following sections.

4.3.1 $\mathcal{H}$-LU and *Prototype* $\mathcal{H}$-LU

Algorithm 1 shows the BRL procedure to compute the LU factorization of a square matrix of dimension n. In the algorithm, the main operations that correspond to three basic linear algebra building blocks (or computational kernels) are highlighted. Particularly, the orange operation corresponds to the LU factorization; the blue ones involve solving triangular systems of equations (with unit lower triangular factor in line 4, and upper triangular factor in line 7); and the purple one is a matrix-matrix multiplication.

Algorithm 1 BRL algorithm for the LU factorization.

```
Require: $A \in \mathbb{R}^{n \times n}$
 1: for $k = 0,1,2,\dots,n_t - 1$ do
 2:     $A_{kk} = L_{kk}U_{kk}$
 3:     for $j = k+1,k+2,\dots,n_t - 1$ do
 4:         $U_{kj} := L_{kk}^{-1}A_{kj}$
 5:     end for
 6:     for $i = k+1,k+2,\dots,n_t - 1$ do
 7:         $L_{ik} := A_{ik}U_{kk}^{-1}$
 8:     end for
 9:     for $i = k+1,k+2,\dots,n_t - 1$ do
10:         for $j = k+1,k+2,\dots,n_t - 1$ do
11:             $A_{ij} := A_{ij} - L_{ik} \cdot U_{kj}$
12:         end for
13:     end for
14: end for
```

As it was explained in Chapter 3, LAPACK and BLAS routines will be invoked to perform the linear algebra operations. The ones associated with these three operations are:

- Routine _GETRF from LAPACK to perform the LU factorization (highlighted in orange in Algorithm 1).
- Routine _TRSM from BLAS to solve the triangular systems of equations (highlighted in blue in Algorithm 1).
- Routine _GEMM from BLAS to perform the matrix-matrix multiplications (highlighted in purple in Algorithm 1).

The adaptation of that algorithm to compute the LU decomposition of $\mathcal{H}$-Matrices is presented in Figure 4.4 for an $\mathcal{H}$-Matrix with the same hierarchy of blocks as that in Figure 4.1, with the blocks indices as in Figure 4.3. In that figure, the colors associated to each operation in Algorithm 1 are equivalently employed to indicate the corresponding operation that is being performed over each block: light gray-colored blocks already have the final result; and the blocks filled in dark gray are the ones whose values are being used in the particular operation. Note that only the main operations included in the BRL algorithm are represented, while complementary ones due to the presence of low-rank blocks, such as SVD for re-compressing data, are not included.

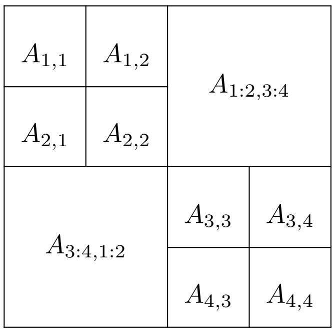

Figure 4.3: Example of an 8×8 $\mathcal{H}$-Matrix with the same block structure as that in Figure 4.1.

In the case of our prototype $\mathcal{H}$-Matrix, having null blocks implies that: 1) the corresponding data (zeros) is not stored in memory; and 2) some operations do not need to be performed, as they would return a null block. However, at certain points of the computation, it can occur that a null block becomes not null. For example, if the block $A_{3,4}$ in Figure 4.3 is null, but $A_{1:2,3:4}$ and $A_{3:4,1:2}$ are not, then after the GEMM operation $A_{3:4,3:4} = A_{3:4,3:4} - L_{3:4,1:2}U_{1:2,3:4}$, the block $A_{1:2,3:4}$ will fill in. Our implementation of the prototype $\mathcal{H}$-LU performs a pre-processing of the matrix in order to identify in advance which null blocks will fill in during the computations. When detecting these situations, the pre-processing inserts zeros in the array storing the values of the matrix to accommodate the space as if the null block was stored. The objective of this preprocessing is ensuring that there already exist sufficient storage space in memory to store its values when the block becomes not null.

4.3.2 $\mathcal{H}$-Cholesky and *Prototype* $\mathcal{H}$-Cholesky

The block-based algorithm to perform the $\mathcal{H}$-Cholesky decomposition is, analogously to the $\mathcal{H}$-LU, an adaptation of the BRL algorithm for the Cholesky factorization shown in Algorithm 2, and is commonly employed to operate with dense matrices.

The LAPACK and BLAS routines that are called to perform the three main operations of this algorithm are:

- Routine _POTRF from LAPACK to perform the Cholesky factorization (highlighted in orange in Algorithm 2).
- Routine _TRSM from BLAS to solve the triangular system of equations (highlighted in blue in Algorithm 2).
- Routine _SYRK from BLAS to perform the symmetric matrix-matrix multiplication (highlighted in purple in Algorithm 2).

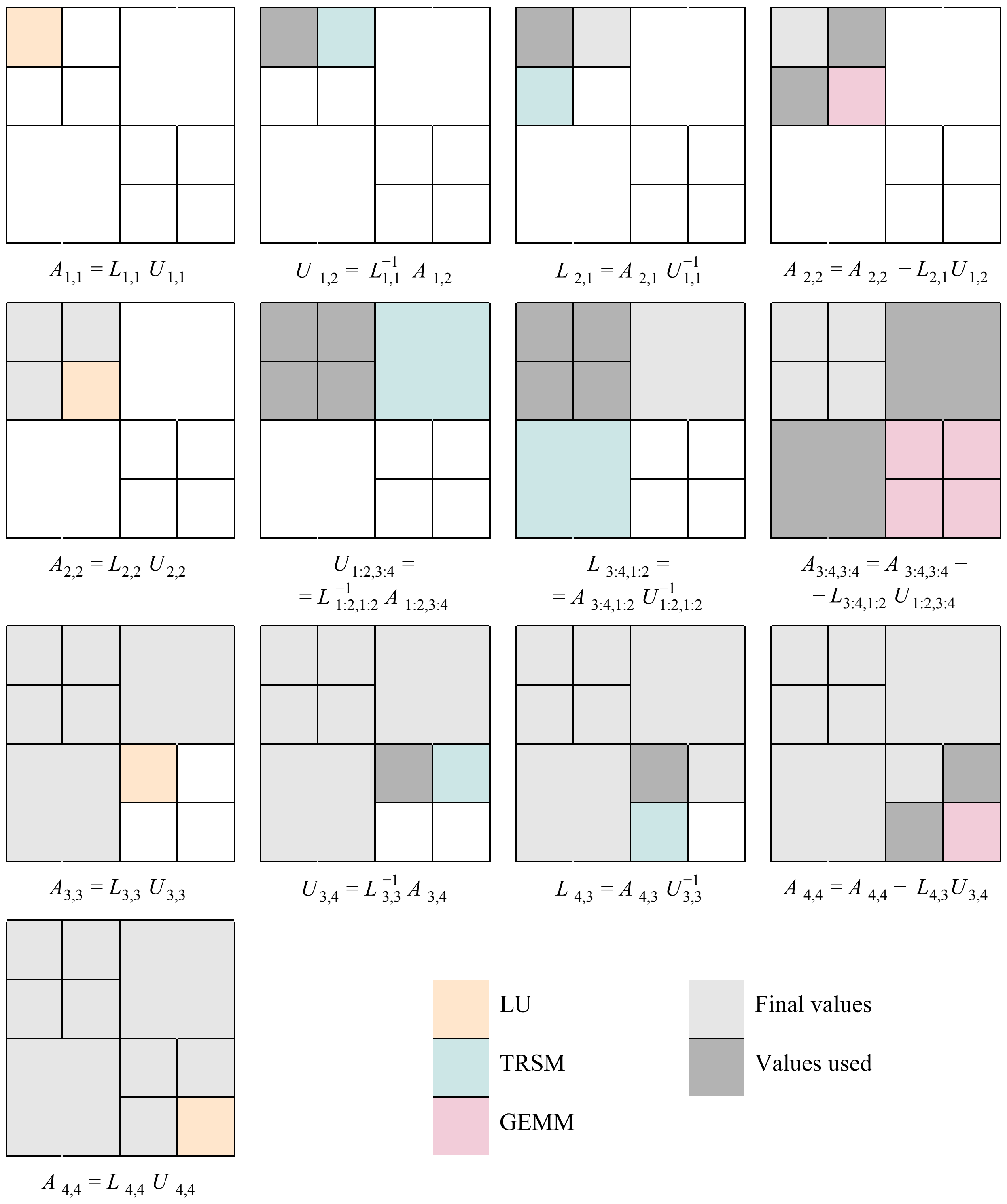
$A_{1,1} = L_{1,1} U_{1,1}$
$U_{1,2} = L_{1,1}^{-1} A_{1,2}$
$L_{2,1} = A_{2,1} U_{1,1}^{-1}$
$A_{2,2} = A_{2,2} - L_{2,1} U_{1,2}$
$A_{2,2} = L_{2,2} U_{2,2}$
$U_{1:2,3:4} = = L_{1:2,1:2}^{-1} A_{1:2,3:4}$
$L_{3:4,1:2} = = A_{3:4,1:2} U_{1:2,1:2}^{-1}$
$A_{3:4,3:4} = A_{3:4,3:4} - - L_{3:4,1:2} U_{1:2,3:4}$
$A_{3,3} = L_{3,3} U_{3,3}$
$U_{3,4} = L_{3,3}^{-1} A_{3,4}$
$L_{4,3} = A_{4,3} U_{3,3}^{-1}$
$A_{4,4} = A_{4,4} - L_{4,3} U_{3,4}$
$A_{4,4} = L_{4,4} U_{4,4}$
LU
TRSM
GEMM
Final values
Values used

Figure 4.4: Graphical representation of the steps to compute the $\mathcal{H}$-LU of the sample $\mathcal{H}$-Matrix in Figure 4.3 following an adapted version of the BRL LU decomposition algorithm.

Algorithm 2 BRL algorithm for the Cholesky factorization.

```
Require: $A \in \mathbb{R}^{n \times n}$
 1: for $k = 0,1,2,\ldots,n_t - 1$ do
 2:     $A_{kk} = L_{kk}L_{kk}^T$
 3:     for $i = k+1,k+2,\ldots,n_t - 1$ do
 4:         $L_{ik} := A_{ik}L_{kk}^{-T}$
 5:     end for
 6:     for $i = k+1,k+2,\ldots,n_t - 1$ do
 7:         for $j = k+1,k+2,\ldots,i$ do
 8:             $A_{ij} := A_{ij} - L_{ik}L_{jk}^T$
 9:         end for
10:     end for
11: end for
```

Again, as with the $\mathcal{H}$-LU, the adaptation of this algorithm to compute the $\mathcal{H}$-Cholesky simply considers the hierarchy and different block sizes when traversing the blocks. Figure 4.5 reflects these adjustments graphically. The color correspondence is the same already described for the $\mathcal{H}$-LU. Also note that only the main operations are represented, while the *auxiliary* ones related to low-rank arithmetic, such as re-compressing, are omitted.

Analogously to the $\mathcal{H}$-LU, the presence of null blocks in our prototype $\mathcal{H}$-Matrix does not require storing part of the data; however, we perform a preprocessing of the matrix information to allocate space in memory for those block entries that are initially null, but fill in during the execution after a SYRK operation.

4.4 All this seems correct... but there are hidden issues

When studied separately, the storage layout based on the BDL format described in Section 4.2, and the adapted BRL algorithms for the $\mathcal{H}$-LU and the $\mathcal{H}$-Cholesky presented in Section 4.3 seem correct and, generally, they are. However, there are some *hidden issues* that need to be analyzed in more detail, and will force us to either refine the storage layout or the algorithms' flow. Let's illustrate this with an example.

Consider the $\mathcal{H}$-Matrix we have used for the examples (Figure 4.1) with the entries stored as shown in the right-hand side scheme in Figure 4.2, and block indices annotated as in Figure 4.3. In addition, let's pay detailed attention to the `TRSM` operation of the $\mathcal{H}$-LU factorization located in the sixth step in Figure 4.4, this is, computing $U_{1:2,3:4} = L^{-1}_{1:2,1:2}A_{1:2,3:4}$. To facilitate the abstraction, Figure 4.6 summarizes all the necessary information. Note that this situation also occurs when computing the $\mathcal{H}$-Cholesky factorization, in an equivalent manner, as well as for some `GEMM` operations.

If the steps listed in Section 4.3 are strictly followed, then the `TASK_TRSM` that computes $U_{1:2,3:4} = L^{-1}_{1:2,1:2}A_{1:2,3:4}$ will *group* blocks $A_{1,1}$, $A_{1,2}$, $A_{2,1}$, and $A_{2,2}$ (let's say forming $A_{1:2,1:2}$) and use this union to directly operate over $A_{1:2,3:4}$, as it will present the same size as $L^{-1}_{1:2,1:2}$ and, therefore, BLAS/LAPACK routines can be employed. However, $A_{1:2,1:2}$ will not have its entries in CMO but in BDL storage format, and this will cause problems when interacting with the entries in $A_{1:2,3:4}$, which are stored in CMO because that is a leaf block; for example the first element of the third

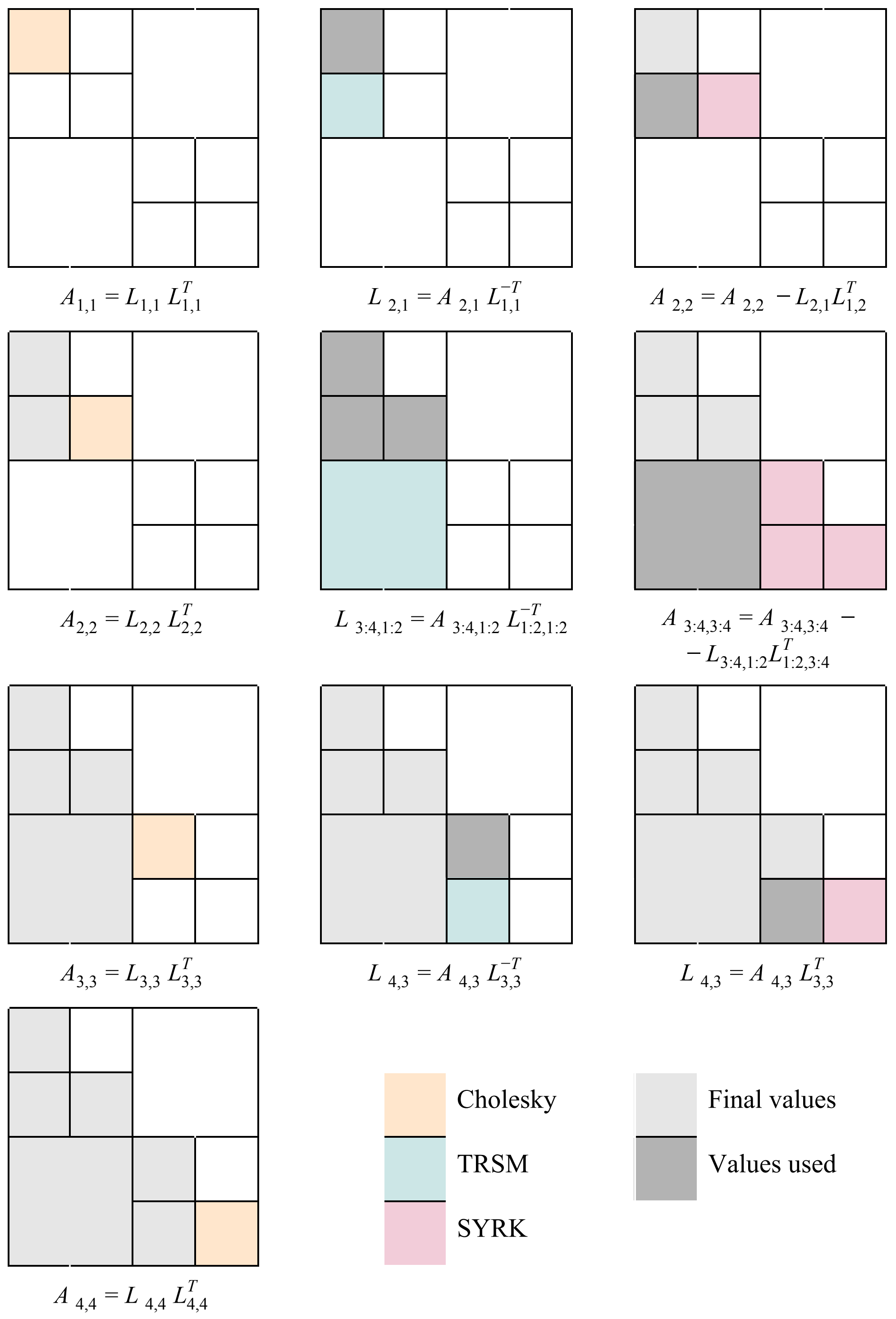
$A_{1,1} = L_{1,1} L_{1,1}^T$
$L_{2,1} = A_{2,1} L_{1,1}^{-T}$
$A_{2,2} = A_{2,2} - L_{2,1} L_{1,2}^T$
$A_{2,2} = L_{2,2} L_{2,2}^T$
$L_{3:4,1:2} = A_{3:4,1:2} L_{1:2,1:2}^{-T}$
$A_{3:4,3:4} = A_{3:4,3:4} - - L_{3:4,1:2} L_{1:2,3:4}^T$
$A_{3,3} = L_{3,3} L_{3,3}^T$
$L_{4,3} = A_{4,3} L_{3,3}^{-T}$
$L_{4,3} = A_{4,3} L_{3,3}^T$
$A_{4,4} = L_{4,4} L_{4,4}^T$
Cholesky
TRSM
SYRK
Final values
Values used

Figure 4.5: Graphical representation of the steps to compute the $\mathcal{H}$-Cholesky of the sample $\mathcal{H}$-Matrix in Figure 4.3 following an adapted version of the BRL Cholesky decomposition algorithm.

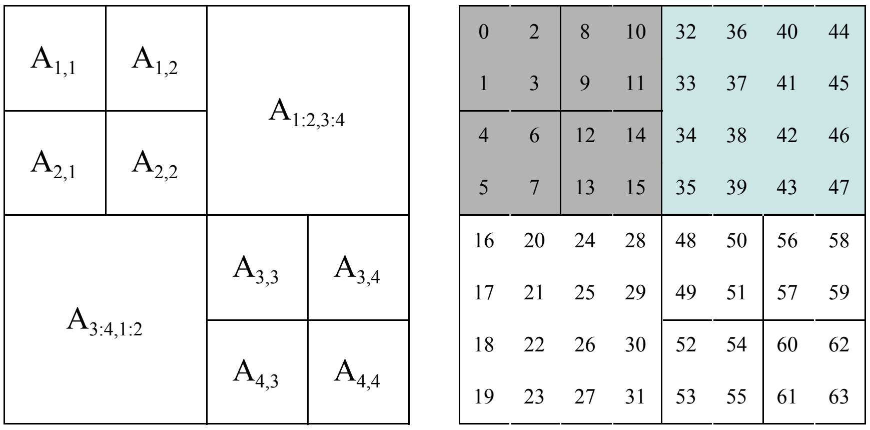

Figure 4.6: Summary of the needed data from previous examples to analyse the possible storage layout and adapted BRL algorithms steps issues, including the matrix sub-blocks indices (left), and the `TRSM` operation (right). Note that the storage ordering is shown, and the colors previously associated to each of the operations are maintained.

row is actually stored in the fifth position of the `values` array, instead of occupying the third one, as is required to correctly call the `TRSM` routine.

To avoid these conflicts, there are only two options: 1) re-formulate the storage layout; or 2) sub-partition the leaf blocks into the smallest sub-block size participating in each specific step of the algorithm, in such a way that using the leading dimension of the big leaf block permits to properly indicate the elements that operate with each of the smallest sub-blocks.

We considered the first option, but soon discarded it for several reasons: first, existing software packages and libraries that operate over $\mathcal{H}$-Matrices utilize similar data representations (of course with an extra of complexity due to the presence of low-rank blocks); second, it is not natural to store an $\mathcal{H}$-Matrix (even if it is a prototype one) repudiating the hierarchy that characterizes it.

Thus, we opted for the second option: refining the adapted BRL $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky algorithms to perform partitions in all the steps where different block sizes are used. In this way, any leaf block that is bigger than the smallest one intervening in the operation is partitioned into blocks whose dimension is equal to the particular smallest leaf size that forms part of the specific operation. This can seem artificial, but there is a positive side effect: finer grain tasks are identified, and a higher concurrency degree can be exploited when using task-based parallel approaches if the sub-operations are annotated as tasks, instead of the original operations defined by the algorithm. Figure 4.7 illustrates the sub-operations and sub-blocks generated as a consequence of the refinement of the adapted BRL $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky factorization algorithms for the particular `TRSM` chosen.

4.5 Parallelization of the algorithms

As exposed earlier, the aim of parallelising this prototype was to evaluate the efficiency of task-based parallel approaches. They usually imply utilizing a runtime that orchestrates the task scheduling by analysing the data dependencies between different tasks. Our proposal decouples the numerical aspects of the linear algebra operation, which are left in the hands of the expert mathematician,

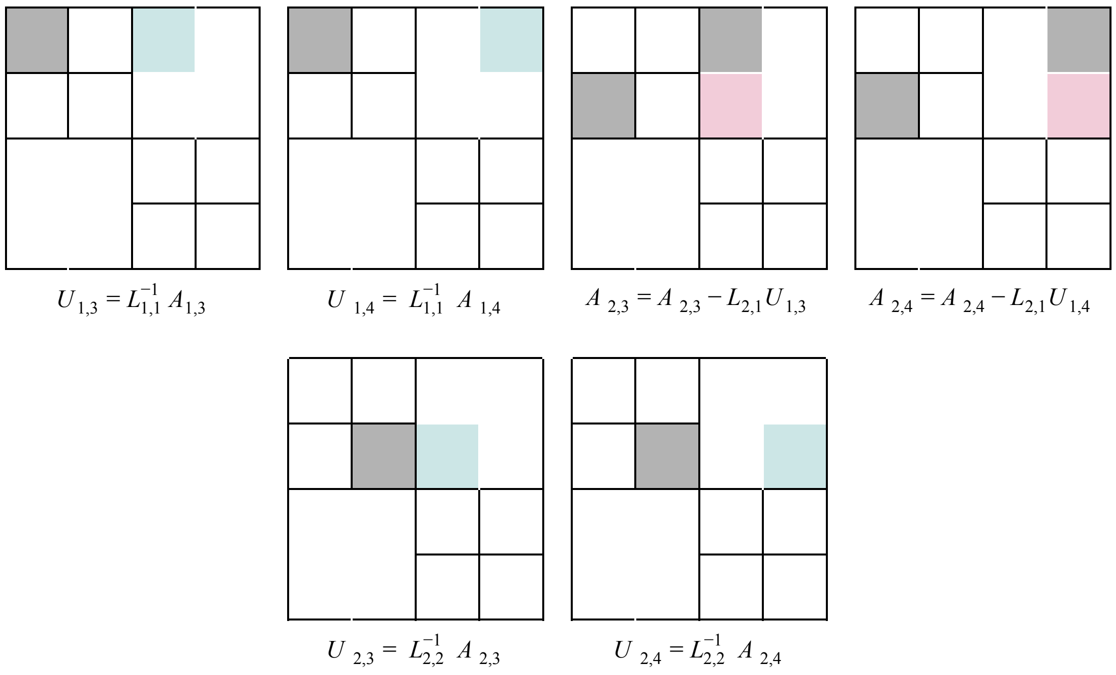
$U_{1,3} = L_{1,1}^{-1} A_{1,3}$
$U_{1,4} = L_{1,1}^{-1} A_{1,4}$
$A_{2,3} = A_{2,3} - L_{2,1} U_{1,3}$
$A_{2,4} = A_{2,4} - L_{2,1} U_{1,4}$
$U_{2,3} = L_{2,2}^{-1} A_{2,3}$
$U_{2,4} = L_{2,2}^{-1} A_{2,4}$

Figure 4.7: Graphical representation of the decomposition into sub-operations of the `TASK_TRSM` that computes $U_{1:2,3:4} = L_{1:2,1:2}^{-1} A_{1:2,3:4}$, employing the sample matrix in Figure 4.1, and the adapted $\mathcal{H}$-LU algorithm represented in Figure 4.4.

physicist or computational scientist, from the difficulties associated with HPC, which are more naturally addressed by computer scientists and engineers.

Both in the $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky, each of the main operations of the algorithm compounds a task, whose dependencies are shown in Table 4.3 for the *k-th* iteration of Algorithms 1 and 2. Moreover, Figure 4.8 shows the Directed Acyclic Graph (DAG) representing the task dependencies for the $\mathcal{H}$-LU (left) and $\mathcal{H}$-Cholesky (right) algorithms.

Algorithm	**Task**	**Data**	**Type of dependencies**
$\mathcal{H}$-LU	`TASK_GETRF`	A_{kk}	input/output (overwritten with L_{kk} and U_{kk})
	`TASK_TRSM` (unit lower factor)	A_{kk}	input (L_{kk}^{-1} entries are read)
		A_{ik}	input/output (overwritten with L_{ik})
	`TASK_TRSM` (upper factor)	A_{kk}	input (U_{kk}^{-1} entries are read)
		A_{kj}	input/output (overwritten with U_{kj})
	`TASK_GEMM`	A_{ik}	input (L_{ik} entries are read)
		A_{kj}	input (U_{kj} entries are read)
		A_{ij}	input/output (overwritten with its updated entries)
$\mathcal{H}$-Cholesky	`TASK_POTRF`	A_{kk}	input/output (overwritten with L_{kk} and L_{kk}^T)
	`TASK_TRSM` (unit lower factor)	A_{kk}	input (L_{kk}^{-T} entries are read)
		A_{ik}	input/output (overwritten with L_{ik})
	`TASK_SYRK`	A_{ik}	input (L_{ik} entries are read)
		A_{kj}	input (L_{jk}^T entries are read)
		A_{ij}	input/output (overwritten with its updated entries)

Table 4.3: Collection of task dependencies for the *k-th* iteration of the $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky algorithms. Input and output dependencies respectively indicate that the specified entries are read and written.

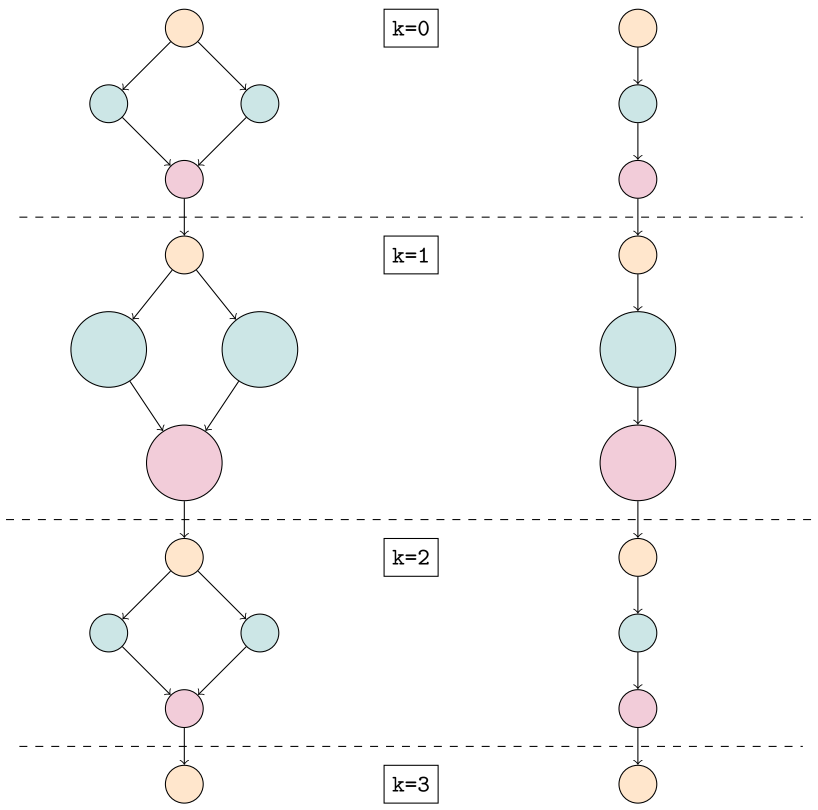

Figure 4.8: DAGs representing the task dependencies associated to the $\mathcal{H}$-LU (left) and $\mathcal{H}$-Cholesky (right) algorithms performed over the sample matrix in Figure 4.3. Colors match those employed in Algorithms 1 and 2, as well as Figures 4.4 and 4.5. Different node sizes illustrate the block size with which each calculation operates. Each k-iteration is separated by dashed lines and labeled.

From the DAGs in Figure 4.8, one could think that there is a low level of concurrency and, consequently, task-based parallel approaches using tools such as OpenMP or OmpSs will not offer any benefit with respect to an execution that simply leverages multithreaded linear algebra packages, such as MKL, neither loop-based parallelism. In fact, in a scenario as simple as that featured by the examples, these strategies would certainly offer lower performance than calling multithreaded kernels. However, when the partitioning does not yield 2×2 sub-blocks, but presents a higher granularity, the number of `TRSM` and `GEMM` operations per level is increased, and so is the number of tasks that can be executed concurrently. Concretely, for a $p \times p$ sub-block partitioning, $2(p-1)$ `TRSM` tasks and $(p-1)^2$ `GEMM` tasks will be performed. Therefore, higher p values imply more abundant opportunities to exploit task parallelism (and, thus, to attain higher performance), and also a greater difference between the efficiency of task-based strategies efficiency and multithreaded kernels based approaches, in favour of the former one. Moreover, with some extra sub-partitionings (as detailed in the Section 4.4) taking place in order to be able to call BLAS/LAPACK routines, an even higher concurrency degree can be exposed.

In OpenMP and OmpSs, the specification of the tasks in the code is simply done by including `pragma` directives which are interpreted by the corresponding scheduler to automatically create the DAG to follow in order to schedule the tasks. These directives encompass all the lines associated with the specific operation that is going to be performed, and are provided with the dependencies specified in Table 4.3. Thus, the tasks annotations are as follows (note that, when using OmpSs, `#pragma oss` is employed instead of `#pragma omp`):

- `TASK_GETRF`:

```
#pragma omp task inout( A[0;M*N] ) {
    void task_getrf( int M, int N, double *A, int LDA, int *IPIV ) {
        int INFO = 0;
        DGETRF( &M, &N, A, &LDA, IPIV, &INFO );
    }
}
```

- `TASK_POTRF`:

```
#pragma omp task inout( A[0;M*N] ) {
    void task_potrf( int M, int N, double *A, int LDA ) {
        char UPLO = 'L'; int INFO = 0;
        DPOTRF( &UPLO, &N, A, &LDA, &INFO );
    }
}
```

- `TASK_TRSM` (the one with unit lower factor is shown; the counterpart with upper triangular factor is analogous):

```
#pragma omp task in( T[0;M*M] ) inout( B[0;M*N] ) {
    void task_utrsm( int M, int N, double *T, int LDT,
                                   double *B, int LDB ) {
        char SIDE = 'L', UPLO = 'L', TRANST = 'N', DIAG = 'U';
        double ALPHA = 1.0;
        DTRSM( &SIDE, &UPLO, &TRANST, &DIAG, &M, &N,
               &ALPHA, T, &LDT, B, &LDB );
    }
}
```

- `TASK_GEMM` (the symmetric case `TASK_SYRK` is analogous):

```
#pragma omp task in( A[0;M*K], B[0;K*N] ), inout( C[0;M*N] ) {
    void task_gemm( int M, int N, int K, double *A, int LDA,
                                         double *B, int LDB,
                                         double *C, int LDC ) {
        char TRANSA = 'N', TRANSB = 'N';
        double ALPHA = -1.0, BETA = 1.0;
        DGEMM( &TRANSA, &TRANSB, &M, &N, &K,
               &ALPHA, A, &LDA, B, &LDB, &BETA, C, &LDC );
    }
}
```

The consideration of the OmpSs programming model, in addition to OpenMP, is because the former one is capable of treating nested data dependency regions, which allows to annotate dependencies taking into account the different block sizes. This capability is useful but not essential for the experiments performed in this chapter, as *representants* of the different blocks (no matter their size) can be used, instead of the whole data region. This is possible thanks to the special scenario in which we are operating now, particularly: 1) there are no low-rank blocks, so there is no block size variation during the computations; 2) the prototype matrix structure is simple (only diagonal blocks are re-partitioned); and 3) a pre-processing to *make room* for blocks data that fill in during the computations, so all the elements are stored in contiguous positions of memory, as if it was a dense matrix. By *representant*, we mean that the first element of the block is taken as the element with respect to which the dependencies are annotated, instead of specifying the whole data region. In contrast with this *idyllic* and simplified situation, in the next chapter the usage of OmpSs with pure $\mathcal{H}$-Matrices will be essential to attain high parallel performance, impossible to reach with current OpenMP capabilities, as will be properly exposed.

4.6 Performance analysis

In this section, we present the tests conducted to elaborate a performance analysis that allows us to discern whether the effort to implement task-based parallel versions of algorithms operating over $\mathcal{H}$-Matrices is worth it. All the experiments were performed using IEEE double precision arithmetic, on a server equipped with two Intel E5-2603v3 sockets, each with a 6-core processor (1.6 GHz), and 32 Gbytes of DDR3 RAM. Our codes were linked with Intel MKL (`composer_xe_2011_sp1`) for the BLAS kernels and the LU/Cholesky factorizations, and OmpSs (version 16.06).

4.6.1 Performance of the MKL kernels

The performance of the presented task-parallel factorizations depends on that of the building blocks, which are computed in our implementation via calls to tuned routines in Intel MKL for this purpose. Note that, as parallelism is extracted by the runtime, we do not link with the multi-threaded implementation of these building blocks in order to avoid *oversubscription* issues.

Figure 4.9 reports the GFLOPS (billions of flops per second) attained by the four building blocks (LU factorization, upper and lower triangular solve, and matrix-matrix multiplication) using a single core of the target platform. We used square operands of dimension t_s (problem dimension in the plot). As could be expected, the highest performance rates are attained by the matrix-matrix multiplication kernel (`DGEMM`). The reason is that this operation makes a quasi-optimal use of the memory subsystem and presents few control dependencies.

Figure 4.9 also reveals that the asymptotic performance for `DGEMM` is around 12.1 GFLOPS. This value is relevant because the `DGEMM` kernel dominates the cost of the factorizations by a large margin. Furthermore, the problem size t_s in the experiment in this figure is related with that of the leaf blocks of the BCT. For example, for an $\mathcal{H}$-Matrix with leaf blocks of dimension $t_s = 1{,}000$, we can expect that an execution of any of the factorizations, using a single core, proceeds at the rate reported for `DGEMM` and that problem size in the plot. The multiplication of the asymptotic rate with the number of cores employed for a task-parallel execution of the factorizations thus offers an upper bound on the highest performance rate that we can observe in a parallel execution.

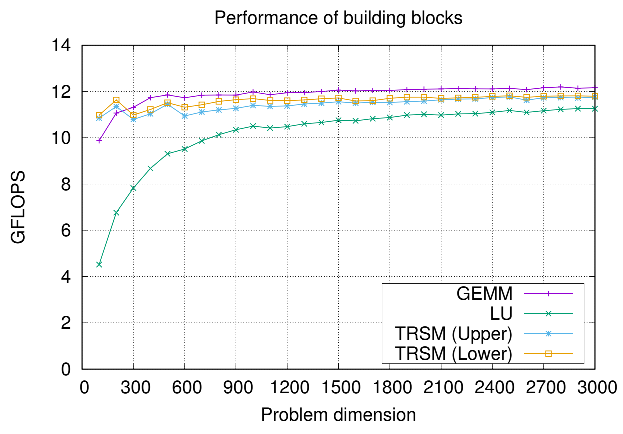

Figure 4.9: Performance of the linear algebra basic building blocks on a single core of the Intel E5-2603v3 server.

It is important to realize that the evaluation of the building blocks was performed using data already stored in the processor cache. For the smallest problems, the GFLOPS rate is much lower if the data has to be fetched from the main memory as part of the execution. However, in the scenario occurring during the factorizations, the operands to a task are the results from a previous task and, therefore, are likely to reside in the higher levels of the memory hierarchy. Thus, the GFLOPS rates in Figure 4.9 are those that we can expect in a practical execution of the factorizations routines.

4.6.2 Performance of our prototype $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky

For the evaluation of the task-parallel prototype $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky, we respectively generated square and symmetric square prototype $\mathcal{H}$-Matrices of dimensions $n = 5{,}000$ and 10,000, whose entries follow a random normal distribution in (0,1). To avoid numerical difficulties, the matrices were enforced to be diagonally dominant. For each case, we varied the number of levels (n_l) and the granularity of the blocks in each level as displayed in Table 4.4. Moreover, the matrices employed in the experiments present different ratios of null blocks (dispersion): 0% (full matrix), 25%, 50%, and 75%. This ratio specifies the number of blocks that are null compared with the total amount of blocks. Note that the actual volume of entries that are zero will depend on the dimensions of the blocks which are randomly selected to be null; thus, a higher ratio of null blocks does not necessarily represent a sparser matrix.

n	n_l	**Block granularity in each level**
5,000	2	5,000, 100
	3	5,000, 500, 100
	4	5,000, 2,500, 1,250, 250
10,000	2	10,000, 500
	3	10,000, 500, 100
	4	10,000, 1,000, 500, 100

Table 4.4: Configurations for the experimental evaluation of the prototype $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky factorizations.

Figures 4.10, 4.11, and 4.12 report the GFLOPS rates attained by the OmpSs version of the $\mathcal{H}$-LU (former one) and $\mathcal{H}$-Cholesky (last two) factorizations routines on the Intel server using 4, 8 and 12 threads/cores, comparing them to other parallel strategies. These performance rates are obtained by calculating the actual flops necessary to factorize each matrix, taking into account that some of its blocks may be null.

Regarding the $\mathcal{H}$-LU, the top of the performance line (upper limit for the y-axis) in all plots is set at 144 GFLOPS, which roughly corresponds to the highest practical performance that we could expect using 12 cores, each delivering about 12 GFLOPS for `DGEMM` (see Figure 4.9). In this factorization, parallelism is extracted 1) from the loop operating on the blocks containing the actual values (represented in the plots under the name "OpenMP"), by including a `#pragma omp parallel for` directive in lines 3, 6, and 9 of the Algorithm 1; and 2) by defining the tasks detailed as in the previous section (named as "OmpSs" in the plots). OpenMP task-based tests instead of OmpSs tasks were also performed, but no remarkable differences were observed. Note that this prototype avoids the problems due to nested dependencies, compared with a pure $\mathcal{H}$-Matrices scenario, where there is an unavoidable need to exploit nested parallelism, as different blocks entries do not lie in contiguous positions of memory (this will be discussed in detail in future chapters).

With respect to the $\mathcal{H}$-Cholesky, a deeper analysis was performed [7] and the associated results are also presented in this dissertation. The parallel variants we evaluated there include:

- MKL Multithread: Parallel algorithm that exploits parallelism inside the BLAS kernels invoked during the execution of the BRL variant of the $\mathcal{H}$-Cholesky factorization.
- OpenMP - Simple: Loop-parallel algorithm that exploits the loop-parallelism present in the BRL variant of the $\mathcal{H}$-Cholesky factorization by including a `#pragma omp parallel for` directive in lines 3 and 6 of Algorithm 2.
- OmpSs and OpenMP - Tasks: Task-parallel algorithms that exploit the parallelism defined by the Task Dependency Graph (TDG) associated with the $\mathcal{H}$-Cholesky factorization.

4.6.3 Conclusions from the experiments

The results of the evaluation in the previous section offer some general conclusions:

- The GFLOPS rates grow with the number of cores in most cases for all the parallelization alternatives and all tested configurations: problem size, number of levels of the hierarchical matrix, and sparsity (dispersion); however, there are notable differences depending on the specific approach.
- In general, the task-parallel versions outperform the loop parallel-based versions and MKL multithread variants. Moreover, the performance differences between the parallelization alternatives tends to become larger, in favor of task-based ones, when the concurrency is reduced. This occurs when the ratio between the factorization cost and number of cores is small, taking into account the rate of nonzeros. A clear example of this can be observed for the $10K$-dimension problem with 2 levels in Figure 4.10, where the performance of the OpenMP version shows a significant drop when the dispersion is increased from 25% to 50%.
- The multithreaded MKL version delivers the worse results (in Figures 4.11 and 4.12). An exception to this is the computation of the $\mathcal{H}$-Cholesky with matrices partitioned into 2 levels, where the OpenMP-Simple solution generally provides smaller GFLOPS rates than MKL.

- When the number of tasks is small, the OpenMP task-parallel implementation offers higher parallel performance than the task-parallel version that relies on OmpSs. For larger matrices, when the number of tasks grows, the difference between the two task-parallel solutions is generally reduced. This is especially visible in Figure 4.10.
- In most cases, the parallel performance is slightly reduced as the amount of null blocks is increased, in comparison to denser configurations.

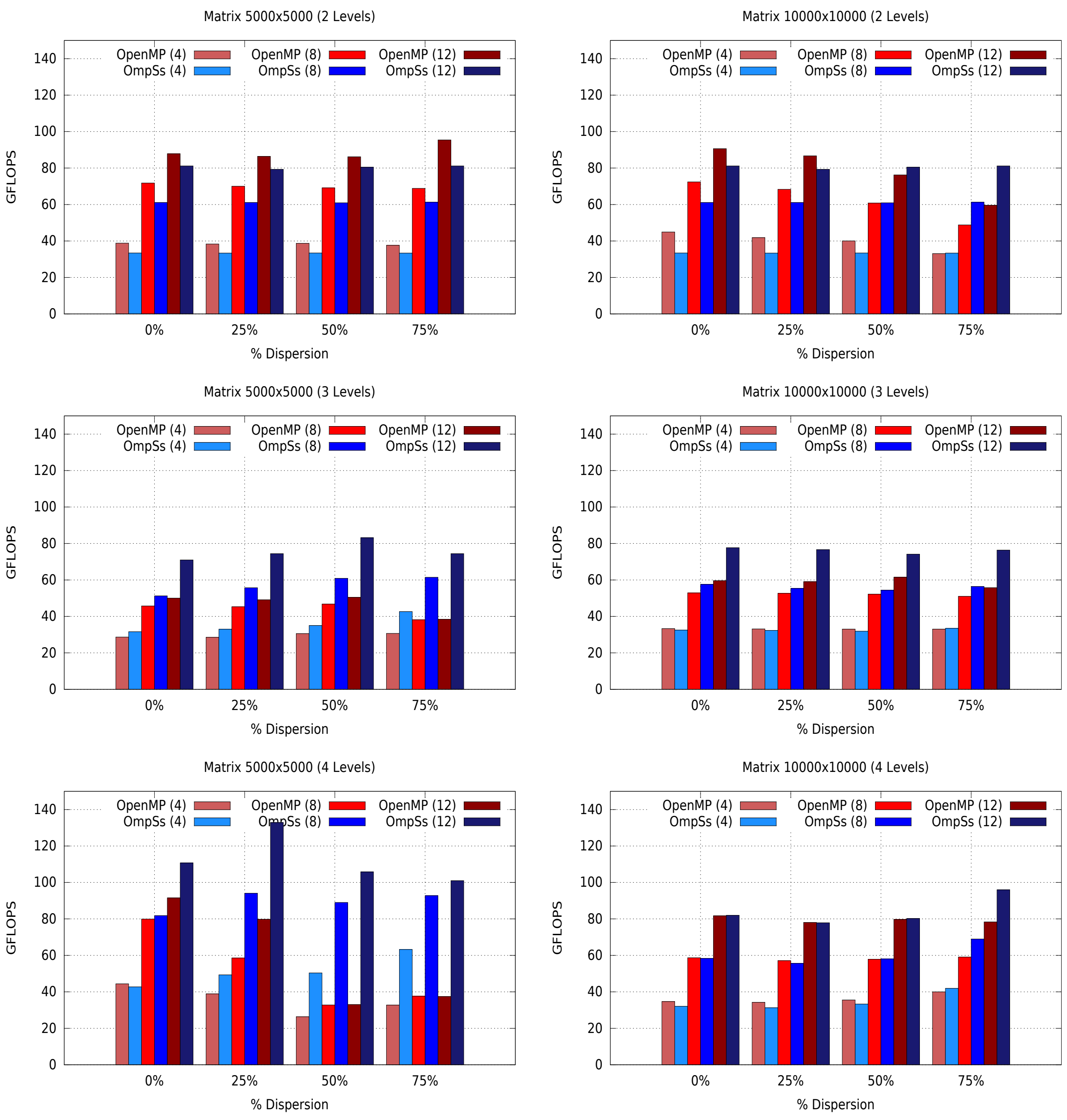

Figure 4.10: Performance of the task-parallel $\mathcal{H}$-LU factorization using OpenMP and OmpSs in the Intel E5-2603v3 server using 4, 8 and 12 threads/cores.

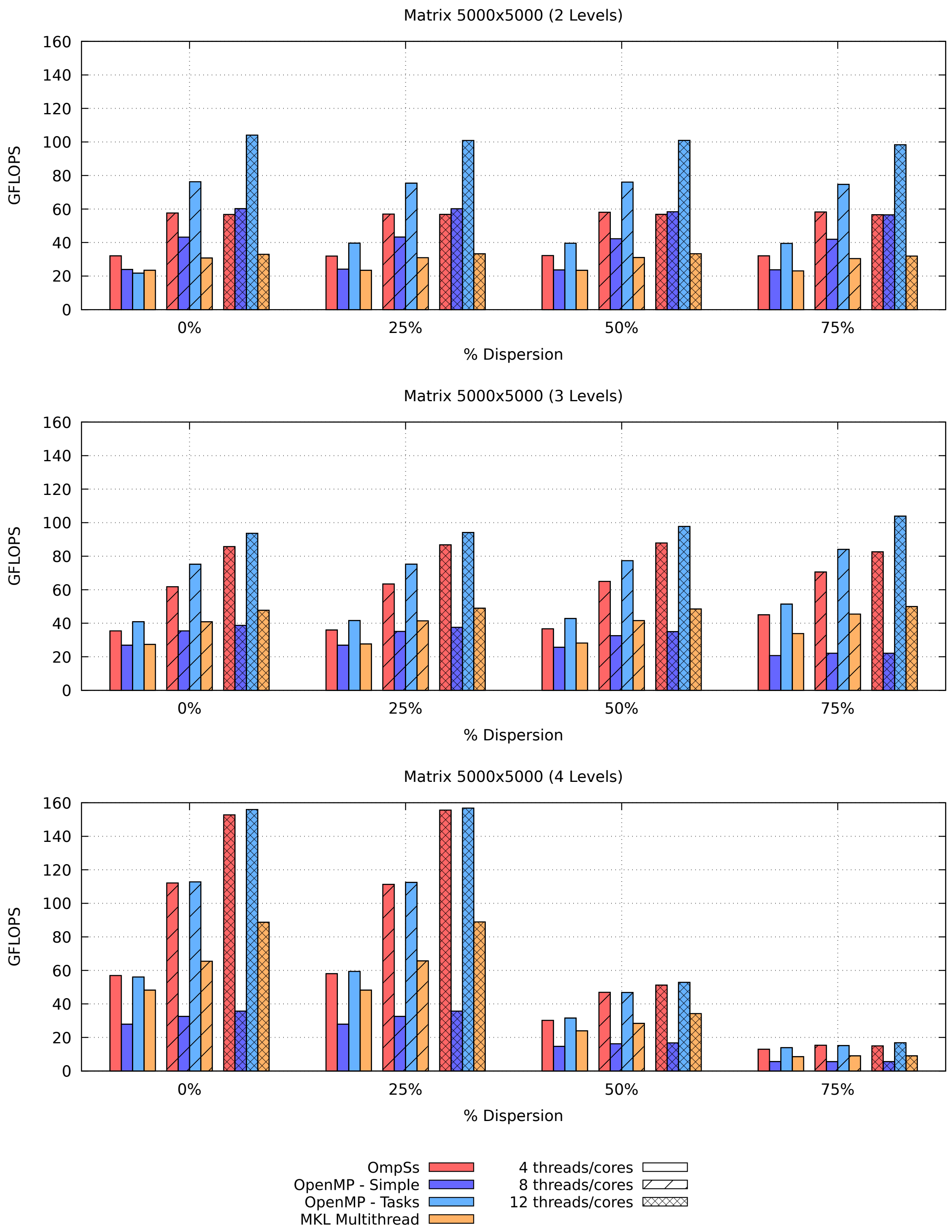

Figure 4.11: Performance of the MKL multithreaded-based version, loop-parallel, and task-parallel $\mathcal{H}$-Cholesky factorization of a matrix of order 5,000 using OpenMP and OmpSs in the Intel E5-2603v3 server using 4, 8 and 12 threads/cores.

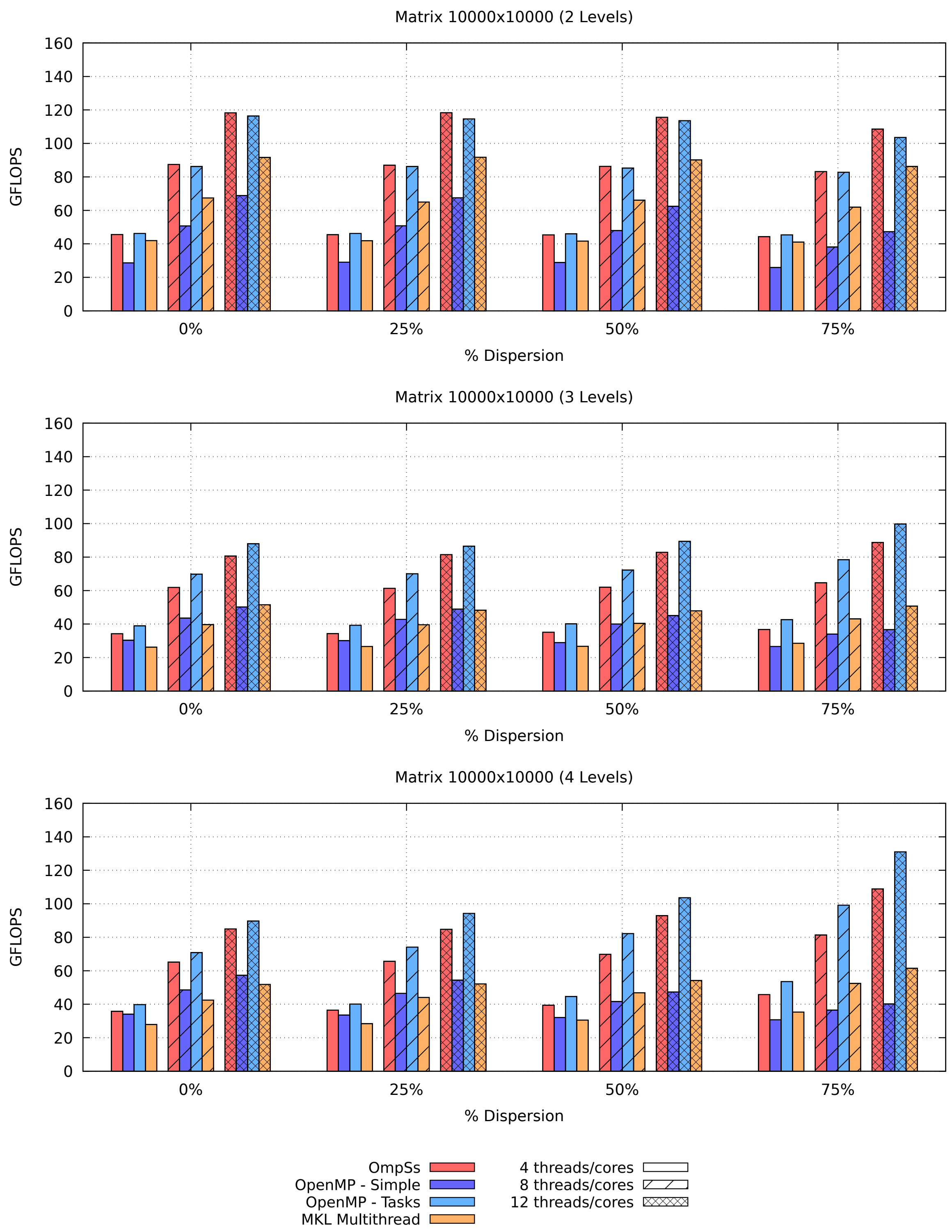

Figure 4.12: Performance of the MKL multithreaded-based version, loop-parallel, and task-parallel $\mathcal{H}$-Cholesky factorization of a matrix of order 10,000 using OpenMP and OmpSs in the Intel E5-2603v3 server using 4, 8 and 12 threads/cores.

4.7 Concluding remarks

We have developed and evaluated several parallel algorithms for the solution of Symmetric Positive Definite (SPD) hierarchical linear systems, via the $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky factorizations on multi-core architectures. In particular, our algorithms investigate the benefits of extracting concurrency from within a multi-threaded implementation of the BLAS; from the loops present in the BRL variant of the Cholesky factorization; or from the tasks that appear when decomposing this operation via a task-parallel implementation with the support of a parallel programming model and runtime as those behind OmpSs and OpenMP. Our results clearly demonstrate that this third option, when combined with the runtimes underlying OmpSs/OpenMP, outperforms the approaches that exploit multi-threaded BLAS and loop-parallelism.

Our design of a task-parallel version of these factorizations using OmpSs had to meet several requirements from three perspectives that guided our implementation, yielding some insights that can be expected to carry over to other linear algebra operations for $\mathcal{H}$-Matrices:

- **Storage:** The data layout has to be efficient, avoiding the storage of zeros (and being prepared to store low-rank blocks in pure $\mathcal{H}$-Matrices scenarios), as well as flexible in order to accommodate variations in the dimensions of the low-rank blocks. These two principles, efficiency and flexibility, led us to select BDL for the problem data.
- **Performance:** For the sake of rapid development and performance portability, the operations on dense blocks (and, to some extent also on low-rank ones) have to rely on routines from LAPACK and BLAS. This moved us to select CMO to store the data inside the blocks. Furthermore, in combination with BDL for the leaf blocks, we have the intuition that, in pure $\mathcal{H}$-Matrices scenarios, certain operations on non-leaf blocks will need to be further decomposed not only to expose a higher concurrency degree, but also to be able to deal with the different leaf block operands sizes.
- **Task-parallelism:** OmpSs provides a powerful tool to exploit task-parallelism.

To conclude, our performance experiments on a server equipped with 12 Intel cores reveal that extracting task-parallelism via OmpSs/OpenMP delivers fair performance and scalability for the task-parallel $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky factorizations.

Chapter 5

Parallelizing the $\mathcal{H}$-LU in H2Lib

Contents of the chapter

5.1 Introduction

As stated in the conclusions of the previous chapter, the results observed in the evaluation of the $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky prototypes exposed that task-based parallelism is suitable for parallel implementations of algorithms involving $\mathcal{H}$-Matrices. However, our intention was never to *reinvent the wheel*, and for this reason we opted for implementing parallel versions of existing packages. Concretely, in this chapter we present the strategy followed to parallelize the $\mathcal{H}$-LU operation included in the H2Lib [54] library. The main objective of the work in this chapter is, thus, to attain fair parallel performance by employing task-based programming models and strategies.

When we developed the implementations that will be shown in this chapter, the H2Lib [54] package offered a limited parallel efficiency, as it was based on basic OpenMP parallel structures (such as sections and loops) but did not exploit tasks. The objective we set ourselves was, consequently, leveraging tasks to attain a higher parallel performance limited only by the data layout and tasks dependencies, but not by the hierarchy levels and nested structures that characterize $\mathcal{H}$-Matrices.

By parallelising the $\mathcal{H}$-LU of H2Lib, the main issues that are addressed are:

- The prototype implementations in Chapter 4 assumed that the blocks of the $\mathcal{H}$-Matrices were either dense or null, and no specialized data structures (neither $\mathcal{H}$-Arithmetic) for low-rank blocks were included. In contrast, H2Lib operates over pure $\mathcal{H}$-Matrices, and thus involves low-rank blocks, low-rank storage, and true $\mathcal{H}$-Arithmetic.
- As a consequence of the previous difference, there is the need to accommodate low-rank data structures that can vary their dimensions at execution time. This is particularly challenging for a runtime-based parallelization because task dependencies are detected via an analysis of the memory addresses of the tasks' operands.
- When utilizing OpenMP/OmpSs tasks as described in the previous chapter, we are forced to operate on fine-grain tasks with operands that were stored in contiguous regions of memory. As it was explained, the practical consequence of this constraint is that it is not possible to exploit the nested task-parallelism intrinsic to the $\mathcal{H}$-LU factorization, both for the prototypes and the true $\mathcal{H}$-LU in H2Lib.

This chapter is structured as follows: we first provide some details about H2Lib, and particularly its $\mathcal{H}$-LU implementation; next we describe the difficulties when parallelizing it, and the new features of the OmpSs-2 programming model that allow us to tackle them; and lastly we show the performance analysis we conducted to evaluate the efficiency of our OmpSs-2 based parallel implementation. At the end of the chapter, we briefly summarize the conclusions extracted from the work described.

5.2 H2Lib

H2Lib [54] (the successor of the HLib package, which is no longer maintained) is an open source library written in `C` language and maintained by a research group which is part of the Department of Mathematics of the Christian-Albrechts-Universität zu Kiel (Germany), leaded by Prof. Steffen Börm. It offers a state-of-the-art implementation of $\mathcal{H}$-Matrix techniques, including sophisticated data structures and support for $\mathcal{H}$-Arithmetic operations.

In the next sections, we will describe the construction of $\mathcal{H}$-Matrices and their storage layout in H2Lib, and explain its sequential $\mathcal{H}$-LU algorithm implementation.

5.2.1 $\mathcal{H}$-Matrices in H2Lib: Construction and Storage layout

The main idea behind $\mathcal{H}$-Matrices is to find a partition of a matrix into blocks which are either small in dimension (and dense) or *admissible* in the sense that they can be stored efficiently using *low-rank data structures* instead of dense ones. As it was described in Chapter 2, utilizing low-rank matrices is a prerequisite for reducing the storage and computational costs down to log-linear functions on the number of elements and flops, respectively.

To be able to properly address the $\mathcal{H}$-Matrix storage layout in H2Lib, it is crucial to understand the steps that form the construction of $\mathcal{H}$-Matrices in that library:

1. The first step consists in "organising" the Degrees of Freedom (DoFs) into sets, which can be handled efficiently via CTs.
2. Secondly, an axis-parallel bounding box $\mathcal{B}_t$ is generated. This contains the union of all extents corresponding to the cluster t.

3. The bounding box is then split into two parts along some geometrical dimension. This yields two disjoint boxes $\mathcal{B}_{t_1}, \mathcal{B}_{t_2}$. From there, H2Lib offers the possibility of classifying each DoF into one of these boxes according to their position in space.

4. Next, both boxes are recursively processed until the number of DoFs located in a box falls below a prescribed constant, which is denoted by *leafsize* (C_{lf}). In order to handle all these boxes efficiently, these clusters are organised into a tree structure expressed by SONS(t) $= \{t_1, t_2\}$.

Algorithm 3 summarizes the construction of the CT, and Listing 5.1 displays the structure (`cluster`) designed to represent a CT in the library.

Algorithm 3 Clustertree construction

Require: Geometric information about the degrees of freedom is stored within an array `dofs` of length `size`.

Ensure: A hierarchical partition of the DoFs is returned via the CT `t`.

```
 1: procedure SETUP_CLUSTERTREE(dofs, size)
 2:     if size > $C_{lf}$ then
 3:         $d \leftarrow$ FIND_SPLITTING_DIMENSION(dofs, size)
 4:         sons $\leftarrow$ 2
 5:         $t \leftarrow$ NEW_CLUSTER(dofs, size, sons)
 6:         {dofs1, dofs2, size1, size2} $\leftarrow$ SORT_DOFS(dofs, size, $d$)
 7:         $t_1 \leftarrow$ SETUP_CLUSTERTREE(dofs1, size1)
 8:         $t_2 \leftarrow$ SETUP_CLUSTERTREE(dofs2, size2)
 9:         SONS(t) $\leftarrow \{t_1, t_2\}$
10:     else
11:         $t \leftarrow$ NEW_LEAF_CLUSTER(dofs, size)
12:     end if
13:     return $t$
14: end procedure
```

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

Listing 5.1: `C` structure for CTs in H2Lib

In Listing 5.1, `size` is the number of elements associated with this cluster and `dofs` is an array specifying the DoFs. The bounding box $\mathcal{B}_t$ for a cluster t is stored within the arrays `bbox_min` and `bbox_max`, respectively. In addition, `son` and `sons` represent the tree structure of the clusters.

Once the necessary CTs are built (one for the rows and a second one for the columns), the setup of the $\mathcal{H}$-Matrix is performed recursively as stated in Algorithm 4, returning a tree-like block structure. Listing 5.2 describes the structure to hold the generated $\mathcal{H}$-Matrices in H2Lib.

Algorithm 4 Blocktree construction

Require: row cluster `t`, column cluster `s`.
Ensure: A blocktree `b` is returned for the pair `(t,s)`.

```
1: procedure SETUP_BLOCKTREE(t, s)
2:     if ADMISSIBLE(t, s) then
3:         b ← NEW_ADMISSIBLE_BLOCK(t, s)
4:     else
5:         if SONS(t) $\neq \emptyset \wedge$ SONS(s) $\neq \emptyset$ then
6:             b ← NEW_PARTITIONED_BLOCK (t, s)
7:             for all t' ∈ SONS(t), s' ∈ SONS(s) do
8:                 b[t'][s'] ← SETUP_BLOCKTREE(t', s')
9:             end for
10:            return b
11:        else
12:            b ← NEW_INADMISSIBLE_BLOCK(t, s)
13:        end if
14:    end if
15:    return b
16: end procedure
```

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

Listing 5.2: `C` structure for $\mathcal{H}$-Matrices in H2Lib.

In Listing 5.2, `rc` and `cc` respectively correspond to the row cluster and column cluster of the current matrix block. In agreement with the three cases occurring in Algorithm 4, the application of that algorithm, at a given level of the recursion, can produce either a low-rank block (i.e., a new admissible block), a new recursive partitioning (via the same algorithm), or a conventional dense (inadmissible) block. Low-rank matrices are stored in the structure `rkmatrix`, whereas dense matrices are stored in `amatrix`. Partitioned matrices are accommodated using the array `son`, whose elements point to the corresponding sons. The structure also provides the amount of sons per row and column, stored in `rsons` and `csons`, respectively. Note that, if a specific block is partitioned, then `rsons`=`rc`→`sons` and `csons`=`cc`→`sons`; however, if it is a leaf block (e.g. because it is admissible), then `rsons`=0 and `csons`=0. Moreover, as we will compute the $\mathcal{H}$-LU for square $\mathcal{H}$-Matrices, then `rsons` = `csons`.

5.2.2 The algorithm for the $\mathcal{H}$-LU in H2Lib

In Chapter 4 we described the recursive algorithm to compute the $\mathcal{H}$-LU. The description there is also applicable to the version of the $\mathcal{H}$-LU implemented in H2Lib. However, in that chapter we were operating with prototype $\mathcal{H}$-Matrices, and that limited the complexity of the algorithm in two ways: 1) on the one hand, as we were not considering low-rank blocks at that moment, we omitted the storage and low-rank/$\mathcal{H}$-Arithmetic details regarding the operations that involve them; and 2) on the other hand, the H2Lib code covers all types oh $\mathcal{H}$-Matrices, and so the algorithm needs to be able explore much more complex hierarchies and, thus, more complex data structures. For those reasons, the implementation of the $\mathcal{H}$-LU in H2Lib is described in detail in this section. This description will be especially convenient to expose the issues that difficult its parallelization.

5.2.2.1 Low-rank storage and low-rank/$\mathcal{H}$-Arithmetic basis

Regarding the storage format, in typical $\mathcal{H}$-matrix implementations, low-rank matrices are represented in the factorized form $X = AB^*$, where A and B have k columns only. Therefore, the rank of X is bounded by k. In practice, k is significantly smaller than the dimensions of the original matrix X. Particularly, the H2Lib package employs the data type in Listing 5.3 to store low-rank matrices. In that structure, `k` represents the maximal rank of the block, and A, B are the left and right factors, respectively.

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

Listing 5.3: `C` structure for low-rank blocks in H2Lib.

With respect to the basic operations for low-rank matrices $X \in \mathbb{R}^{n\times m}$, we find:

- matrix-vector multiplication $y := Xz$, performed using $y := Xz = A(B^*z)$;
- multiplication of X by an arbitrary matrix $Z \in \mathbb{R}^{\ell\times n}$, using $Y := ZX = (ZA)B^*$;
- triangular system solve $LY = X$ or $YU = X$, by applying forward or backward substitution to the k columns of A or the k rows of B, respectively.

Adding two low-rank matrices poses a challenge, since the sum of two matrices $X_1 = A_1B_1^*$ and $X_2 = A_2B_2^*$, of ranks k_1 and k_2, may have a rank $k_1 + k_2$. Fortunately, in typical applications a low-rank approximation can be constructed by computing, e.g., an SVD of

$$X_1 + X_2 = A_1B_1^* + A_2B_2^* = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 & B_2 \end{pmatrix}^*$$

and discarding the smallest singular values. The same approach can be employed to convert an arbitrary matrix into a factorized low-rank matrix.

From the point of view of peak floating-point performance, working with factorized low-rank matrices implies an additional challenge. Concretely, the multiplication of two $n \times n$-matrices requires $2n^3$ operations, i.e., n operations for each coefficient transferred from main memory. In contrast, only $2kn^2$ operations are required if one of the factors is a factorized low-rank matrix, i.e., only k operations for each coefficient. Therefore, when optimizing the $\mathcal{H}$-LU, we have to deal with the fact that the speed of the operations involving low-rank matrices and, in consequence, $\mathcal{H}$-matrices, is generally limited by the memory bandwidth, instead of the floating-point throughput.

5.2.2.2 Procedures that form the $\mathcal{H}$-LU

In order to illustrate and analyse properly the issues that need to be tackled when computing the $\mathcal{H}$-LU, we will employ the two simple $\mathcal{H}$-Matrices shown in Figure 5.1 (on the left, the $\mathcal{H}$-Matrix A utilised in the previous chapter; on the right, an $\mathcal{H}$-Matrix B of the same size but with a different partitioning).

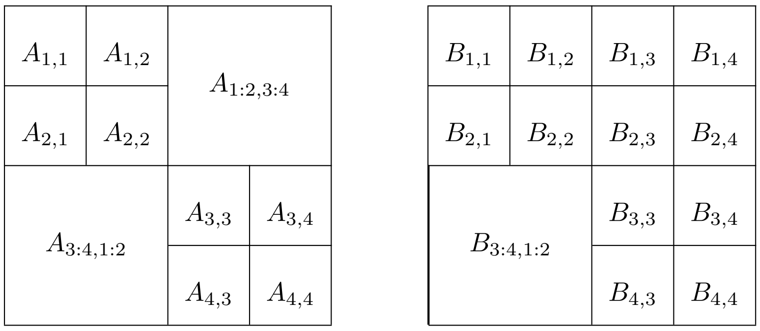

Figure 5.1: Two examples of 8×8 $\mathcal{H}$-Matrices, A and B, with different partitionings.

For the $\mathcal{H}$-Matrix A in Figure5.1, the following sequence of operations computes its $\mathcal{H}$-LU factorization following the Algorithm 1 described in Chapter 4:

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	:	$A_{4,4}$	$=$	$L_{4,4}U_{4,4}$

This is the *natural* extension of the BRL algorithm for the LU factorization, together with its introduction; however, in Chapter 4 we already explained the need of partitioning certain leaf blocks to be able to perform part of the operations. In fact, re-using the mentioned example, we exposed that O2 is actually decomposed into the following six (sub-)operations:

O2.1 :	$U_{1,3}$	$=$	$L_{1,1}^{-1}A_{1,3}$
O2.2 :	$U_{1,4}$	$=$	$L_{1,1}^{-1}A_{1,4}$
O2.3 :	$A_{2,3}$	$:=$	$A_{2,3} - L_{2,1} \cdot U_{1,3}$
O2.4 :	$A_{2,4}$	$:=$	$A_{2,4} - L_{2,1} \cdot U_{1,4}$
O2.5 :	$U_{2,3}$	$=$	$L_{2,2}^{-1}A_{2,3}$
O2.6 :	$U_{2,4}$	$=$	$L_{2,2}^{-1}A_{2,4}$

and the same applies to O3.

Additionally, O4 is decomposed into these eight (sub-)operations:

O4.1 :	$A_{3,3}$	$:=$	$A_{3,3} - L_{3,1} \cdot U_{1,3}$
O4.2 :	$A_{3,3}$	$:=$	$A_{3,3} - L_{3,2} \cdot U_{2,3}$
O4.3 :	$A_{3,4}$	$:=$	$A_{3,4} - L_{3,1} \cdot U_{1,4}$
O4.4 :	$A_{3,4}$	$:=$	$A_{3,4} - L_{3,2} \cdot U_{2,4}$
O4.5 :	$A_{4,3}$	$:=$	$A_{4,3} - L_{4,1} \cdot U_{1,3}$
O4.6 :	$A_{4,3}$	$:=$	$A_{4,3} - L_{4,2} \cdot U_{2,3}$
O4.7 :	$A_{4,4}$	$:=$	$A_{4,4} - L_{4,1} \cdot U_{1,4}$
O4.8 :	$A_{4,4}$	$:=$	$A_{4,4} - L_{4,2} \cdot U_{2,4}$

The analysis of these sequences of (sub-)operations and Figure 5.1 reveals that computing the $\mathcal{H}$-LU of A implies the same operations than doing it for B. The only difference is that the $B_{1:2,3:4}$ block is already partitioned, and thus there is no need to *force* its division into sub-blocks as it is done for that block in A. This variety of partitionings and decomposition of operations implies that the algorithm to compute the $\mathcal{H}$-LU needs to be able to detect the original $\mathcal{H}$-Matrix partitioning and follow its pattern, as well as to perform all the necessary subdivisions to ensure correct data accesses. To sum up, the implementation of the $\mathcal{H}$-LU needs to properly address the recursive nature of the algorithm and also the nested (recursive) block structure imposed by the matrix hierarchy.

Algorithm 5 summarizes all the steps to compute the LU factorization of an $\mathcal{H}$-Matrix, and corresponds to the (graphically) exposition in Chapter 4: an extension for $\mathcal{H}$-Matrices of the BRL algorithm for the LU Factorization. The call to `Compute_HLU` in line 7 is recursive (because the block (i,j) is not a leaf of H); Algorithm 6 is called by `Compute_TRSM_Right` (the algorithm for `Compute_TRSM_Left` is equivalent but solves $U = L^{-1} \cdot A$ instead of $L = A \cdot U^{-1}$); lastly, Algorithm 7 is called by `Compute_GEMM`. There are some considerations to take into account when inspecting these three algorithms:

- If both `rsons` and `csons` of a certain $\mathcal{H}$-Matrix are zero, then it has no sons, which in practice means that the given $\mathcal{H}$-Matrix is a leaf block.
- In $\mathcal{H}$-Matrices, the diagonal blocks are always dense. Consequently, any diagonal block which is also a leaf is stored as a dense matrix (`amatrix`), and the call to `_GETRF` is a call to the LAPACK function that computes its LU decomposition directly, overwriting the initial values with the result. The same applies to the `_TRSM` and `_GEMM` calls, where U and L are always dense if the input $\mathcal{H}$-Matrix is a leaf.

- We have not included all the parameters in the LAPACK calls. *Auxiliary* parameters such as dimension (`M`, `N`), leading dimension (`LDA`), etc. are not indicated in the `_GETRF`, `_TRSM` or `_GEMM` calls to simplify them and ease the reading.
- The SVD operations have also been omitted, but should be performed each time a `_GEMM` call is given a C matrix that is originally low-rank, with A, B or both being also low-rank.
- $\mathcal{H}$-Arithmetic related details, such as giving the rank instead of the full block dimension, have also been omitted.
- The `Subpartition` function is in charge of partitioning the leaf blocks in such a way that they have the same sub-blocks as the smallest block size involved in the operation (see the decomposition of `O2` of A into six (sub-)tasks, or the one for `O4` expressed in the previous paragraphs).
- Lastly, special structure configurations such as those in which two blocks are partitioned following a different pattern, are covered in H2Lib, but they are not addressed in Algorithms 5, 6 and 7 for simplicity.

Algorithm 5 LU Factorization of $\mathcal{H}$-Matrices

```
Require: $H \in \mathcal{H}$-Matrix
Ensure: The LU Decomposition of $H$ (overwriting $H$)
 1: procedure COMPUTE_HLU($H$)
 2:     if $H \to rsons = 0 \wedge H \to csons = 0$ then
 3:         _GETRF($H \to amatrix$)
 4:     else
 5:         for $k = 0 \dots H \to rsons$ do
 6:             $H \to son(k,k)$ = COMPUTE_HLU($H \to son(k,k)$)
 7:             for $i = k+1 \dots H \to rsons$ do
 8:                 $H \to son(i,k)$ = COMPUTE_TRSM_RIGHT($H \to son(k,k)$, $H \to son(i,k)$)
 9:             end for
10:             for $j = k+1 \dots H \to csons$ do
11:                 $H \to son(k,j)$ = COMPUTE_TRSM_LEFT($H \to son(k,k)$, $H \to son(k,j)$)
12:             end for
13:             for $i = k+1 \dots H \to rsons$ do
14:                 for $j = k+1 \dots H \to csons$ do
15:                     $H \to son(i,j)$ = COMPUTE_GEMM($H \to son(i,k)$, $H \to son(k,j)$, $H \to son(i,j)$)
16:                 end for
17:             end for
18:         end for
19:     end if
20:     return $H$
21: end procedure
```

Algorithm 6 TRSM-Right involving $\mathcal{H}$-Matrices

```
Require: $A$, $U \in \mathcal{H}$-Matrix
Ensure: $L = A \cdot U^{-1}$ (overwriting $A$ with $L$)
 1: procedure COMPUTE_TRSM_LEFT($H$)
 2:     if $A \rightarrow rsons = 0 \wedge A \rightarrow csons = 0 \wedge U \rightarrow rsons = 0 \wedge U \rightarrow csons = 0$ then
 3:         if $A \rightarrow rkmatrix$ then
 4:             _TRSM($A \rightarrow rkmatrix$, $U \rightarrow amatrix$)
 5:         else
 6:             _TRSM($A \rightarrow amatrix$, $U \rightarrow amatrix$)
 7:         end if
 8:     else
 9:         for $k = 0 \ldots U \rightarrow rsons$ do
10:             for $i = k \ldots U \rightarrow rsons$ do
11:                 if $A \rightarrow rsons = 0 \wedge A \rightarrow csons = 0$ then
12:                     $A_{ik}$ = SUBPARTITION($A$, $i$, $k$)
13:                     $U_{kk} = U \rightarrow son(k,k)$
14:                 else
15:                     $A_{ik} = A \rightarrow son(i,k)$
16:                     if $U \rightarrow rsons = 0 \wedge U \rightarrow csons = 0$ then
17:                         $U_{kk}$ = SUBPARTITION($U$, $k$, $k$)
18:                     else
19:                         $U_{kk} = U \rightarrow son(k,k)$
20:                     end if
21:                 end if
22:                 $A_{ik}$ = COMPUTE_TRSM_LEFT($A_{ik}$, $U_{kk}$)
23:                 for $j = i + 1 \ldots U \rightarrow csons$ do
24:                     if $A \rightarrow rsons = 0 \wedge A \rightarrow csons = 0$ then
25:                         $A_{ij}$ = SUBPARTITION($A$, $i$, $j$)
26:                         $U_{kj} = U \rightarrow son(k,j)$
27:                     else
28:                         $A_{ij} = A \rightarrow son(i,j)$
29:                         if $U \rightarrow rsons = 0 \wedge U \rightarrow csons = 0$ then
30:                             $U_{kj}$ = SUBPARTITION($U$, $k$, $j$)
31:                         else
32:                             $U_{kj} = U \rightarrow son(k,j)$
33:                         end if
34:                     end if
35:                     $A_{ij}$ = COMPUTE_GEMM($A_{ik}$, $U_{kj}$, $A_{ij}$, $-1$, 1)
36:                 end for
37:             end for
38:         end for
39:     end if
40:     return $A$
41: end procedure
```

Algorithm 7 GEMM involving $\mathcal{H}$-Matrices

```
Require: $A$, $B$, $C \in \mathcal{H}$-Matrix; $\alpha$, $\beta \in \mathbb{R}$
Ensure: $C = \beta C + \alpha AB$
 1: procedure COMPUTE_GEMM($A$, $B$, $C$, $\alpha$, $\beta$)
 2:     if $A \rightarrow rsons = 0 \wedge A \rightarrow csons = 0 \wedge B \rightarrow rsons = 0 \wedge B \rightarrow csons = 0 \wedge C \rightarrow rsons = 0 \wedge C \rightarrow csons = 0$ then
 3:         if $A \rightarrow amatrix$ then
 4:             $Agemm = A \rightarrow amatrix$
 5:         else
 6:             $Agemm = A \rightarrow rkmatrix$
 7:         end if
 8:         if $B \rightarrow amatrix$ then
 9:             $Bgemm = B \rightarrow amatrix$
10:         else
11:             $Bgemm = B \rightarrow rkmatrix$
12:         end if
13:         if $C \rightarrow amatrix$ then
14:             $Cgemm = C \rightarrow amatrix$
15:         else
16:             $Cgemm = C \rightarrow rkmatrix$
17:         end if
18:         _GEMM($\alpha$, $Agemm$, $Bgemm$, $\beta$, $Cgemm$)
19:     else
20:         for $k = 0 \dots B \rightarrow rsons$ do
21:             for $i = k \dots B \rightarrow rsons$ do
22:                 for $j = k \dots B \rightarrow csons$ do
23:                     if $A \rightarrow rsons = 0 \wedge A \rightarrow csons = 0$ then
24:                         $A_{ik}$ = SUBPARTITION($A$, $i$, $k$)
25:                     else
26:                         $A_{ik} = A \rightarrow son(i,k)$
27:                     end if
28:                     if $B \rightarrow rsons = 0 \wedge B \rightarrow csons = 0$ then
29:                         $B_{kj}$ = SUBPARTITION($B$, $k$, $j$)
30:                     else
31:                         $B_{kj} = B \rightarrow son(k,j)$
32:                     end if
33:                     if $C \rightarrow rsons = 0 \wedge C \rightarrow csons = 0$ then
34:                         $C_{ij}$ = SUBPARTITION($C$, $i$, $j$)
35:                     else
36:                         $C_{ij} = B \rightarrow son(i,j)$
37:                     end if
38:                     COMPUTE_GEMM($Agemm$, $Bgemm$, $Cgemm$, $\alpha$, $\beta$, )
39:                 end for
40:             end for
41:         end for
42:     end if
```

```
43:     return $A$
44: end procedure
```

5.3 Parallel $\mathcal{H}$-LU

When we implemented the parallel version of the $\mathcal{H}$-LU included in H2Lib, there were already some alternative libraries that exploited multi-threaded parallel algorithms, but those were mainly based on OpenMP or Intel's TBB (see Section 1.4), and most of the parallelism limitations exposed in the previous chapter held. By parallelising H2Lib, as we describe next, we are able to explore a new task-based multi-threaded parallel approach which allows to: 1) define nested tasks and, 2) execute each of the tasks as soon as the data they depend on is ready. This second capability is crucial to attain a fair parallel performance because, with this technique, the task-based parallelism is not constrained by the hierarchy of nested levels. Thus, thanks to the competitiveness of the sequential algorithms included in H2Lib, and the new task-parallel techniques that we describe in this section, our parallel implementation is able to reach a good parallel efficiency.

Designing a task-based implementation requires a previous analysis of the dependencies among the different operations (which will become tasks) involved in the specific algorithm. We want to note that the only difference between tasks that involve low-rank blocks and those that do not, is the fact that the internal operations are different, but the data dependencies remain the same. For that reason, we do not specify the difference in the task description.

Particularly, for the sample matrices in Figure 5.1 (and, thus, the sequences of operations into which O1 - O5 have been decomposed and analysed in the previous section), Figure 5.2 provides a graphical representation of the dependencies among the operations (tasks), exposing the implicit recursion in each of the operations. The factorization can be initially decomposed into 5 tasks: O1, O2, O3, O4, and O5, with the dependencies among them displayed as shown in the figure. Moreover, O1 and O5, which perform the factorizations of the diagonal blocks of A, are also decomposed into 5 (sub-)tasks each, reproducing the dependency pattern as that of the initial factorization. Besides, O2 and O3 are decomposed each into six (sub-)tasks, and O4 into eight (sub-)tasks, no matter which is the partition of the blocks they involve ($A_{1:2,3:4}$, $B_{1:2,3:4}$, $A_{3:4,1:2}$, $B_{3:4,1:2}$). These partitionings are necessary for those operations involving the values from the diagonal blocks.

The sample matrix is simple yet useful to expose the existence of nested parallelism in the $\mathcal{H}$-LU factorization (and discuss how to tackle it next). However, it only has two levels in the hierarchy, and the dependency graph in Figure 5.2 seems to show that there is little task-parallelism to be exploited, as only these tasks can run in parallel:

- O1.2 with O1.3;
- O2 with O3 (and inside them, O2.1 with O2.2, and O3.1 with O3.2);
- O4.1 with O4.3, O4.5, and O4.7;
- O5.2 with O5.3.

A larger recursive division (this is, a hierarchy composed of more levels) yields a rapid explosion of the degree of task-level parallelism, featuring a richer set of dependencies.

The first conclusion that can be extracted from this dependency analysis is that taking advantage of the intrinsic parallelism of complex $\mathcal{H}$-Matrix structures may require dealing with a considerable

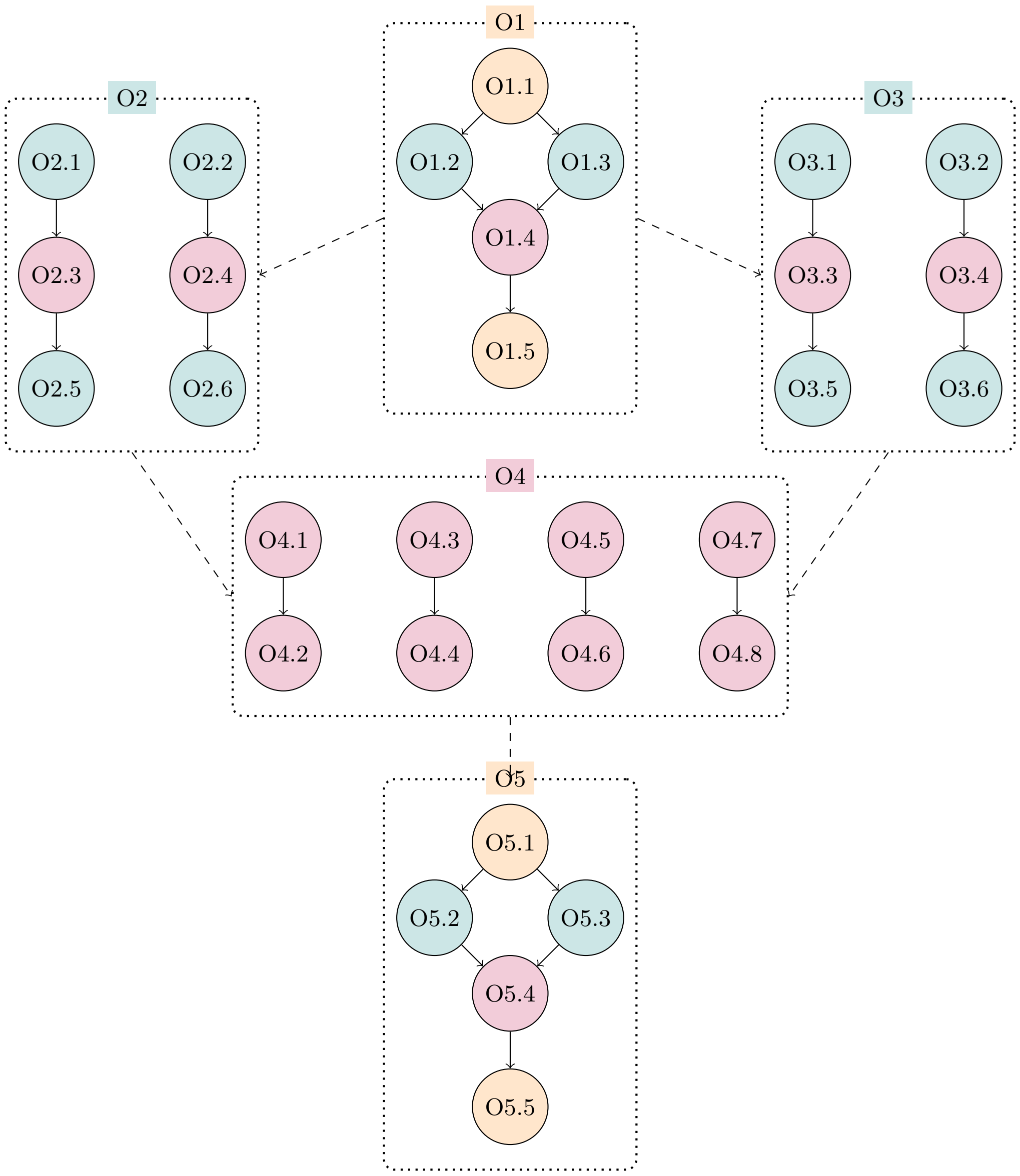

Figure 5.2: DAGs representing the task dependencies associated to the $\mathcal{H}$-LU algorithm performed over the sample matrices in Figure 5.1. The colors match these employed in Algorithms 1, 5, 6, and 7.

amount of nested partitionings. This in turn makes necessary to adapt the "granularity" of the tasks, probably at execution time, in order to maintain performance. In particular, a fine granularity yields more leaf tasks in the structure, but it could occur that it is more efficient to merge groups of them into coarser-grain tasks to attain a better use of the memory cache. Also, a coarser-grain task can be tackled by a group of threads, using for example a multi-threaded version of BLAS. Selecting the optimal option is a difficult scheduling problem.

In addition, it is also natural to conclude that, even though they belong to different "parent" tasks, some (sub-)tasks depend only on other *parent* (sub-)tasks, but not all of them. Looking at a particular example to illustrate this, O2.1 only depends on O1.1; however, due to the recursion of the nested tasks, O2 will not begin its execution until O1 is fully computed, and consequently none of the O2 (sub-)tasks will be initiated until all the five (sub-)tasks of O1 are complete.

The discussed limitations imply difficulties when trying to achieve a fair parallel performance employing the classical task-based approaches. In the next sections, we analyse in more detail the difficulties of the parallel $\mathcal{H}$-LU. Furthermore, we describe the features of the OmpSs-2 programming model that allowed us to overcome these issues and expose a greater amount of concurrency, which reflects more accurately the true data dependencies, and is less dependant on the nested nature of the $\mathcal{H}$-Matrices.

5.3.1 Difficulties observed in our analysis

5.3.1.1 Using representants

The OpenMP and OmpSs runtimes identify task dependencies, at runtime, via the analysis of the memory addresses of the task operands (variables) and their directionality. In order to specify the dependencies between tasks, in dense linear algebra operations we often use a "representant" for each task operand, which is then passed to the runtime system in order to detect these dependencies [20]. This representant is the memory address of the matrix block computed by the corresponding operation; concretely, the top-left entry of the output matrix block. We next discuss the problem with this approach in the context of $\mathcal{H}$-Matrices.

Let us consider, for example, the dependency O1.1→O1.2, between the LU factorization

$$\mathsf{O1.1}: \quad A_{1,1} \;=\; L_{1,1}U_{1,1},$$

and the triangular system solve

$$\mathsf{O1.2}: \quad U_{1,2} \;:=\; L_{1,1}^{-1}A_{1,2};$$

and the dependency O1→O2, between the LU factorization

$$\mathsf{O1}: \quad A_{1:2,1:2} \;=\; L_{1:2,1:2}U_{1:2,1:2},$$

and the triangular system solve

$$\mathsf{O2}: \quad U_{1:2,3:4} \;:=\; L_{1:2,1:2}^{-1}A_{1:2,3:4}.$$

For simplicity, let us assume that all the blocks involved in these operations are dense. (The analysis of the dependencies for low-rank blocks is analogous.) The problem with the use of representants is that it is not possible to distinguish a dependency with the input $A_{1:2,1:2}$ from one that has its origin in the input $A_{1,1}$. In particular, since both $A_{1:2,1:2}$ and $A_{1,1}$ share the same representant, with this technique it is not possible to know whether O1.2 and O2 depend either on O1.1 or O1. As a consequence, when operating with $\mathcal{H}$-Matrices we need to be able to annotate tasks with respect to data regions, and not only employing "representants".

5.3.1.2 Leveraging regions instead of representants

OpenMP and OmpSs offer enough flexibility to specify the shapes/dimensions of the input/output operands passed to a task as *regions*, which can then be used to detect dependencies between the tasks. In principle, it might seem that this mechanism could be leveraged to avoid the ambiguity due to the use of representants. However, the following discussion illustrates that this is still insufficient for H2Lib.

To expose the problem, consider again the dependencies O1.1→O1.2 and O1→O2 where, for simplicity, we still assume that all blocks involved in these operations are dense. To tackle this case, it might seem that we could simply specify the dimensions of the operands. For example, in OmpSs, the lower triangular system solves O1.2 and O2 could be annotated as

```
#pragma omp task in( L[0;M*M] ), inout( B[0;M*P] )
void task_trsm_left( int M, int P, double *L, int LDL,
                                   double *B, int LDB )
```

where `L` corresponds to the base memory address of the M×M lower triangular factor, and `B` is the base memory address of the M×P right-hand side. The data indicated in the square brackets means `X[distance of the first element of the region of X to consider, with respect to its the base address; number of elements that form the region to consider]`.

The problem with this solution is that, in H2Lib, the entries of a block which is further partitioned into sub-blocks (as is the case for $A_{1:2,1:2}$) are not stored contiguously in memory. Therefore, the use of a region to specify the memory address of the contents of such block is useless.

Our workaround to this problem in the previous chapter was to divide all the necessary blocks into the smallest block size in the matrix. Unfortunately, this solution implies the need to explicitly decompose all tasks in the $\mathcal{H}$-LU factorization to operate with blocks of the "base" granularity, so that a region only spans data which is contiguous in memory. The practical consequence is that, with that approach, it was not truly possible to exploit nested task parallelism. Furthermore, in the case of small leaf blocks, the overhead introduced by the dependency-detection mechanism can be considerable, reducing the performance of the solution.

5.3.1.3 Dealing with non-contiguous regions

This issue is difficult to address, as it is rooted on the hierarchical nature of the problem and the use of $\mathcal{H}$-Arithmetic, which derives in the need to embody a data structure that can vary at runtime. Thus it becomes necessary to maintain a tree-like structure of the matrix contents, where only the leaf blocks (either dense or low-rank) store their data contiguously in memory. As a result, we cannot leverage this data structure to specify dependencies between tasks involving non-leaf blocks.

Our solution to this problem is application-specific (but can be leveraged in scenarios involving dynamic and/or complex data structures [4]) and consists of an auxiliary skeleton data structure that reflects the block structure of the $\mathcal{H}$-Matrix. In particular, this data structure can be realized using an array with one representative per leaf (i.e., non-partitioned) block in the original matrix, where the representatives that pertain to the same block appear in contiguous positions of memory. For the particular simple example $\mathcal{H}$-Matrix A in Figure 5.1 this means that, in order to detect dependencies, we use an additional array of representants to ensure the desired specific order:

$A_{1,1}$	$A_{2,1}$	$A_{1,2}$	$A_{2,2}$	$A_{3:4,1}$	$A_{1:2,3}$	$A_{3,3}$	$A_{4,3}$	$A_{3,4}$	$A_{4,4}$

Operating in this manner, we decouple the mechanism to detect the dependencies (based on the previous array) from the actual layout of the data in memory, which can vary during the execution.

With this solution, the ambiguity between O1.1 and O1 when dealing with the dependencies O1.1→O1.2 and O1→O2 is easily tackled. Concretely, although both operands share the same base address in memory (that of $A_{1,1}$ in the skeleton array), the region for O1.1 comprises a single representant while that of O1 comprises four representants in the skeleton array.

Following this approach, there is no longer the need to partition all the blocks in the $\mathcal{H}$-Matrix into the smallest block size. However, as we have described in previous sections, due to the need of storing the data in contiguous positions of memory when calling the LAPACK functions, the need to partition certain leaf blocks into the smallest block size intervening in a specific operation still applies. Consequently, we will not only employ one representant per leaf block, but also include extra representatives for the leaf blocks. The reason is that when they involve data coming from diagonal blocks in a certain operation, they will be inevitable partitioned. Thus, for the examples in Figure 5.1, we will actually use the following representants (note that BDL format is employed to determine the way in which the representatives are stored):

$A_{1,1}$	$A_{2,1}$	$A_{1,2}$	$A_{2,2}$	$A_{3,1}$	$A_{4,1}$	$A_{3,2}$	$A_{4,2}$	$A_{1,3}$	$A_{2,3}$	$A_{1,4}$	$A_{2,4}$	$A_{3,3}$	$A_{4,3}$	$A_{3,4}$	$A_{4,4}$

We emphasize that these representants are stored contiguously in memory and this skeleton data structure does not vary during the execution (in contrast with the structure storing the actual data). Therefore, it can be built before the operations commence, and the cost of assembling it can be amortized over enough computation thanks to the higher degree of concurrency it exposes.

5.3.2 OmpSs2-2 new features: weak dependencies and early release

The tasking model of OpenMP 4.5 supports both nesting and the definition of dependencies between sibling tasks. Many operations with $\mathcal{H}$-Matrices are recursive, so the natural strategy to parallelize them is to leverage task nesting. However, this top-down approach has some drawbacks since combining nesting with dependencies usually requires additional measures to enforce the correct coordination of dependencies across nesting levels. For instance, most non-leaf tasks need to include a `taskwait` construct at the end of their code. While this enforces the correct order of execution, as a side effect, it also constrains both the generation and discovery of task parallelism. In this paper we leverage the enhanced tasking model recently implemented in OmpSs-2 [101] to exploit both nesting and fine-grained data-flow parallelism.

The OmpSs-2 tasking model introduces two major features: *weak dependencies* and *early release* of dependencies. The dependencies due to task operands annotated as weak are ignored by the runtime when determining whether a task is ready to be executed. This is possible because the operands marked as weak can only be read or written by child tasks. Using weak dependencies, subtasks can thus be instantiated earlier and in parallel. The early release of dependencies allows a fine-grained release of dependencies to sibling tasks. Concretely, with this advanced release, when a task ends, it immediately releases the dependencies that are not currently used by any of its child tasks. Furthermore, as soon as the child tasks finish, they release the dependencies that are not currently used by any of their sibling tasks.

To further clarify this, we remark that the correct use of task nesting and dependencies has to fulfil the following rule to avoid data-races between tasks that are second (or above)-degree relative: the dependency set of a child task has to be a subset of the dependency set of its parent task.

Only those dependencies declared on data that is not available in the scope of the parent task, such as data dynamically allocated when the body of the parent task is executed, are excluded from this rule. Although this rule guarantees a correct execution, it usually introduces artificial coarse-grained dependencies between sibling tasks, which are only required to enforce the proper synchronization of their sibling tasks.

To address the previous issue, we can leverage weak dependencies because this type of dependencies are just ignored by the runtime when determining whether a task is ready to be executed. This is possible because operands marked as weak can only be read or written by child tasks. Using weak dependencies, more tasks can be thus instantiated earlier and in parallel, and we can avoid the insertion of a `taskwait` construct, at the end of each parent task, to enforce a barrier which synchronizes all the child tasks before releasing all the dependencies. This means that, in Algorithm 5 we would need to include a `taskwait` after the `endfor` in line 17 (this is, at the end of each iteration of the k-loop); in Algorithm 6 after the line 38 (again, at the end of each iteration of the k-loop, and analogously for the `TRSM-Left`); and in Algorithm 7 after the line 41 (once more, at the end of each iteration of the k-loop). One can easily see that this substantially limits the concurrency allowed in the parallel execution, as the k-loop is the one in charge of traversing the hierarchy levels.

By combining these two contributions, dependencies can cross the boundaries initially set up by the nesting contexts. The resulting behavior is equivalent to performing all the dependency analysis in a single domain. Achieving a similar effect in OpenMP/OmpSs eliminated the possibility of nesting. In addition, that approach also reduced the programmability and restricted the instantiation of tasks to a single generator. In constrast, the dependency model of OmpSs-2 can extract the same amount of task parallelism, without impairing programmability and without the loss of the parallel generation of work that is possible through nesting.

In order to illustrate the implications of these two advanced features of OmpSs-2 on the $\mathcal{H}$-LU factorization, let us consider the partitioning of the sample matrices in Figure 5.1 and the associated sequences of operations to perform the $\mathcal{H}$-LU which have been already described. In the application of nested parallelism to that scenarios, we assume that O1, O2, O3, O4, and O5 are each annotated as a (coarse-grain) task, and they respectively produce the (sub-)operations in O1.1–O1.5, O2.1–O2.6, O3.1–O3.6, O4.1–O4.8, O5.1–O5.5, each annotated as a (fine-grain) task.

A rapid analysis reveals that, for example, the coarse-grain dependency O1→O2 boils down (among others) to the finer-grain cases O1.1→{O2.1, O2.2}, as the former operation (LU factorization)

$$\text{O1.1}: \quad A_{1,1} \quad := \quad L_{1,1}U_{1,1}$$

yields the unit lower triangular factor $L_{1,1}$ required by the latter two operations (triangular solves) O2.1, O2.2.

The problem with OmpSs and OpenMP 4.5 is that ensuring a correct result requires the introduction of a `taskwait` at the end of the code for O1. In contrast, the support for weak dependencies and early release in OmpSs-2 implies that (provided the operand $L_{1:2,1:2}$ for O2 is annotated as weak), the boundaries between the coarse-grain tasks O1 and O2 can be crossed and the execution of O2.1 and O2.2 can commence as soon as O1.1 is computed. In order to attain this effect, in OmpSs-2 we should annotate O2 as a task with weak operands (via the corresponding representants):

```
#pragma oss task weakin( RepL[0;S] ), weakinout( RepB[0;S] )
void task_trsm_left( int M, int P, double *L, int LDL,
                                   double *B, int LDB )
```

while O2.1, O2.2 are both specified as tasks with strong operands:

```
#pragma oss task in( L[0;M*M] ), inout( B[0;M*P] )
void task_trsm_left( int M, int P, double *L, int LDL,
                                  double *B, int LDB )
```

In general, if we go back to the Algorithms 5, 6, and 7, all the recursive calls are annotated as weak tasks, and the tasks that actually update the data values (that is, the ones that call BLAS/LAPACK functions) as regular (strong) tasks. This is a summary of the tasks annotations:

- For Algorithm 5 ($\mathcal{H}$-LU factorization):
  - Weak: `Compute_HLU` (line 6), `Compute_TRSM_Right` (line 8), `Compute_TRSM_Left` (line 11), and `Compute_Gemm` (line 15).
  - Strong: `_GETRF` (line 3).
- For Algorithm 6 (`TRSM-Right`) and analogously for the `TRSM-Left`:
  - Weak: `Compute_TRSM_Left` (line 22), and `Compute_Gemm` (line 35).
  - Strong: `_TRSM` (lines 4 and 6).
- For Algorithm 7 (`GEMM`):
  - Weak: `Compute_Gemm` (line 38).
  - Strong: `_GEMM` (line 18).

This simple example illustrates that the use of weak dependencies and early release can unleash a higher degree of task-parallelism during the execution of the $\mathcal{H}$-LU factorization. Figure 5.3 reflects the new dependency graph displaying the weak dependencies with dashed arrows, and regular (strong) dependencies, which are the ones that actually orchestrate the parallelism, using solid arrows. We have highlighted in colored arrows the new dependencies that can be exposed thanks to the OmpSs-2 weak dependencies and early release of tasks; particularly, in blue the ones from O1 to O2 and O3 (sub-)tasks; in green the ones from O2 to O4 (sub-)tasks; in orange the ones from O3 to O4 (sub-)tasks; and in red the ones from O4 to O5 (sub-)tasks. As an example, after the execution of O1.1, now not only O1.2 and O1.3, but also O2.1, O2.2, O3.1 and O3.2 can commence their execution, without these last four needing to wait until O1.5 is completed, as it was the case with OpenMP/OmpSs.

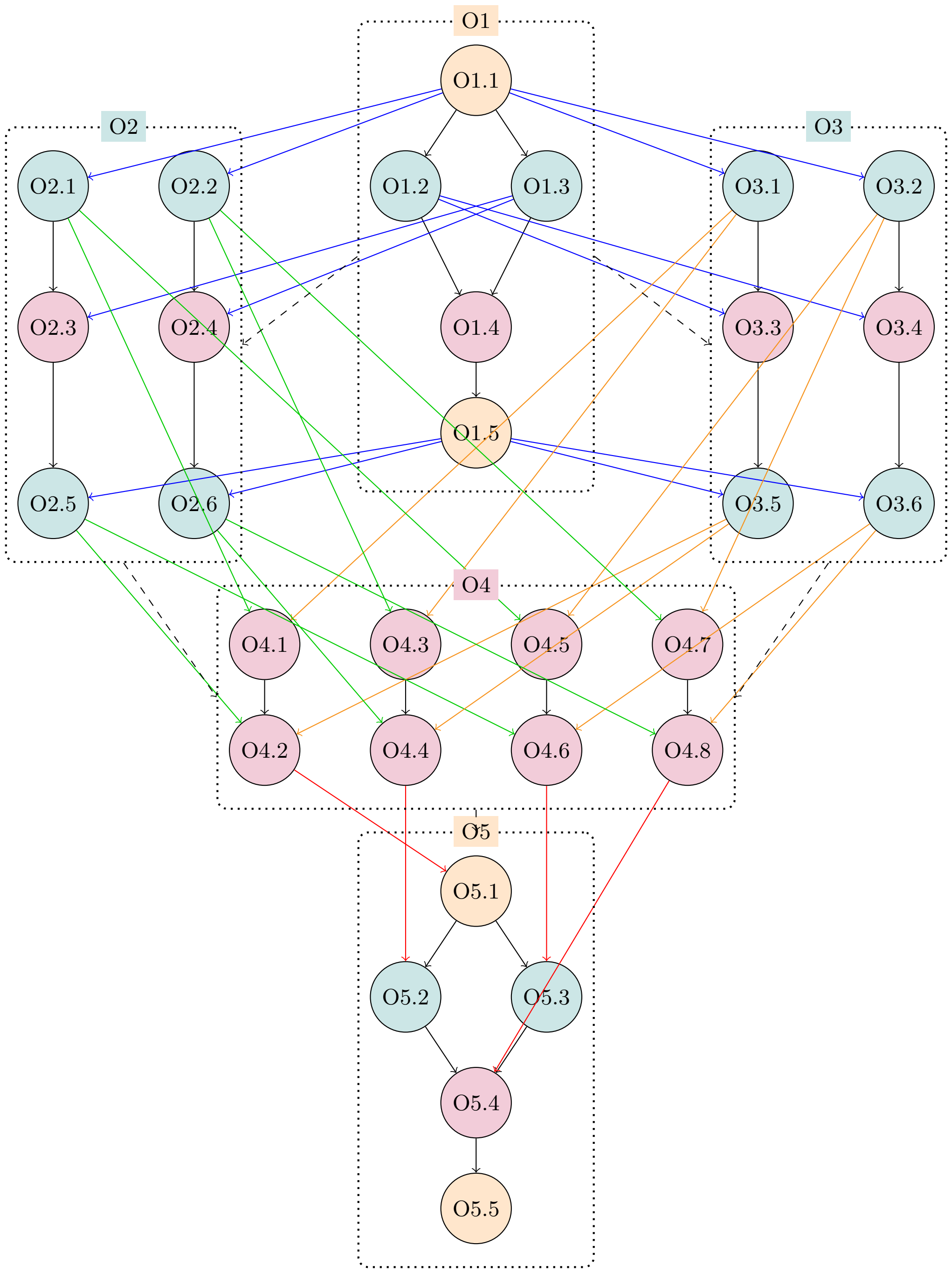

Figure 5.3: DAG for the $\mathcal{H}$-LU task dependencies over the sample matrices in Figure 5.1, when using OmpSs-2. Dashed arrows symbolise weak dependencies, while solid arrows represent regular (strong) dependencies. Colored full arrows show the dependencies exposed thanks to OmpSs-2 new features.

5.4 Performance analysis

In this section we first describe the problem setup and target architecture employed in our experiments. Next, we analyze the efficiency of the parallel implementations of the code for the $\mathcal{H}$-LU factorization in H2Lib.

5.4.1 Mathematical problems

The usage of $\mathcal{H}$-matrices often appears in the context of BEM [72]. The reason is that the discretization of boundary integral equations often yields matrices that are densely populated and have to be stored efficiently, where $\mathcal{H}$-Matrices come in handy. There is also the need to construct efficient preconditioners for this type of equations, which can be carried out in $\mathcal{H}$-Arithmetic. In particular, in the experiments in this section we consider integral equations of the form

$$\int_\Gamma g(x,y)\, u(y)\, dy = f(x), \qquad \text{for almost all } x \in \Omega,$$

where Ω can be some d-dimensional bounded domain for $d \in \{1, 2,3\}$. By choosing suitable test-and-trial spaces $\mathcal{U}_h$ and $\mathcal{V}_h$, equipped with some bases $(\varphi_i)\,, i \in \mathcal{I}$, and $(\psi_j)\,, j \in \mathcal{J}$, we can apply a Galerkin discretization and obtain a variational formulation of the type

$$\int_\Omega v_h(x) \int_\Gamma g(x,y)\, u_h(y)\, dy\, dx = \int_\Omega v_h(x)\, f(x)\, dx\,, \qquad \text{for all } v_h \in \mathcal{V}_h.$$

Employing finite element basis functions for these spaces, we directly obtain a system of linear equations

$$Gu = f,$$

where all the entries of the matrix

$$g_{ij} = \int_\Omega \varphi_i(x) \int_\Gamma g(x,y)\, \psi_j(y)\, dy\, dx\,, \qquad \text{for all } i \in \mathcal{I}, j \in \mathcal{J},$$

are non-zero.

In particular, we consider the Laplace equation in $d \in \{1,2,3\}$ dimensions. In these cases, the underlying kernel functions are

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

For the construction of low-rank blocks in our experiments, we choose the analytical method of tensor-interpolation [57], which is applicable in all dimensions. For the sake of lighter storage requirements and faster setup times of the $\mathcal{H}$-LU, we further re-compress all low-rank blocks using a fast SVD [51].

5.4.2 Setup

All the experiments in this section were performed using IEEE 754 double-precision arithmetic, on a single node of the MareNostrum 4 system at Barcelona Supercomputing Center [88]. The node contains two Intel Xeon Platinum 8160 sockets, with 24 cores per socket, and 96 Gbytes per of DDR4 RAM. In Turbo frequency mode (3.7 GHz), the theoretical peak performance for a single core is 59.2 GFLOPS (billions of flops per second) when using AVX2 instructions. This rate is reduced to 33.6 GFLOPS when using a single core running at the base frequency (2.1 GHz). At this point we note that the aggregated (theoretical) peak performance of this machine is a linear function of the operation frequency which, in turn, depends on the specific type of vector instructions that are executed (AVX, AVX2, AVX-512) and the number of active cores [52].

In the experiments we employed `gcc` 4.8.5, Intel MKL 2017.4 (with AVX2 instructions enabled), and OmpSs-2 (`mcxx` 2.1.0).

5.4.3 Matrix-matrix multiplication

Our first experiment is designed to assess the performance of the implementation of the matrix-matrix multiplication routine (`_GEMM`) in Intel MKL. This is relevant because it offers an upper bound of the actual performance that can be obtained for the LU factorization of a hierarchical matrix. This bound will be tight in case most of the blocks involved in the decomposition are dense and the fragmentation of the blocks implicit to the matrix hierarchy is not too fine-grained.

Figure 5.4 reports the GFLOPS *per core* attained by Intel's `_GEMM` routine using 1, 8, 16,. . . , 48 cores (of a single node with two sockets) and square operands all of the same dimension b. (Note that the limit of the y-axis in this plot and all subsequent ones is fixed to 60, which basically corresponds to the theoretical peak performance with 1 core.) This experiment reveals two important aspects. First, the execution of the sequential instance of `_GEMM` delivers 57.0 GFLOPS for a problem of order $b = 150$, and 58.9 GFLOPS for the largest problem dimension, $b = 1000$. These values represent 96.2% an 99.4% of the peak rate, respectively (when using AVX2 instructions). Thus, even for problems that are rather small, it is already possible to attain a large fraction of the peak performance when using a single thread. Second, as the number of threads/cores grows, the multi-threaded instance of `_GEMM` requires considerably larger problems to attain a relevant fraction of the theoretical peak. (As argued earlier, the peak rate of this processor is "variable" because it depends on the operation frequency and this parameter is constrained by the number of active cores [52].)

5.4.4 Results

In this section, we first provide some references about the performance of the parallel $\mathcal{H}$-LU implementation when using three *simpler* parallel strategies: 1) using the multithreaded version of MKL (that is, extracting fine-grain parallelism from within BLAS kernels); 2) exploiting loop parallelism targeting a coarser-grain layer using OpenMP; and 3) annotating tasks using OmpSs (without leveraging OmpSs-2 described features) and, hence, using `taskwait` directives. Secondly, we provide a strong/weak scalability analysis to illustrate the performance gain when using OmpSs-2 including weak dependencies and early release, in contrast to the same task-based parallel implementation without leveraging the mentioned features (and consequently, incorporating the needed `taskwait`). Lastly, we show the efficiency of our OmpSs-2 based parallel implementation of the $\mathcal{H}$-LU in larger test cases, in the execution of 1D, 2D, and 3D cases arising from BEM.

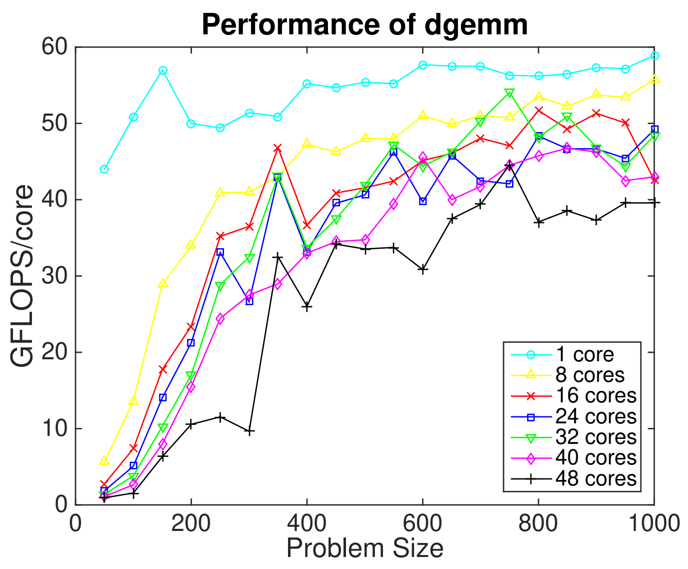

Figure 5.4: Performance of the matrix-matrix multiplication routine in Intel MKL.

5.4.4.1 Basic parallel solutions

The following experiment exposes the drawback of a parallelization that simply relies on a multi-threaded instance of the BLAS, providing initial evidence that a runtime-based approach can offer higher performance. In order to do so, we compare three different parallelization strategies applied to the $\mathcal{H}$-LU factorization:

- MKL extracts fine-grain loop-parallelism from within the BLAS kernels only. As argued in the introduction of this paper, this approach is rather appealing as it only requires a low programming effort. In particular, provided the sequential routine for the $\mathcal{H}$-LU factorization already casts most of its operations in terms of BLAS, the code can be executed in parallel by simply linking in a multi-threaded instance of this library such as that in Intel MKL. The downside of this approach is that it constrains the parallelism that can be leveraged to that inside individual kernels, which may be insufficient if the number of cores is large.
- OpenMP aims to exploit loop-parallelism (like MKL) but targets a coarser-grain layer, by applying the parallelization to the loops present in the $\mathcal{H}$-LU routine. To clarify this, consider for example a single-level hierarchical matrix that is decomposed into 8×8 blocks. After the factorization of the leading block of the matrix, this approach will compute in parallel the remaining 7+7 triangular system solves in the same column+row of the matrix; and next update the 7×7 blocks of the trailing submatrix in parallel. In summary, instead of extracting the parallelism from within the individual BLAS kernels, this approach targets the parallelism existing between the independent BLAS kernels (tasks) comprised by a loop.
- OmpSs discovers tasks dynamically and takes into account the dependencies among them to schedule their execution when appropriate. (This version does not include the advanced features supported by OmpSs-2.)

To simplify the following analysis, we will employ a hierarchical matrix with a 2×2 recursive structure defined on the diagonal blocks. Concretely, starting with a hierarchical matrix of order

n, we define a 2×2 partitioning, which is recursively applied to the inadmissible blocks until a minimum leaf size is reached; see Figure 5.5. The admissibility condition we have employed in this case is:

$$\max\{\operatorname{diam}(\mathcal{B}_t), \operatorname{diam}(\mathcal{B}_s)\} \leq \eta \operatorname{dist}(\mathcal{B}_t,\mathcal{B}_s) \textit{ where } \eta \in \mathbb{R}_{>0}.$$

This type of data structure appears, for example, in BEM with $d = 1$, as those described in subsection 5.4.1. For simplicity, we will also consider dense blocks only. With these considerations, the cost of the LU factorization of a hierarchical matrix of order n is (approximately) the standard $2n^3/3$ flops.

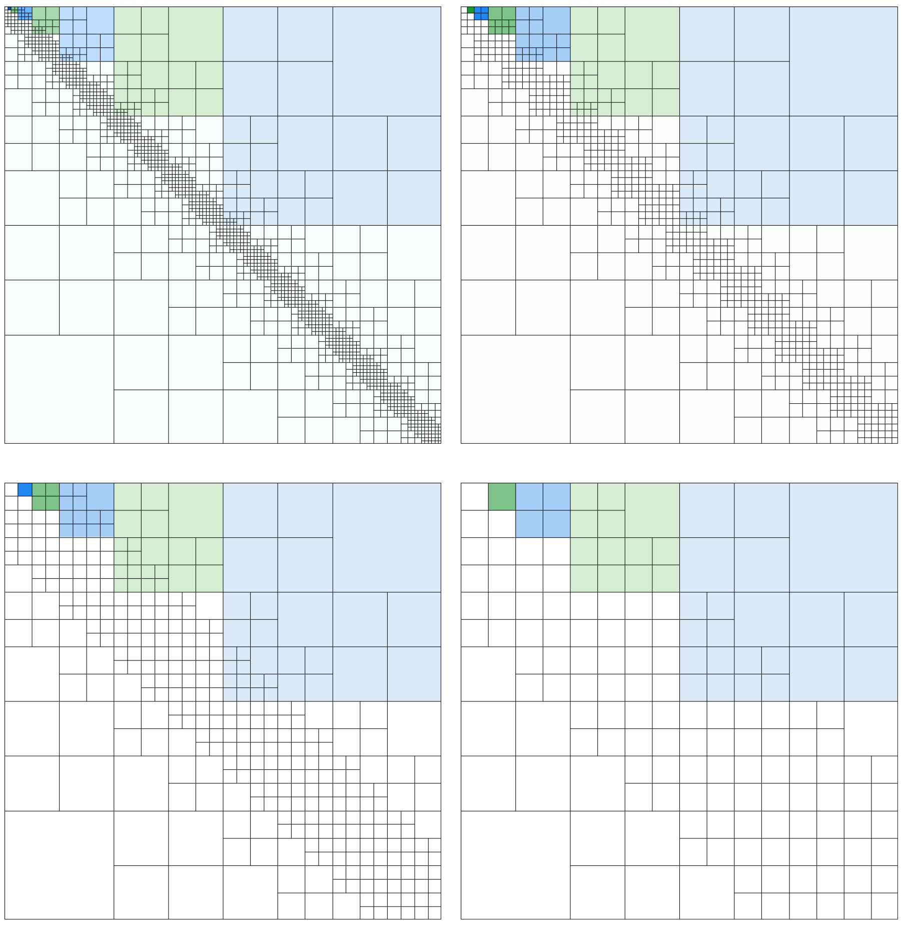

Figure 5.5: Hierarchical structures of the $\mathcal{H}$-Matrices employed in the evaluation of the parallelization strategies (with, at most, r = 7, 6, 5 and 4 recursive partitionings of the inadmissible blocks, corresponding respectively to the top-left, top-right, bottom-left and bottom-right matrix representations; note that the colors indicate the amount of levels defined in each structure).

Figure 5.6 reports the GFLOPS per core for the three different parallelization strategies described above. The results there correspond to a square $\mathcal{H}$-Matrix of dimension n = 10K with, at most, r = 4, 5, 6 and 7 recursive partitionings applied to the inadmissible blocks until a minimum leaf size is reached. This implies that the smallest blocks on the diagonal are of order $b_{\min}$ = 10K$/2^r \approx$ 625, 312, 156 and 78, respectively. This experiment offers some interesting insights:

- The performance of MKL greatly benefits from problems with large block sizes, which is consistent with the trends in the GFLOPS rates observed for the multi-threaded instance of Intel's D_GEMM in the previous experiment. This option is competitive with the task-parallel OmpSs-based routine when the number of cores is reduced or the partitioning features large diagonal blocks ($r = 4$, $b_{\min} = 625$).
- The parallel performance of OpenMP is practically negligible as the GFLOPS per core decrease linearly with the number of cores. This is not a total surprise as, due to the 2×2 organization of the $\mathcal{H}$-Matrix, the operations that can be performed independently are reduced to the two triangular system solves at each partitioning.
- When the number of cores is small, the OmpSs-based parallelization attains mild GFLOPS rates. Here, the coarse-grain partitioning of the blocks and the existence of synchronization points constrain the degree of parallelism that can be exploited and limit the performance of this approach when the number of cores is large.

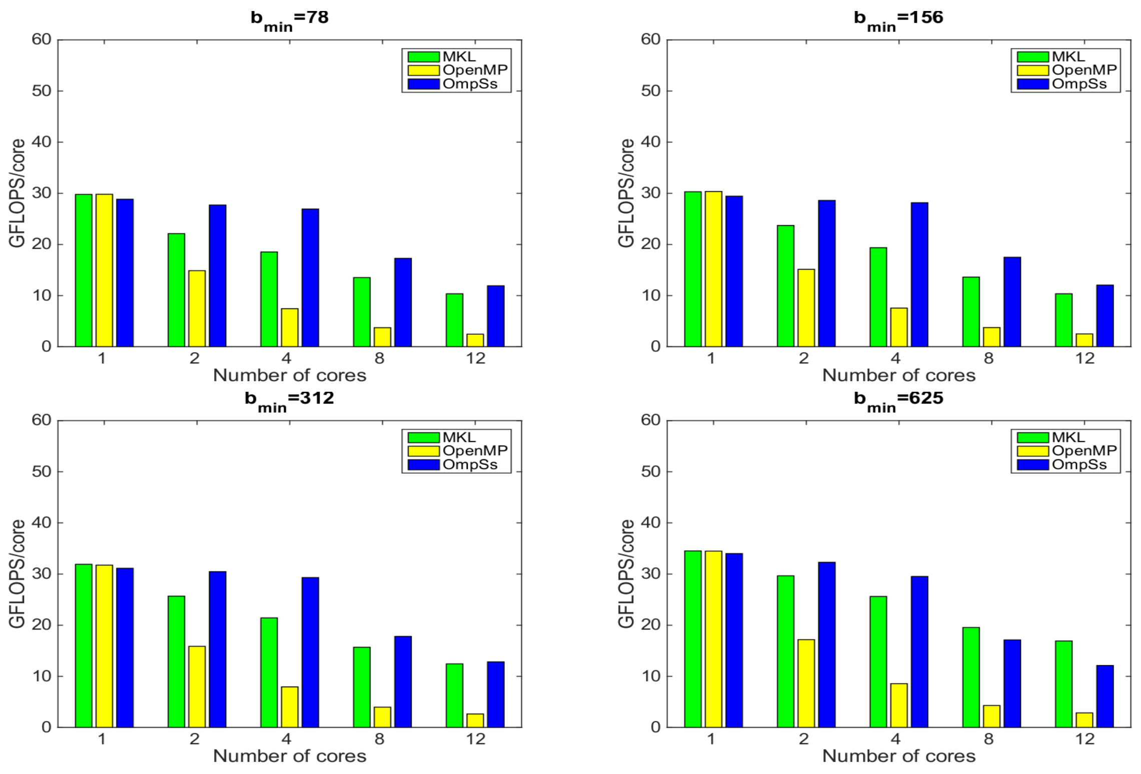

Figure 5.6: Performance of basic parallelization strategies applied to an $\mathcal{H}$-Matrix of order $n = 10$K, with dense blocks, and a recursive 2×2 hierarchical partitioning of the inadmissible blocks; see Figure 5.5.

To complete the analysis of this experiment, we remark that a comparative analysis of the GFLOPS observed in these executions with those of _GEMM is delicate. In particular, the execution using a single core can be expected to set the processor to operate at a higher frequency rate than a parallel multi-threaded execution using several cores. Unfortunately, the exact frequency is difficult to determine, as it depends on the number of cores as well as the arithmetic intensity of the operations (and it can even vary at execution time).

5.4.4.2 Scalability of task-parallel routines

Our next experiments aim to demonstrate the benefits that the WD+ER (weak dependency and early release) mechanisms exert on the scalability of the task-parallel codes based on OmpSs-2. For this purpose, we next conduct an analysis of the strong and weak scalabilities, using a complete node (48 cores) and the same hierarchical matrix employed in the previous study, with a 2×2 recursive structure defined on the diagonal blocks and dense blocks only.

In the following analysis of strong scalability, we set the problem dimension to three different values, n =10K, 15K and 30K, and progressively increase the amount of cores up to 48 while measuring the GFLOPS per core. In this type of experiment, we can expect that the GFLOPS/core rates eventually drop as the problem becomes too small for the volume of resources that are employed to tackle it. Figure 5.7 confirms that this is the case for both implementations, which exploit/do not exploit the new features in OmpSs-2 (lines labeled as with WD+ER and w/out WD+ER, respectively). In addition, the results also show that the exploitation of WD+ER, made possible by OmpSs-2, offers a GFLOPS/core rate that clearly outperforms that of the implementation that is oblivious of these options.

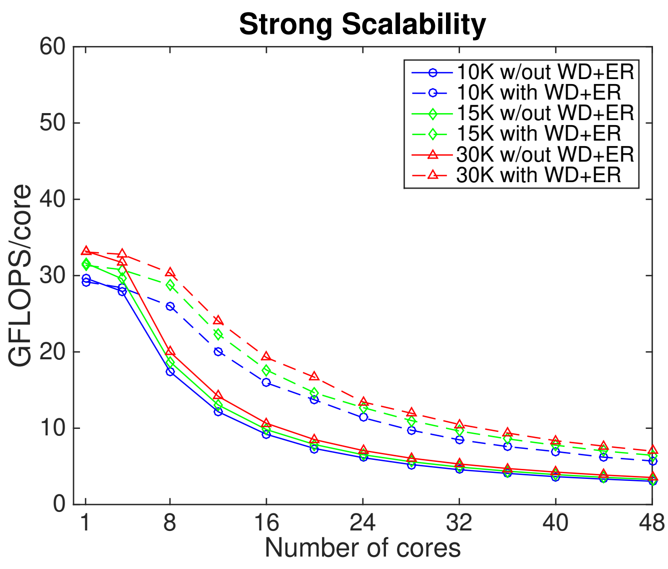

Figure 5.7: Strong scalability of the advanced parallelization strategies applied to $\mathcal{H}$-Matrices of order n = 10K, 15K and 30K (b_{min} =156, 234 and 234, respectively), with dense blocks, and a recursive 2×2 hierarchical partitioning of the inadmissible blocks; see Figure 5.5.

For the analysis of weak scalability (see Figure 5.8), we utilize a problem of dimension $n \times n$ that grows proportionally to the number of cores c, so that the ratio n^2/c = 12K×12K holds while c grows to 48. As the problem size per core is constant, we can expect that the GFLOPS/core remains stable, showing the possibility of addressing larger problems by increasing proportionally the amount of resources up to a certain point. (This is not totally exact, as the cost of the factorization for dense matrices grows cubically with the problem dimension while, in the conditions set for this experiment, the amount of resources only does so quadratically.) Unfortunately, the results of this experiment reveal that the weak scalability of both algorithms suffers an important drop as the number of cores is increased, though in the variant equipped with WD+ER this occurs in the transition from 8 to 12 cores while, in the implementation that does not exploit these mechanisms, the gap is already visible in the increase from 4 to 8 cores.

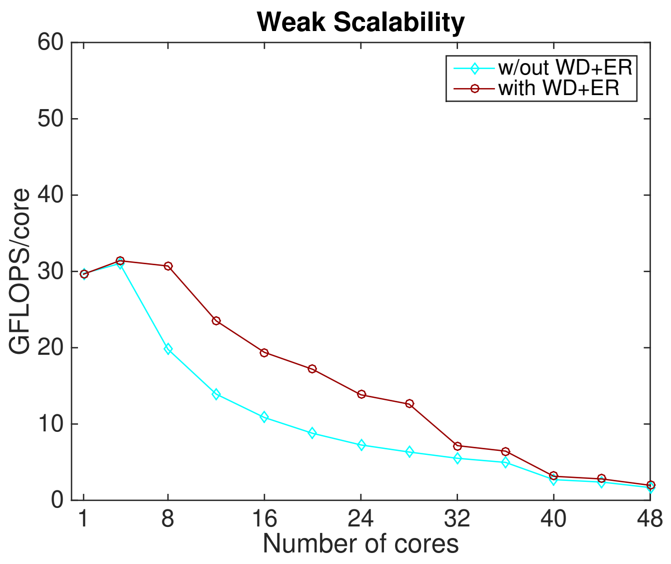

Figure 5.8: Weak scalability of the advanced parallelization strategies applied to an $\mathcal{H}$-Matrix of dimension $n \times n$ = 12K×12K per core ($b_{\min}$ =234 in all cases, except with 8 cores where $b_{\min}$ =166), with dense blocks, and a recursive 2×2 hierarchical partitioning of the inadmissible blocks; see Figure 5.5.

There are two aspects to take into account when considering the GFLOPS/core rates observed in the strong scaling analysis and, especially, the weak scaling counterpart. The first one refers to the CPU frequency, which decreases with the number of cores that are active (see [52] and Figure 5.4) and affects the performance of the task-parallel routines, reducing it with the number of cores. The second one is a consideration of the structure of the hierarchical matrix employed in these experiments (see Figure 5.5). In particular, when all the blocks are dense, and the matrix is decomposed into a task per block in this partitioning, the result is a problem where a reduced collection of coarse-grain tasks concentrate a large fraction of the flops. This effect is exacerbated with the problem order (n) and its negative impact is more visible when the number of cores is increased because the task-parallel algorithms confront then a suboptimal scenario consisting of a very reduced number of tasks (low degree of task-parallelism) of (very) coarse-grain operations.

5.4.4.3 Parallelism of task-parallel routines with low-rank cases

Our final round of experiments assesses the performance of the WD+ER mechanism using several BEM cases, of dimensions d =1, 2 and 3, involving low-rank blocks. The "sparsity" pattern of these blocks is controlled via a parameter η that we set to four different values, 0.25, 0.5, 1.0 and 2.0. The structure of these cases is illustrated in Figure 5.9.

The bottom-left $\mathcal{H}$-Matrix in Figure 5.9 is close to a dense one. Solving a dense problem with a specialized version of our software is possible; however it should be more efficient to do so with a solution that stores the matrix employing a regular column-major structure, and then use a general runtime software specialized for dense matrices, for example, based on OpenMP. For this reason, a

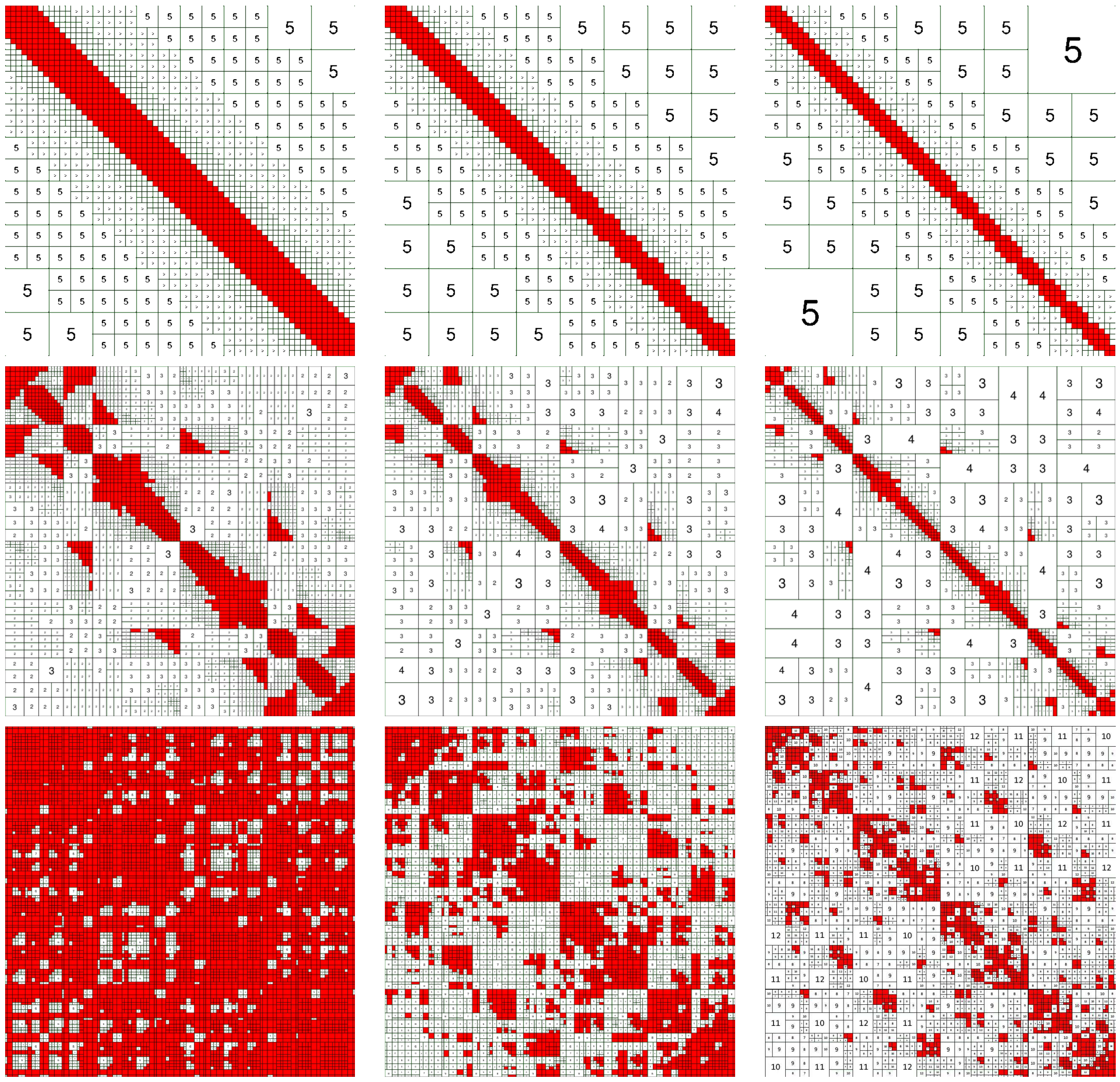

Figure 5.9: Hierarchical structure of the $\mathcal{H}$-Matrix of dimension 30K employed in the evaluation of the task-parallel routines. The red areas denote dense blocks and the number inside the white blocks specifies the rank of the corresponding (low-rank) block. From top to bottom: d =1, 2 and 3; and from left to right: η =0.25, 0.5, 1.0 in the first two rows, and η =0.5, 1.0, 2.0 in the last one.

comparison of the timings and speed-up achieved in this scenario by using $\mathcal{H}$-Arithmetic over full arithmetic, as we consider that it would be out of the scope of this study.

Tables 5.1 and 5.2 report the acceleration factors (or speed-ups) attained by the task-parallel codes with respect to the corresponding sequential code/case, using problems of order $n \approx$ 30K with up to 24 cores and order $n \approx$ 42K with up to 48 cores, respectively.

These final experiments illustrate the performance advantage of exploiting the WD+ER also for $\mathcal{H}$-Matrices with low-rank blocks. In general, the speed-up increases with the ratio of dense blocks, reporting notably high values for $d = 3$ (provided the number of cores is not too large compared

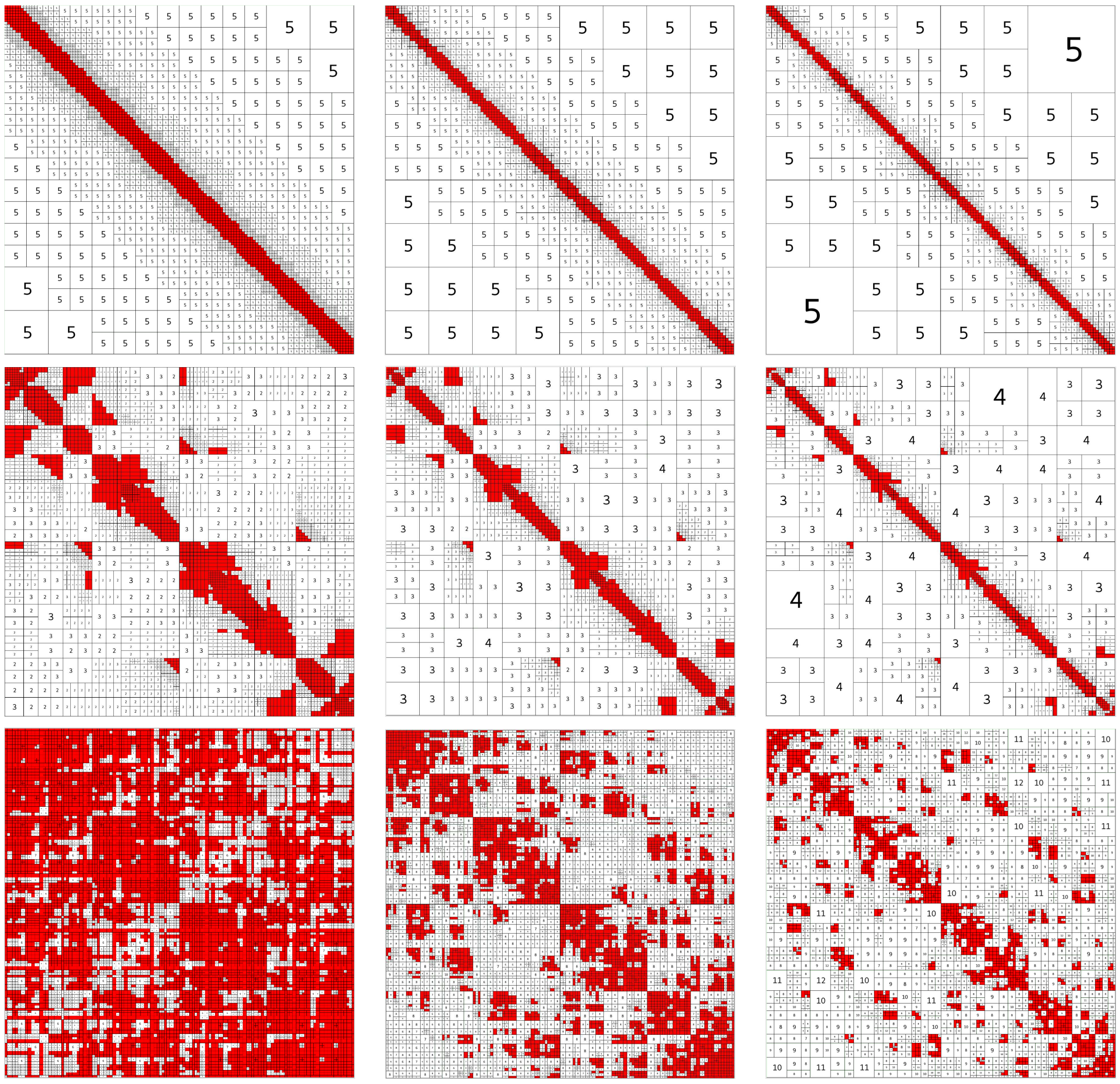

Figure 5.10: Hierarchical structure of the $\mathcal{H}$-Matrix of dimension 42K employed in the evaluation of the task-parallel routines. The red areas denote dense blocks and the number inside the white blocks specifies the rank of the corresponding (low-rank) block. From top to bottom: d =1, 2 and 3; and from left to right: η =0.25, 0.5, 1.0 in the first two rows, and η =0.5, 1.0, 2.0 in the last one.

with the problem dimension), and lower for those cases with $d = 1,2$. These differences in the speed-up can be justified by analyzing the structure of the matrices. Concretely, for $d = 3$, the recursive partitioning of the inadmissible blocks leads to leaf blocks which are quite balanced; this entails more uniform task granularities than in $d = 1,2$ cases. Moreover, in the $d = 3$ case, there exists a larger amount of blocks, which results into a larger number of tasks and helps to expose a higher concurrence degree in that particular case. In some cases we even observe a super-linear speed-up, due to cache effects.

η	d	WD+ ER?	Seq. time	Speed-up with #cores 4	8	12	16	20	24
0.25	1	No	89.2	3.51	5.24	5.58	5.70	5.70	5.69
		Yes		4.05	7.82	11.56	14.62	17.37	19.04
	2	No	118.9	3.70	6.12	7.28	7.79	7.88	8.02
		Yes		3.96	7.64	10.99	13.55	17.36	18.49
0.50	1	No	38.1	2.60	2.74	2.73	2.71	2.71	2.70
		Yes		3.92	7.45	9.57	9.74	9.69	9.45
	2	No	37.0	3.04	3.68	3.82	3.90	5.74	3.92
		Yes		3.96	6.87	8.81	9.68	10.18	10.44
	3	No	1,099.2	4.00	7.83	13.48	14.44	16.88	18.71
		Yes		4.00	7.99	11.85	15.63	19.13	21.57
1.00	1	No	12.6	1.55	1.58	1.57	1.57	1.56	1.57
		Yes		2.50	2.46	2.48	2.46	2.44	2.43
	2	No	12.0	1.92	1.99	2.00	2.00	1.99	2.00
		Yes		3.31	4.22	4.38	4.50	4.44	4.36
	3	No	1,049.8	3.96	7.54	10.84	13.28	15.64	17.63
		Yes		4.03	7.91	14.32	15.55	18.99	21.73
2.00	3	No	204.1	3.59	5.78	7.39	7.88	8.26	8.44
		Yes		4.99	7.87	11.27	14.65	17.33	17.83

Table 5.1: Execution time of the sequential algorithm in H2Lib (in sec.) and parallel speed-up of the advanced parallelization strategies applied to an $\mathcal{H}$-Matrix of order $n \approx$ 30K ($b_{\min}$ =234), with dense and low-rank blocks; see Figure 5.9.

η	d	WD+ ER?	Seq. time	Speed-up with #cores 8	16	24	32	40	48
0.25	1	No	57.5	4.64	5.18	5.17	5.16	5.04	5.20
		Yes		7.93	14.51	16.89	18.26	18.28	17.56
	2	No	300.8	6.04	7.68	7.82	7.95	7.97	7.93
		Yes		7.81	14.32	15.52	16.81	17.45	17.91
0.50	1	No	21.7	2.81	2.96	3.01	2.96	2.96	2.95
		Yes		6.82	8.11	8.22	7.93	7.96	7.91
	2	No	102.4	3.79	4.07	4.10	4.08	4.08	4.10
		Yes		6.95	9.18	9.63	9.84	9.89	9.95
	3	No	3,819.8	7.78	14.73	19.82	25.32	29.36	31.99
		Yes		7.92	15.82	22.05	29.23	36.33	42.05
1.00	1	No	10.9	1.89	1.91	1.91	1.89	1.90	1.90
		Yes		4.98	5.01	4.89	4.80	4.75	4.76
	2	No	34.5	1.76	1.77	1.76	1.76	1.76	1.76
		Yes		3.27	3.37	3.35	3.32	3.30	3.34
	3	No	1,332.8	7.38	12.82	16.41	18.83	20.53	21.44
		Yes		7.91	15.54	21.79	28.69	35.19	40.26
2.00	3	No	291.1	5.73	7.43	8.03	8.13	8.30	8.31
		Yes		7.86	14.89	19.73	24.03	26.87	27.87

Table 5.2: Execution time of the sequential algorithm in H2Lib (in sec.) and parallel speed-up of the advanced parallelization strategies applied to an $\mathcal{H}$-Matrix of order $n \approx$ 42K ($b_{\min} \approx 500$), with dense and low-rank blocks; see Figure 5.10.

5.5 Concluding remarks

In this chapter, we have demonstrated a fair parallel efficiency for the calculation of the $\mathcal{H}$-LU factorization on a state-of-the-art Intel Xeon socket with 24 cores. A key component to attain this high performance is the exploitation of weak dependencies and early release introduced in OmpSs-2. Armed with these mechanisms, the OmpSs-based parallel codes can cross the dependency domains, discovering and exploiting a notably higher degree of task-parallelism, which results in higher performance in the execution of 1D, 2D and 3D cases arising from BEM.

Chapter 6

Moving further: $\mathcal{H}$-Chameleon, a parallel $\mathcal{H}$-library

Contents of the chapter

6.1 Introduction

The analysis in Chapter 4 confirmed that task-parallelism is a suitable approach to accelerate the calculations with $\mathcal{H}$-Matrices. In consequence, we opted for parallelising the implementation of the $\mathcal{H}$-LU in H2Lib employing OmpSs-2 as shown in Chapter 5, and we achieved a fair parallel performance (concretely, a speedup of 42× when using up to 48 cores in the best case). However, accomplishing that efficiency would have not been possible without using weak dependencies and early release of tasks, together with a good annotation of tasks with regards to (nested) regions instead of only representatives. This exposes two issues that should be emphasized: 1) weak dependencies and early release are only available in OmpSs-2, while other widely used programming models, such as OpenMP, do not include them and, thus, cannot be employed with the expectation of reaching such performance; and 2) the complexity of pure $\mathcal{H}$-Matrices implies dealing not only with recursion and nesting (both in terms of algorithms and the data itself), but also with data locality and re-sizing operations due to the re-compression of certain low-rank blocks.

With these lessons learnt, we decided to explore a strategy that allowed us to 1) leverage the task-parallelism when applied to regular tiles in contrast to extracting parallelism over many different block sizes and the nested structure; and, at the same time, 2) preserve the benefits of using $\mathcal{H}$-Matrices, that is, maintain a logarithmic cost both in terms of storage and execution time. In summary, we wanted to identify more regular tasks (both in terms of size and desirably load), while leveraging the advantages that make an $\mathcal{H}$-Matrix profitable in order to produce efficient solvers for linear systems arising in BEM.

Thus, the main goal of the work presented in this chapter is to validate whether it is possible to obtain an efficient open-source library for $\mathcal{H}$-Matrices at a *small development* cost leveraging existing libraries, software, programming models and runtime systems to scale $\mathcal{H}$-Matrices solvers, via the classic tile-based approach commonly used in dense linear algebra, in order to deliver good parallel performance. The work that we will describe in this chapter summarizes the effort of building $\mathcal{H}$-Chameleon, an extension of the Chameleon library [2, 3, 38], so that combined with Hmat-oss library [64, 86], this is capable of composing factorizations such as the $\mathcal{H}$-LU over a particular type of matrices that we named Tile $\mathcal{H}$-Matrices. In short, these special structures are matrices that have been subdivided into regular tiles in a first level, with each of those tiles converted into an $\mathcal{H}$-Matrix.

The rest of the chapter is structured as follows: we first describe the basics of $\mathcal{H}$-Chameleon, that is, the building blocks employed to form the new library, in particular the storage layout and clustering algorithm, necessary for storing and building the Tile $\mathcal{H}$-Matrices respectively, as well as the kernels that operate over $\mathcal{H}$-Matrices; next we analyse the performance analysis results we have obtained from the experiments with $\mathcal{H}$-Chameleon; and finally we highlight some remarks.

6.2 The basics of $\mathcal{H}$-Chameleon

Improving the performance of $\mathcal{H}$-Arithmetic operations is an active area of research that has recently produced a fair number of libraries, as well as many interesting algorithmic developments (see Section 1.2). These research efforts are surely motivated by the relevance of the applications that can be efficiently tackled with $\mathcal{H}$-Matrices, but also because of the complexity and benefits of $\mathcal{H}$-Arithmetic. On the one hand, $\mathcal{H}$-operations require dealing with both low-rank and dense blocks – which often implies re-compressing some of the intermediate results – while following a nested structure, and usually a recursive algorithm. On the other hand, in general, the definition and storage of $\mathcal{H}$-Matrices leads to complex data accesses. This fact promoted the appearance of alternative structures, such as BLR [10, 89, 103] and lattice $\mathcal{H}$-Matrices [68, 116], that trade off slightly higher time and memory costs in exchange for superior simplicity. One asset of these approaches it that they make it easier to exploit parallelism, as they present more regular structures.

In the work presented in this chapter, we combine existing efficient numerical kernels for hierarchical, low-rank and full-rank matrices, together with an efficient task-based implementation designed to solve dense linear systems. As we will show, this allows us to leverage modern programming models and runtime systems yielding to a single open-source solution that achieves fair parallel performance, while avoiding re-implementing pure $\mathcal{H}$-Arithmetic. Concretely, our work integrates the following three components:

- For $\mathcal{H}$-Arithmetic, we choose the Hmat [64, 86] kernels to operate with low-rank blocks, as they have been proved to offer fair efficiency in industrial applications [86]. This is a library for $\mathcal{H}$-Matrices maintained by Airbus, whose public version (Hmat-oss) is sequential.
- In addition, for the task-based implementation of matrix operations, we leverage the Chameleon library [2, 3, 38], a dense linear algebra package that relies on the sequential task flow programming model to schedule tiled algorithms on top of runtime systems such as OpenMP [99], PaRSEC [66], StarPU [17, 110], or Quark [117].

- Finally, we focus on the specific StarPU runtime system [17] support of the Chameleon library to exploit task-parallelism in our solution. StarPU is in charge of issuing the tasks to the system cores while fulfilling inter-task dependencies.

Our strategy to re-utilize structures that are similar to $\mathcal{H}$-Matrices while reducing their complexity is close to what is referred to as "lattice $\mathcal{H}$-Matrices" in [68, 116]. As opposed to those works, in our case we need to split the initial matrix into regular tiles (i.e., tiles of the same size) to meet Chameleon's algorithmic requirements, and each of these tiles are individually turned into $\mathcal{H}$-Matrices. We will refer to our structure as a "Tile $\mathcal{H}$-Matrix".

In the next sections, we present details about the building blocks, the data layout, the clustering algorithm implemented to build Tile $\mathcal{H}$-Matrices, and the kernels from Hmat-oss that we invoke from Chameleon.

6.2.1 The building blocks

$\mathcal{H}$-Chameleon relies on three building blocks or components: the dense linear algebra library Chameleon, the runtime system StarPU, and the $\mathcal{H}$-Matrices library Hmat-oss.

Chameleon [2, 3] is an open source software for dense linear algebra written in C. It relies on the sequential task flow programming model supported by runtime systems such as OpenMP, StarPU, PaRSEC-DTD, and Quark. All dense linear algebra algorithms are expressed as tile algorithms with sequential tasks that are submitted to the underlying runtime system. The runtime automatically infers the data dependencies in these algorithms thanks to keywords that specify the data accesses. Chameleon covers all BLAS, as well as one-sided factorizations (Cholesky, LU, QR), and supports multiple runtime systems to schedule the tasks. StarPU is one of them, and the most integrated one into Chameleon, besides being the second building brick of our proposal.

StarPU provides tools to describe the pieces of data, such that the data transfers from device to device or node to node, are transparent to the algorithm developer. This is a key feature when developing $\mathcal{H}$-Matrices algorithms, as it will suffice to provide the runtime with pack/unpack functions in order to transfer data (if needed).

Hmat-oss is the open source version of the Hmat library developed by Airbus. It is a sequential library written in C++. This library provides up-to-date implementations of $\mathcal{H}$-Matrix operations and techniques, clustering algorithms and orderings, and real world applications examples. The non-public Hmat library will be employed as a performance reference or baseline in the experiments shown later in this chapter.

6.2.2 The storage layout for Tile $\mathcal{H}$-Matrices

The baseline realization of Chameleon only supports dense matrices, which are stored employing a descriptor that specifies, among other information, the matrix dimensions, number and size of the tiles which define the matrix partitioning, pointers to data addresses in memory, and some control parameters to test certain features (e.g., whether there is an overall data pointer).

In order to accommodate Tile-$\mathcal{H}$ Matrices in our hierarchical extension of Chameleon, we have expanded the reference descriptor structure to accomodate a collection of tiles. Each of these tiles is potentially an $\mathcal{H}$-Matrix, a low-rank block, or a full-rank matrix. To do this change, we enriched the Chameleon matrix descriptor (Listing 6.1) with an array for the new tile structures, and a helper function, `get_blktile`, to extract the correct tile pointer from the tile indices in the matrix.

```
typedef struct chameleon_desc_s {
    ...
    blktile_fct_t  get_blktile;
    CHAM_tile_t   *tiles;
    ...
} CHAM_desc_t;
```

Listing 6.1: `CHAM_desc_t` datatype modifications to handle more generic tile formats.

In addition to the main data structure, the tile description was transformed from the simple data pointer used in runtime systems, such as OpenMP or Quark, to handle the dependencies into a more complex structure that was able to handle different data formats. The new datatype, `CHAM_tile_t` (Listing 6.2), allows to simply store different matrix formats, defined by the `format` field, and a pointer to the matrix, `mat`, which can be either a full-rank matrix, or a more complex datatype, such as an $\mathcal{H}$-Matrix coming from an external library. The data dependencies that were initially tracked down using the pointer to the data in the full-rank matrix are thus now followed by the pointer to this tile descriptor. This means that all the algorithms from the Chameleon library could work out-of-the-box with an $\mathcal{H}$-Matrix format, assuming that there exist kernels to handle this type of matrices.

```
typedef struct chameleon_tile_s {
    int8_t format;
    int    m, n, ld;
    void  *mat;
} CHAM_tile_t;
```

Listing 6.2: `CHAM_tile_t` data structure to accommodate any format of tiles in the Chameleon library.

Finally, the global representation of the matrix that links a Chameleon descriptor (`CHAM_desc_t`) with an Hmat-oss descriptor (`hmat_matrix_t`) in a unique element is done via an additional new data structure (Listing 6.3).

```
struct HCHAM_desc_s {
    CHAM_desc_t              *super;
    hmat_cluster_tree_t    **clusters;
    hmat_admissibility_t    *admissibilityCondition;
    hmat_interface_t        *hi;
    hmat_matrix_t           *hmatrix;
    int                     *perm;
};
```

Listing 6.3: `HCHAM_desc_s` structure created to handle the complexity of the Tile-$\mathcal{H}$ structure.

In this new datatype, `super` represents the Chameleon library tile descriptor; `clusters` stores an array of the CTs created to partition the original data; `admissibilityCondition` contains this parameter value, necessary to determine whether a certain block is already admissible (and consequently converted to a low-rank block) or needs to be re-partitioned; `hi` is the interface defined in the Hmat-oss library to deal with $\mathcal{H}$-Matrices, which contains general information (for example, the clustering algorithm needed to construct the $\mathcal{H}$-Matrix; and data precision format); `hmatrix` contains the $\mathcal{H}$-Matrix built employing Hmat-oss construction kernels (which is the descriptor of the library whose content will be employed in the Hmat-oss kernels operations); and `perm` stores the permutation array.

6.2.3 The clustering algorithm to build Tile $\mathcal{H}$-Matrices

In order to use Tile-$\mathcal{H}$ matrices in the Chameleon library, we need an adapted clustering tree. Indeed, the Chameleon library works exclusively on regular tile sizes, with the exceptions of the padding row and column. Thus, it is not sufficient to flatten the first levels of the clustering tree as it is done in [116]. We extended the Hmat-oss library with a recursive tile clustering algorithm, named "`NTilesRecursive`", that recursively divides a given CT into clusters that follow a regular partitioning into tiles of size `NB`.

This process is illustrated in Algorithm 8, and the parameters and functions employed are described next:

- `CT` is the CT to partition;
- `axis` is the main axis of the current slice, which is exploited by the geometric clustering techniques to split along the largest dimension;
- `NB` is the desired tile size;
- `offset` represents the coordinates (values) of the first value in the current CT;
- `size` is the size of the current CT;
- the function `slice` returns a portion of the current cluster according to the given offset and size;
- the function `largestDimension` returns the largest dimension in the current cluster; and
- the function `sortByDimension` orders the current cluster of unknowns according to the given dimension;

At each level, this function performs a pseudo-bisection aligned with the tile size along the largest dimension and returns the concatenation of the recursive call to each subset of unknowns. This provides a regular clustering of the unknowns that matches both the constant size requirement of the Chameleon library and the Tile-$\mathcal{H}$ format. A median bisection algorithm is then called within each cluster to refine the clustering of each tile.

6.2.4 The Hmat-oss kernels

In order to implement a hierarchical LU factorization, we leveraged a number of sequential numerical kernels from the Chameleon and Hmat-oss libraries, which provide the necessary operations to factorize our Tile-$\mathcal{H}$ matrix. To this end, we modified the main kernels of the Chameleon library because, like the data dependencies tracking system, they integrated pointers to full-rank matrices. To limit the changes to the library, we just introduced an intermediate layer to enable the switch between full-rank and $\mathcal{H}$-kernels. Thus, the task insertion functions are not modified, and the

Algorithm 8 NTilesRecursive clustering algorithm to build Tile $\mathcal{H}$-Matrices

```
Ensure: A (sub)partitioned CT defining the Tile $\mathcal{H}$-Matrix structure.
 1: procedure NTilesRecursive(CT, NB, offset, size, axis)
 2:     $n_t = \lceil \frac{\text{size}}{NB} \rceil$
 3:     if $n_t == 1$ then return CT
 4:     end if
 5:     dim = getLargestDimension($CT, axis$)
 6:     sortByDimension(CT, dim)
 7:     $\text{offset}_L = \text{offset}$
 8:     $\text{size}_L = NB * \lceil \frac{n_t}{2} \rceil$
 9:     $\text{offset}_R = \text{offset} + \text{size}_L$
10:     $\text{size}_R = \text{size} - \text{size}_L$
11:     $\text{CT}_L = \text{slice}(\text{CT}, \text{offset}, \text{offset} + \text{size}_L)$
12:     $L = \text{NTilesRecursive}(\text{CT}_L, NB, \text{offset}_L, \text{size}_L, \text{dim})$
13:     $\text{CT}_R = \text{slice}(\text{CT}, \text{offset} + \text{size}_L, \text{size}_R)$
14:     $R = \text{NTilesRecursive}(\text{CT}_R, NB, \text{offset}_R, \text{size}_R, \text{dim})$ return $(L, R)$
15: end procedure
```

`CHAM_tile_t` datatype helps us to switch from one kernel type to another, thanks to the `format` field.

On the Hmat-oss library side, we provide a similar interface to BLAS for `GETRF`, `TRSM`, and `GEMM` operations, and an intermediate internal layer which allows us to switch from this $\mathcal{H}$-Matrix interface to the more classical BLAS interface.

6.3 Performance analysis

In this section, we first present the source environment for the data, and a description of the PlaFRIM platform used for the experiments. Afterwards, we analyze the parallel performance of $\mathcal{H}$-Chameleon on a multicore system.

6.3.1 Experimental context

The test case used is the TEST_FEMBEM [111] application, which generates a real or complex matrix, which has features similar to real industrial applications in aeronautics. For a number of unknowns n, we create a cloud of points $(x_i)_{1 \leq i \leq n}$ located on the surface of a cylinder of chosen height and width, as illustrated in Figure 6.1. These points are equally spaced in both directions on the cylinder surface. Then, we define the interaction kernel between two points x_i and x_j separated by a distance $d = |x_i - x_j|$, with $K(d) = \exp(ikd)/d$ in the complex case, and $K(d) = 1/d$ in the real one. In the complex case, k plays the role of a wave number, and it is chosen with the "rule of thumb" of having 10 points per wavelength, which is commonly used in the wave propagation community. The singularity at $d = 0$ is simply removed by setting d equal to half the mesh step in that case.

In the real case, the rank of the $\mathcal{H}$-Matrix blocks is mostly independent of their sizes, and therefore most of the data (in terms of storage) is located near the diagonal of the matrix. In the complex

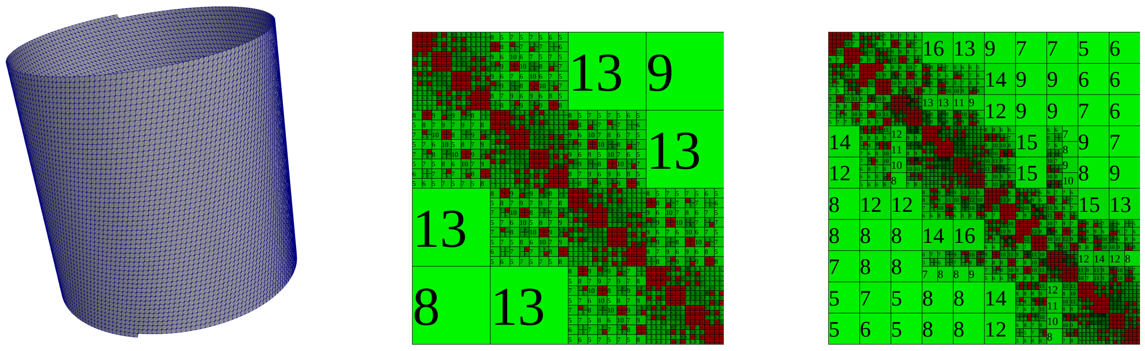

Figure 6.1: Illustration of the test case used in the experiments. On the left, the mesh of the cylinder with the distribution of the unknowns on the surface for 10K points. In the middle, the associated compressed real matrix in the HMAT format (classical $\mathcal{H}$-Matrix). On the right, the associated compressed real matrix in the proposed fixed-sized Lattice or Tile-$\mathcal{H}$ based matrix format. In the matrices, low-rank blocks are represented in green (with a number specifying the rank), while dense blocks are coloured in red.

case, the rank grows with the size of the blocks, and the data is much more evenly distributed in the matrix. Hence, the amount of storage and work is far more important in the complex case, and the work-load distribution is much more challenging too.

6.3.2 Experiments platform

All our experiments have been performed on the PlaFRIM [104] test bench, and more specifically on the `bora` cluster. Each node is equipped with two Intel Xeon Skylake Gold 6240 processor (with 18-cores per processor), running at 2.60 GHz, and equipped with 192 GB of memory. The application is compiled with GCC 9.2.0, and Intel MKL 2019 is used for the BLAS and LAPACK kernels. StarPU 1.3.0 is used, and our proposal is built on revisions 33aa719 of Hmat-oss [64, 86] and 08cf0cd1 of Chameleon [2, 3, 38]. (Note that, when evaluating $\mathcal{H}$-Chameleon, its performance results are compared to the H-Mat proprietary library, and not to Hmat-oss.)

6.3.3 Experiments

Our first experiment aims to demonstrate that our implementation, though simpler than constructing a classical $\mathcal{H}$-Matrix, still enables a good compression ratio. To this end, Figure 6.2 shows a comparison between the Hmat-oss (dashed lines) and $\mathcal{H}$-Chameleon (full lines) compression ratios for real (left plot) and double precision (right plot) precisions, employing different matrix dimensions, from $10K$ to $200K$, and various tile sizes, from 500 to $10K$. The results show a negligible difference in all cases, so that we can affirm that the clustering with fixed tile sizes does not impact the compression ratio on the studied test case, and can even provide better results than the classical median bisection used in Hmat(`-oss`).

Second, as precision is the *bargaining chip* in $\mathcal{H}$-Scenarios to permit time and memory savings, it is also necessary to control that the proposed clustering does not affect the numerical accuracy of the $\mathcal{H}$-Chameleon $\mathcal{H}$-LU factorization operation. To prove this, Figure 6.3 presents forward error measurements, defined as $||x - x_0||_f/||x||_f$, for different $\mathcal{H}$-LU executions with the same matrix configurations employed in the previous experiment on the compression ratio. Note that the accuracy parameter is set at 10^{-4}, both in Hmat and $\mathcal{H}$-Chameleon. The largest observable differences are around $1.5 \cdot 10^{-4}$, which means we stay within the same order of magnitude.

Figures 6.4 and 6.5 offer multicore parallel performance comparisons (employing up to 35 threads), for various matrix dimensions, from $10K$ to $200K$, both in real and complex double precision scenarios. These figures study different scheduling strategies proposed by the StarPU runtime system, while comparing their performance to that provided by the StarPU based implementation of the (parallel closed-source) Hmat library that deals with all the fine grain dependencies.

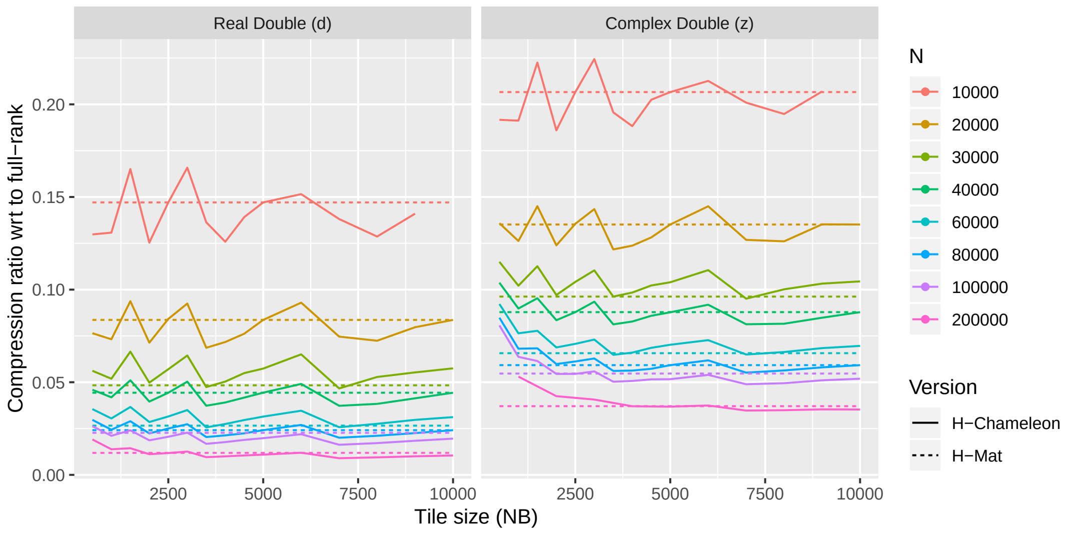

Figure 6.2: Comparison of the compression ratio between the Hmat-oss original clustering algorithm (dashed line) and that obtained with $\mathcal{H}$-Chameleon (full lines), as function of the tile size, for real precision (left) and complex precision data (right).

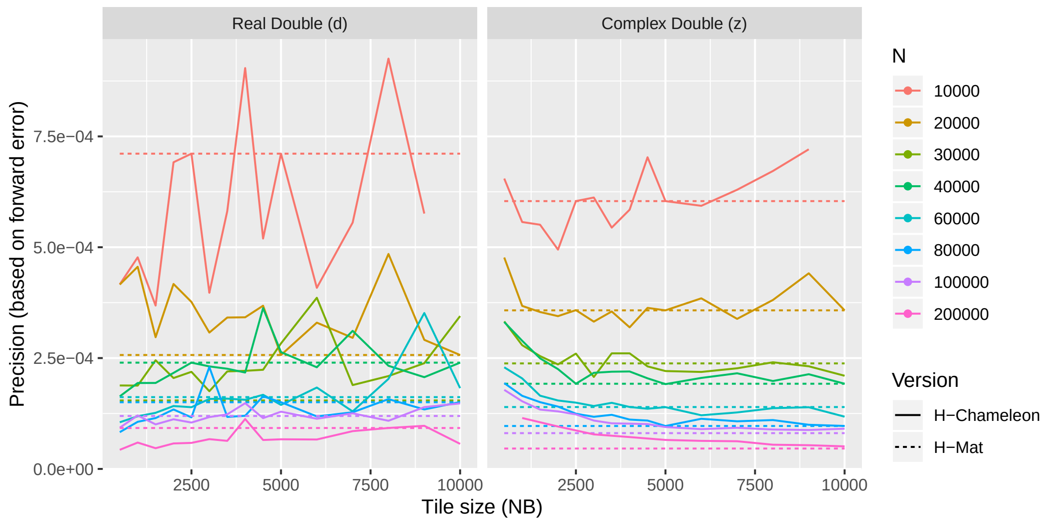

Figure 6.3: Comparison of the precision of the solver (based on forward error) between the Hmat-oss original clustering algorithm (dashed line) and that obtained with $\mathcal{H}$-Chameleon ntiles clustering algorithm (full lines) in function of the tile size, for real precision (left) and complex precision data (right).

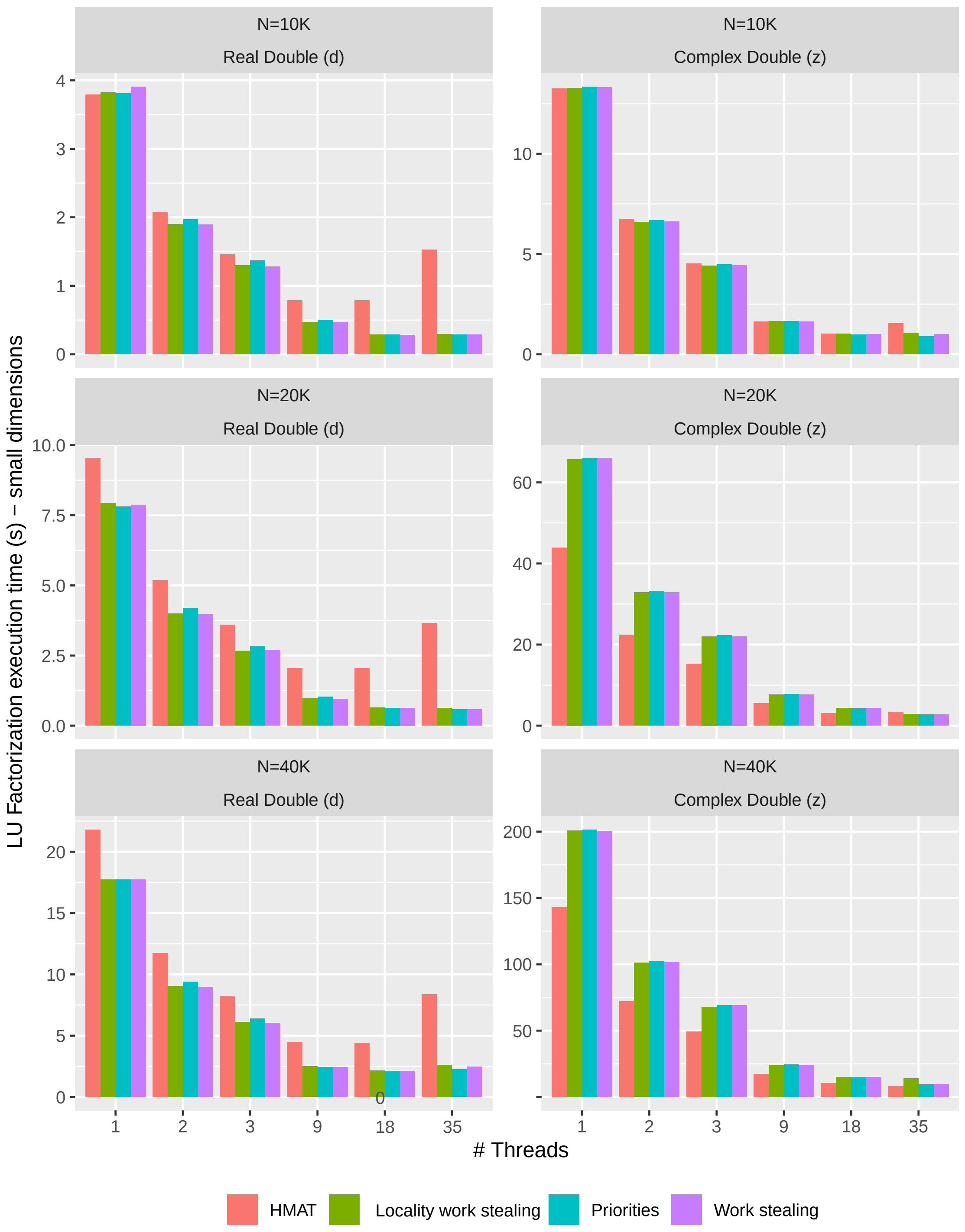

Figure 6.4: Comparison of the multicore parallel executions between $\mathcal{H}$-Chameleon and Hmat-oss LU factorization, employing up to 35 threads. Results are shown for real (left) and complex (right) double precision, and for small matrix dimensions: $10K$ (with $NB = 250$ for d, $NB = 500$ for z), $20K$ (with $NB = 500$), $40K$ (with $NB = 1000$).

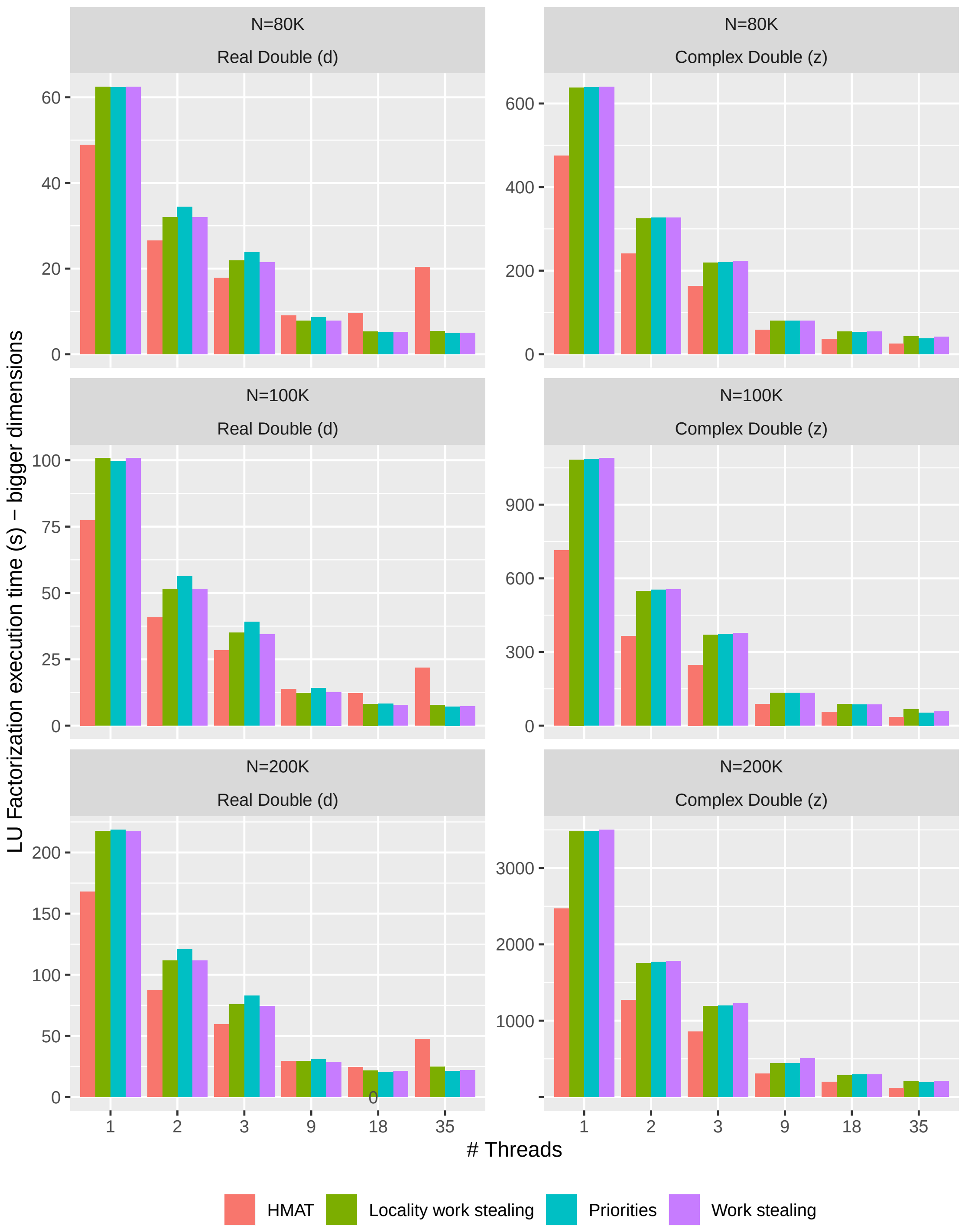

Figure 6.5: Comparison of the multicore parallel executions between $\mathcal{H}$-Chameleon and Hmat-oss LU factorization, employing up to 35 threads. Results are shown for real (left) and complex (right) double precision, and for larger matrix dimensions: $80K$ (with $NB = 1000$ for d, $NB = 2000$ for z), $100K$ (with $NB = 1000$ for d, $NB = 2000$ for z), $200K$ (with $NB = 2000$ for d, $NB = 4000$ for z).

Three scheduling strategies are studied:

- The Work Stealing (WS) strategy uses a queue per worker and schedules the tasks on the worker which released them by default. Whenever a worker becomes idle, it steals a task from the most loaded worker.
- The Locality Work Stealing (LWS) strategy similarly uses a queue per worker that is now sorted by the priorities assigned to the tasks. New ready tasks are scheduled on the worker which released them by default. Whenever a worker becomes idle, it steals a task from neighbor workers while respecting the priority order.
- The priority-based (`prio`) approach uses a central task queue in which ready tasks are sorted by decreasing priority. All threads try to pull work out of this central queue.

Note that for our proposed implementation we never employ more than 35 worker threads to keep a core dedicated to the task submission. The experiments have shown that this was more efficient than oversubscribing the system with 36 worker threads plus the thread in charge of the task submission.

In the $\mathcal{H}$-Chameleon implementation, all tile sizes presented in the compression and accuracy curves were tested and we chose the best one for each dimension and precision.

In most of the test cases, we observe that $\mathcal{H}$-Chameleon presents slower execution times when using 1, 2 or 3 threads. As the tile size is optimized for the 35-thread case, this induces an overhead of memory and required flops, which impacts the executions with a low number of threads. However, when a larger number of threads is used, this is compensated by the higher degree of parallelism it exposes, and so it enables a good scalability of the library. Hmat is not impacted by the tile size and manages to deliver higher performance for the execution on the small numbers of threads.

The comparison of the real and complex double precision results shows that Hmat is superior in the complex cases, while $\mathcal{H}$-Chameleon shows a better scaling in the real ones. This can be explained by the difference in the number of operations of the two test cases, due to the arithmetic and to their configuration, as previously explained in Section 6.3.1. In the complex-arithmetic scenario, the overhead of the kernels is high enough to amortize the overhead of handling the large number of dependencies, which benefits Hmat. However, in the real case, the cost of handling all fine grain dependencies becomes too high with respect to the cost of the computational tasks, and therefore $\mathcal{H}$-Chameleon outperforms Hmat.

The comparative study of the StarPU scheduling strategies in these figures reflects that, in general, the three variants deliver similar execution times. However, the strategies based on priorities provide higher performance, and the simple priority strategy turns to be the best option in most of the cases, except for the smaller dimensions. In the real double precision cases with $N = 10K$ and $N = 20K$, the priority scheduler does not provide the fastest solution, as the computational tasks are too small and the idle threads create a contention in the single global task queue of this scheduler.

6.4 Concluding remarks

We have proposed an extension of the Chameleon library that takes advantage of $\mathcal{H}$-Matrices and $\mathcal{H}$-Arithmetic to accelerate the execution time and reduce the memory footprint of the LU factorization. More precisely, our approach takes advantage of the sequential kernels in Hmat-oss to perform $\mathcal{H}$-Arithmetic, and the task-based approach of Chameleon to exploit parallelism. The original large matrix is split into a set of tiles, where each tile can be represented either as a dense or an $\mathcal{H}$-Matrix. Then, a runtime system, as StarPU does in our experiments, schedules the tasks on the system, following the approaches developed in Chameleon for the dense case.

We have conducted experiments on a multicore machine for a large real-life case, and our results have demonstrated that this approach is competitive with the proprietary Hmat library. Thus, it provides one of the first open-source library that is able to reach a good level of performance using $\mathcal{H}$-Matrices. A main asset of the approach is that it will directly benefit from all improvements on the runtime side, as it is now integrated in the Chameleon library.

Chapter 7

Last remarks

Contents of the chapter

7.1 Concluding remarks and main contributions

The main goal of this thesis was designing, implementing and evaluating implementations for $\mathcal{H}$-Arithmetic operations capable of being efficiently executed in multicore architectures by leveraging task-based parallelization strategies.

The lack of open source packages for solving $\mathcal{H}$-Arithmetic operations efficiently in multicore architectures, together with the increasing popularity of task-based parallel programming models to target this type of platforms, motivated us to analyse whether task-based parallel strategies could be an appropriate solution for the algorithms designed to operate with $\mathcal{H}$-Matrices.

The following list summarizes the main contributions and associated conclusions that derive from this dissertation:

- In the beginning, we implemented prototype versions of the $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky operations with the purpose of evaluating (in a simplified scenario) whether task-based parallelism was suitable for parallelising operations with $\mathcal{H}$-Matrices. From the design of the mentioned prototypes, we learnt that the packages to build, manipulate, and operate (in parallel) with $\mathcal{H}$-Matrices demand special considerations and requirements from three different perspectives: storage, performance, and parallel strategies. The main reason behind these necessities is the nested structure inherent to $\mathcal{H}$-Matrices which, in turn, naturally leads to recursive formulations of the algorithms to perform $\mathcal{H}$-Arithmetic operations.

- From that initial work, we were able to confirm that task-based parallelism is suitable not only for parallelising implementations of the $\mathcal{H}$-Arithmetic operations, but also that it overcomes the alternative classical parallel strategies (such as the utilization of multithreaded implementations of BLAS, or loop-parallelism) when operating with $\mathcal{H}$-Matrices.
- The next important contribution of this dissertation is the parallelization of the $\mathcal{H}$-LU provided in H2Lib. The main conclusion from that work is that an approach that performs a traditional annotation of dependencies supported by OpenMP and OmpSs presents some limitations which force us to employ barriers that constrain the parallel efficiency. In contrast, the OmpSs-2 programming model provides new features that allow us to achieve a fair parallel efficiency.
- The last contribution is the design and implementation of $\mathcal{H}$-Chameleon. The associated key conclusion is that it is possible to leverage the advantages of $\mathcal{H}$-Matrices (in terms of storage and computations savings) while *simplifying/stabilising* the construction of the associated matrix structure. By using Tile $\mathcal{H}$-Matrices, we were able to combine the functionalities in the Chameleon and Hmat-oss packages functionalities to create $\mathcal{H}$-Chameleon. This open-source library is competitive with the proprietary Hmat library in multicore architectures, providing a good parallel performance.

In the following subsections, we provide some additional remarks detailing these contributions and conclusions.

7.1.1 The first prototypes: $\mathcal{H}$-Matrices special requirements

As described in Chapter 4, the first objective of this thesis was to evaluate whether the task-parallel strategies were suitable for algorithms that operate with $\mathcal{H}$-Matrices. In order to discern that, we designed and implemented prototype $\mathcal{H}$-Matrices, in which we kept the nested structure of blocks that characterizes $\mathcal{H}$-Matrices partitionings, but avoided including low-rank blocks to simplify the design and implementation of the $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky operations. In order to reproduce the load imbalance between operations that involve low-rank blocks and those which deal with dense blocks, we included dense and null blocks in our prototype structure, and generated prototype $\mathcal{H}$-Matrices with different levels of *dispersion* (based on the amount of null blocks).

In the process of designing how to build, store, and update our prototype $\mathcal{H}$-Matrices, we discovered that the structure to store them introduces several requirements:

- The hierarchy of the matrix needs to be efficiently defined in such a way that it is easy and fast to find parent/descendant blocks (if any) of each block.
- Leaf (not partitioned) vs. non-leaf blocks require different information to be stored. While for the first ones, the dimension and entries are stored, the later ones need to have a list of pointers to each of the sub-blocks.
- As a consequence of the previous requirements, the prototype storage definition needed to include three different structures to respectively represent non-leaf blocks, null blocks, and dense blocks. Moreover, auxiliary structures (such as arrays to point to the different parent/descendant blocks) were also necessary to define the matrix hierarchy.

- Thanks to our simplified version of $\mathcal{H}$-Matrices, we stored all the entries in a pre-adjusted array. The adjustment performed a preprocessing of the data solving a *symbolic* LU to detect which null blocks would be filled in during the $\mathcal{H}$-LU operation, with the purpose of *making room* for the new values (this is, including all the needed zeros in the values array). This adjustment eases the parallelization, as all the data is stored in contiguous positions of memory. Nevertheless, when operating with pure $\mathcal{H}$-Matrices this cannot be done, and we realized that each leaf block would need to store its values independently from those of other blocks, occupying non-contiguous positions of memory.
- We needed to define our own storage ordering: the BDL format, which can be understood as a CMO storage ordering that takes into account the nested structure of blocks when ordering data in memory.

Besides the mentioned data layout requirements, the algorithms to solve the $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky also showed some issues to consider. Theoretically, they are equivalent to the well-known tile-based ones to compute the LU and Cholesky counterparts, and the only modifications that should be applied are simple adjustments to take into account the presence of different block types and mixed sizes. However, the prototype implementations of the $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky revealed some issues that increased the complexity of designing the algorithms:

- As there are different block sizes in the $\mathcal{H}$-Matrix and the partitioning is not regular (that is, not all the blocks are divided into the same number of levels), some of the leaf blocks need to be treated as if they were sub-partitioned when performing operations that involve smaller block sizes. As a consequence, each time an operation is performed, all the involved leaf blocks need to be partitioned into blocks whose size is equivalent to the smallest block size involved in the operation.
- An alternative to the additional partitioning, which may seem artificial, is re-formulating the storage layout. However, similar data representations are defined when utilizing pure $\mathcal{H}$-Matrices, and changing the storage layout would mean neglecting the hierarchy that characterizes $\mathcal{H}$-Matrices, which is much more counter-productive than performing some additional partitionings.

7.1.2 Task-parallelism offers a fair performance when applied to the $\mathcal{H}$-LU

We chose the OpenMP and OmpSs programming models to evaluate the suitability of task-parallelism in $\mathcal{H}$-Matrices scenarios. Even with the limitations of our prototype $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky, we were able to attain a fair performance that provided initial evidence of the benefits of this strategy in the context of $\mathcal{H}$-Matrices. Moreover, compared to the classical alternative approaches, such as loop-parallelism or the inclusion of Intel MKL multithreaded calls to BLAS routines, task-parallelism was more efficient.

A remarkable side effect of the *artificial* sub-partitionings mentioned in the previous subsection is the fact that finer grain tasks are identified, and a higher concurrency degree can be exposed when using task-based parallel approaches if the sub-operations are annotated as tasks, instead of doing so for the original operations defined by the algorithm. However, the fact that data regions of different sizes are annotated in the tasks dependencies implies some additional issues to consider:

- The Program Objects that are associated with each data portion when annotating dependencies cannot be utilized for different lengths. If the prototypes were not simulating such a simple scenario, this fact would have forced us to artificially partition even more leaf blocks, not only to the smallest block size in each specific operation, but to the smallest block size existing in the $\mathcal{H}$-Matrix.
- The recursive nature of the (prototype) $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky forces us to include barriers at the end of each level of the hierarchy to ensure the proper fulfillment of the data dependencies of nested tasks. This is a performance limitation that we detailed in Chapter 5 instead of Chapter 4, because with our prototypes we were pursuing to simply test task-parallelism, without going further in the optimization of the tasks' execution flow. Nevertheless, this issue was already observed in the design of the prototype $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky and is worth it mentioning as a *detected* issue.

7.1.3 Leveraging OmpSs-2 when parallelising the $\mathcal{H}$-LU in H2Lib

Equipped with the lessons learnt from the prototype evaluations, we opted for parallelising the $\mathcal{H}$-LU in the H2Lib package. The initial analysis of the library storage layout and algorithms revealed the following insights:

- The structures to store the $\mathcal{H}$-Matrices in H2Lib were similar to our prototype ones, with the main difference being the fact that matrix entries were stored in each of the structures defining leaf blocks, which means that the values are not in contiguous positions of memory unless they belong to the same not-partitioned block. Key structures such as arrays of pointers to reference sub-blocks of non-leaf blocks are also used in the library, in a similar way to what we did for our prototypes.
- The similarities with our prototype implementation made us infer that our task-parallelism conclusions would probably apply to the $\mathcal{H}$-LU in H2Lib.
- As low-rank blocks and the associated $\mathcal{H}$-Arithmetic operations are indeed part of H2Lib, compared to our prototypes, the algorithms' complexity is higher in terms of the variety of operations and combinations when performing the sub-operations that compute the $\mathcal{H}$-LU.
- The recursive algorithm nature still applies, and consequently the *basic* task-parallelism offered by OpenMP and OmpSs is limited to what can be concurrently executed at each level in the hierarchy, due to the need of including barriers to ensure the correct treatment of data dependencies.

The OmpSs-2 programming model gave us the possibility of leveraging novel task-parallelism features that allowed us to avoid the need of including barriers at the end of each recursion level. Particularly, we were able to execute tasks belonging to different levels thanks to the possibility of annotating weak dependencies and through them *informing* the runtime that the associated task does not effectively modify the data, but some of its sub-tasks do, together with the early release of tasks. With the mentioned features we increased the concurrency degree thanks to anticipating the execution of tasks that belong to different levels or different parent blocks, executing them as soon as the required data is ready.

The performance evaluation was conducted in the Marenostrum supercomputer at BSC, and we employed data arising from BEM in 1D, 2D and 3D scenarios. We reached a maximum speedup of $42\times$ when utilizing up to 48 cores.

7.1.4 $\mathcal{H}$-Chameleon: combining Tile $\mathcal{H}$-Matrices, Chameleon and Hmat-oss

We are satisfied with the performance offered by our parallel version of the $\mathcal{H}$-LU included in the H2Lib package. However, the features we need to be able to attain the mentioned efficiency are only available in the OmpSs-2 programming model, and the complexity of properly annotating all the task (weak/strong) dependencies and going through the pure $\mathcal{H}$-Matrices hierarchy is high. For this reason, we considered exploring an approach that simplified the data layout in such a way that conventional tile approaches could be directly applied and thus the parallelization process could be easier and faster, while still offering efficient results, competitive with the ones attained by pure $\mathcal{H}$-Matrices solutions. To this end, we designed Tile $\mathcal{H}$-Matrices, which in essence are regular tiled matrices in which each tile contains an $\mathcal{H}$-Matrix; this is, matrices that are partitioned into regular blocks (tiles) at the first level, and where each tile is then converted into an $\mathcal{H}$-Matrix.

From the perspective of the design and implementation, we decided to leverage the Chameleon and Hmat-oss packages to respectively tackle the tiled part of the new structure (this is, the first level of the partitioning) and solve the $\mathcal{H}$-Arithmetic operations. Chameleon is a dense linear algebra software that has shown good efficiency when solving tiled implementations of dense factorizations. Hmat-oss is the set of sequential routines in which Airbus' Hmat proprietary library is based, which is also highly efficient. $\mathcal{H}$-Chameleon was created thanks to the definition of new data structures, the inclusion of certain wrappers to communicate both libraries, and the definition of a new clustering algorithm to define the partitioning to build Tile $\mathcal{H}$-Matrices.

Our evaluation of $\mathcal{H}$-Chameleon offers the following conclusions:

- The compression and precision rates offered by Tile $\mathcal{H}$-Matrices are almost equivalent to the ones that characterise pure $\mathcal{H}$-Matrices.
- By leveraging the StarPU runtime to perform the parallel executions in multicore architectures, we are able to offer an efficient solver for problems arising from BEM. We included different priorities in the sub-operations that conform the $\mathcal{H}$-LU and tested different scheduling strategies in order to attain the best possible efficiency. The best strategy was the one that takes priorities into account.
- The experiments conducted in a multicore machine, for a large real-life case, demonstrated that $\mathcal{H}$-Chameleon is competitive with the proprietary Hmat library. Thus, it provides one of the first open-source library that is able to reach a good level of performance using $\mathcal{H}$-Matrices.
- Two non-despicable side effects of employing Chameleon and Hmat-oss libraries are that 1) $\mathcal{H}$-Chameleon will directly benefit from all improvements on the runtime side, as our library is now integrated in the Chameleon library; and 2) any optimizations made in the $\mathcal{H}$-Matrices kernels will also improve the performance of $\mathcal{H}$-Chameleon.

7.2 Related publications

In this section, we list the publications associated to the contributions of this dissertation, which include papers submitted to international conferences and indexed journals, all of them validated through peer-review processes. First, we enumerate the scientific contributions directly related to the dissertation. Afterwards, we list other works that have also been conducted and either have a relationship with linear algebra and task parallelism, or reflect educational research efforts. Together with the scientific research carried out to reach the objective of implementing efficient parallel algorithms for $\mathcal{H}$-Matrices, during the development of this thesis, 170 hours were taught in bachelor degrees in Engineering, and this is the reason why Computer Science educational research studies were also performed and are mentioned as indirectly related publications.

7.2.1 Directly related publications

Chapter 4. It all began with prototypes

The first prototype implemented in the context of this thesis was the one to solve the $\mathcal{H}$-LU, and afterwards we also implemented the prototype $\mathcal{H}$-Cholesky. As mentioned in Chapter 4, the purpose of these designs was not to create a whole new library to operate with $\mathcal{H}$-Matrices, but to test whether task parallelism is an appropriate option to tackle these special matrices. The three scientific contributions listed next evidence that this parallelization strategy was suitable and, in fact, more competitive than classical ones. Moreover, through these works we expose the difficulties that need to be addressed when parallelising algorithms that involve $\mathcal{H}$-Matrices.

Conference Proceedings [6]

Aliaga, José I., Carratalá-Sáez, Rocío, Kriemann, Ronald and Quintana-Ortí, Enrique S. Task-Parallel LU Factorization of Hierarchical Matrices using OmpSs. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPS)* (2017), pp. 1148–1157.

> In this paper we investigated the multithreaded parallelization of the prototype $\mathcal{H}$-LU described in Chapter 4, using the OpenMP and OmpSs task-parallel programming models. The focus of that study was on the adoption of an efficient storage layout for this type of matrices, and the analysis of the consequences that this decision exerts on the detection of task dependencies, the programming effort, and the performance of the solution. The performance evaluation showed that task-parallelism outperforms loop-parallelism, as well as parallel executions based on multithreaded calls to BLAS kernels.

Conference Proceedings [5]

Aliaga, José I., Barreda, Maria, Carratalá-Sáez, Rocío, Catalán, Sandra and Quintana-Ortí, Enrique S. A Unified Task-Parallel Approach for Dense, Hierarchical and Sparse Linear Systems. In *XXV Congreso de Ecuaciones Diferenciales y Aplicaciones + XV Congreso de Matemática Aplicada (CEDYA+CMA17)* (2017), pp. 152–159.

> This work reflected a unified approach to solve systems of linear equations, offering three different analysis that involve dense, hierarchical and sparse coefficient matrices of large dimension, respectively. In all cases, the task-parallelism intrinsic to the algorithms was exploited to reach an efficient execution on a multicore processor. OpenMP and OmpSs were the employed programming models, and the emphasis was put on describing the

task-strategy major advantage: adopting a parallelizing runtime system to decouple the numerical aspects of the method and application (left in the hands of the expert mathematicians, physicists or computational scientists), from the difficulties associated with high performance computing (more naturally addressed by computer scientists and engineers).

Aliaga, José I., Carratalá-Sáez, Rocío and Quintana-Ortí, Enrique S. Parallel Solution of Hierarchical Symmetric Positive Definite Linear Systems. In *Applied Mathematics and Nonlinear Sciences (AMNS)* (2017), Vol. 2(1), pp. 201–212. Journal [7]

In this paper we presented the prototype $\mathcal{H}$-Cholesky algorithm described in Chapter 4, and the associated parallelization employing the OpenMP and OmpSs programming models. Its performance efficiency was compared to that attained by alternative parallelization approaches that exploit either multithreaded calls to BLAS kernels or loop-parallelism via a runtime. As with the $\mathcal{H}$-LU, we observed that task-parallelism offered the best performance.

Chapter 5. Parallelizing the $\mathcal{H}$-LU in H2Lib

Thanks to the lessons learnt from the three previously-mentioned works, we could confirm that task parallelism was suitable for algorithms involving $\mathcal{H}$-Matrices, but some special considerations needed to be taken into account with respect to their recursive nature, as well as to the associated nested structure that requires special storage and treatment. As we did not aim to *reinvent the wheel*, we opted for parallelising the $\mathcal{H}$-LU provided by the H2Lib package, and that is the work reflected in the next publication.

Carratalá-Sáez, Rocío, Christophersen, Sven, Aliaga, José I., Beltran, Vicenç, Börm, Steffen and Quintana-Ortí, Enrique S. Exploiting Nested Task-Parallelism in the H-LU Factorization. In *Journal of Computational Science* (2019), Vol. 33, pp. 20–33. Journal [34]

This work described the efforts conducted to implement the task-based parallel version of the $\mathcal{H}$-LU algorithm implemented in the H2Lib package. It included an exhaustive description of the benefits that arise from using OmpSs-2 in contrast to OpenMP or OmpSs, which have already been described in Chapter 5. The possibility of annotating weak dependencies, together with the tasks early release, allowed us to reach a fair parallel performance which reached an speedup of $42x$ when testing our parallel $\mathcal{H}$-LU with problems arising from BEM, and using 48 cores of the Marenostrum supercomputer in the BSC.

Chapter 6. Moving further: $\mathcal{H}$-Chameleon, a parallel $\mathcal{H}$-library

High efficiency was reached when parallelising the $\mathcal{H}$-LU implementation in H2Lib. Nevertheless, this would have not been possible without leveraging several OmpSs-2 special features that are not available in other programming models. The $\mathcal{H}$-Matrices structure complexity is responsible of the issues that make the mentioned features necessary, and this is the reason why we decided to explore if we could design a custom version of the pure $\mathcal{H}$-Matrices in order to reduce the structure complexity. For this purpose we designed Tile $\mathcal{H}$-Matrices, with the aim of incorporating more regular patterns at the highest level of the hierarchy (in terms of the blocks size), while maintaining the storage and computational savings that characterise $\mathcal{H}$-Matrices. We implemented the $\mathcal{H}$-Chameleon package to operate with Tile $\mathcal{H}$-Matrices, whose details are described in the following publication.

Conference Proceedings [35] Carratalá-Sáez, Rocío, Faverge, Mathieu, Pichon, Grégoire, Sylvand, Guillaume and Quintana-Ortí, Enrique S. Tiled Algorithms for Efficient Task-Parallel $\mathcal{H}$-Matrix Solvers. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2020), pp. 757–766.

Lastly, this paper provided a description of the $\mathcal{H}$-Chameleon library, including the changes needed to call the Hmat-oss kernels (to operate with $\mathcal{H}$-Matrices) from the Chameleon functions, as well as the new partitioning algorithm designed to build Tile $\mathcal{H}$-Matrices, and the new data structures created to store them. As a result, efficient solvers for linear systems arising in BEM were created. One of the aspects to remark of this work is the capability of maintaining the $\mathcal{H}$-Matrix benefits (in terms of storage and computations savings) while being able to leverage classical tiled algorithms that allow to expose a higher concurrency degree. The performance evaluation demonstrated a fair efficiency, and hence, we presented the (at the moment) most efficient fully open-source software stack to solve dense compressible linear systems on shared memory architectures.

7.2.2 Indirectly related publications

Research collaborations

Thanks to the lessons learnt during the evaluation of the different task-based programming models and strategies, as well as the SVD knowledge gained through the analysis of how to operate with $\mathcal{H}$-Matrices, an indirectly-related-to-the-thesis-objective research collaboration was conducted resulting in the following publication.

Journal [112] Tomás, Andrés E., Rodríguez-Sánchez, Rafael, Catalán, Sandra, Carratalá-Sáez, Rocío and Quintana-Ortí, Enrique S. Dynamic look-ahead in the reduction to band form for the singular value decomposition. In *Parallel Computing* (2019), Vol. 81, pp. 22–31.

This paper showed the benefits from an alternative reduction to a intermediate by-product after the first stage in the two-stage algorithms for the SVD. In contrast with the conventional approach, which produces an upper triangular-band matrix, the proposed one consisted of a band matrix with the same upper and lower bandwidth. Thanks to it, a look-ahead strategy could be easily applied with minor constraints on the relationship between the algorithmic block size and the bandwidth, yielding a high-performance implementation on servers equipped with multicore technology and graphics processors.

Publications related to educational research

The 170 hours of teaching in Engineering degrees during the time in which this thesis was conducted, reflected there is a lack of High Performance Computing (HPC) related contents in the engineering syllabus. As a consequence, together with other researchers and professors, some non official courses were offered to the undergraduate students in order to *bridge the gap* between HPC and the future engineers. The following research publications were conducted thanks to the course success and the attendees feedback.

Carratalá-Sáez, Rocío, Catalán, Sandra and Iserte, Sergio. Teaching on Demand: an HPC Experience. In *2019 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)* (2017), pp. 32–41. Conference Proceedings [33]

The course "Build your own supercomputer with Raspberry Pi" was offered as a non-mandatory workshop with the purpose of introducing HPC to the bachelor students of Universitat Jaume I (UJI). The educational research interest did not only lay in the learning results, but also in the *personalized* experience offered to the students, thanks to which each of them could fulfill his/her curiosity about specific topics either presented and discussed in the class or not, independent of the other attendees. Two surveys, respectively filled by the students at the beginning and the end of the course, reflected that they acquired HPC knowledge while enjoying the course, especially the *hands-on* part in which they build a *supercomputer* with Raspberry Pi devices, and the *on-demand* section that allowed them to guide the learning process in a personalized way that kept their motivation alive.

Catalán, Sandra, Carratalá-Sáez, Rocío and Iserte, Sergio. Leveraging Teaching on Demand: Approaching HPC to Undergrads. Submitted to *Journal of Parallel and Distributed Computing* (2020). Journal

This work is an extension of the previous one [33]. Thanks to the second edition of the already described course, a deeper analysis of the learning results and process could be performed. Moreover, additional learning objectives were covered, and the insights and lessons learnt from the first experience were leveraged, so an enriched program was offered.

7.3 Open research lines

The work described in this dissertation evidences that the initial goal of the thesis has been fulfilled. Nevertheless, there are some issues that could be explored in order to extend certain aspects of the described contributions:

- In our work we tackle multicore architectures. There exist recent research efforts to execute $\mathcal{H}$-Matrices operations in distributed systems [74, 116]. In fact, we have already developed a distributed memory implementation of the $\mathcal{H}$-Chameleon library. However, the particularities of the nested $\mathcal{H}$-Matrix structure, together with the high degree of data dependencies that characterizes the $\mathcal{H}$-LU, severely constrain the efficiency that can be attained. The improvements that need to be done on this side to make the distributed memory implementation of the $\mathcal{H}$-Chameleon library competitive are part of future work. Possibly, the lessons learnt from this dissertation and also the ones extracted from the distributed memory implementation of the $\mathcal{H}$-Chameleon library (when available with a good performance) could be leveraged to consider similar strategies to make the pure $\mathcal{H}$-LU in H2Lib available for distributed memory platforms. This is an open research line that would focus on finding a strategy that nowadays is unclear: how to efficiently compute distributed memory operations that deal with $\mathcal{H}$-Matrices.

- With respect to our parallel implementation of the $\mathcal{H}$-LU in H2Lib, we observed that the efficiency we could achieve in 1D and 2D scenarios was not as high as the one we reached in 3D cases. Analysing the reasons behind these differences is interesting not only for improving the 1D and 2D performance, but also for learning how to better tackle 3D scenarios and problems arising from alternative applications.
- Regarding the evaluations, we based our tests in operating with data arising from BEM and we particularly considered the Laplace equation in 1D, 2D and 3D. Alternative contexts, such as, for example, BEM for the Helmholtz equation, could also be explored. Moreover, in the $\mathcal{H}$-Chameleon evaluation, we utilized data chosen from the surface of a cylinder, and other cylinder dimensions or different surfaces could expose new open research lines in the sense of how to sort and compress the data to build $\mathcal{H}$-Matrices.
- In relation to the employed programming models, OmpSs-2 and StarPU were the key in our work to attain a good parallel efficiency. Alternative programming models and strategies could also be explored to analyse if they present any feature that avoids the limitations that still apply to our developments, or somehow improve the performance.
- There exist works [30, 113] that leverage the GPUs in the system to improve the performance of $\mathcal{H}$-Matrices operations. Exploring the integration of GPU support in H2Lib and/or $\mathcal{H}$-Chameleon would also be interesting.

Capítulo 8

Conclusiones

Índice del capítulo

8.1 Conclusiones y contribuciones principales

El objetivo general de esta tesis era diseñar, implementar y evaluar implementaciones para operaciones pertenecientes a la Aritmética con $\mathcal{H}$-Matrices, capaces de ser ejecutadas en arquitecturas multinúcleo aprovechando estrategias de paralelismo basadas en tareas.

La falta de bibliotecas de código abierto para resolver eficientemente operaciones de la Aritmética con $\mathcal{H}$-Matrices en arquitecturas multinúcleo, junto con la creciente popularidad de los modelos de programación paralela basados en tareas enfocados a dichas plataformas, nos motivaron a analizar si las estrategias de paralelismo basado en tareas podían ser una solución apropiada para paralelizar algoritmos diseñados para operar con $\mathcal{H}$-Matrices.

La lista que sigue resume las contribuciones principales que se han obtenido gracias a esta disertación, así como las conclusiones asociadas a las mismas:

- Inicialmente implementamos prototipos de las descomposiciones $\mathcal{H}$-LU y $\mathcal{H}$-Cholesky para evaluar (en un escenario simplificado) si el paralelismo basado en tareas era apropiado para paralelizar operaciones con $\mathcal{H}$-Matrices. Mediante el diseño de los mencionados prototipos, aprendimos que los paquetes de software diseñados para construir, manipular y operar (en paralelo) con $\mathcal{H}$-Matrices requieren consideraciones y requisitos particulares en lo relativo a tres perspectivas: almacenamiento, rendimiento y estrategias de paralelismo. El motivo principal detrás de estas necesidades es la estructura anidada inherente a las $\mathcal{H}$-Matrices que, a su vez, conlleva de una forma natural el diseño de implementaciones recursivas de los algoritmos que realizan operaciones de la Aritmética con $\mathcal{H}$-Matrices.

- A partir de ese trabajo inicial, pudimos confirmar que el paralelismo basado en tareas no solamente es apropiado para las implementaciones de las operaciones de la Aritmética con $\mathcal{H}$-Matrices, sino que, además, supera el rendimiento que las estrategias clásicas de paralelismo (tales como el uso de implementaciones multihilo de BLAS, o el paralelismo a nivel de bucle) cuando se opera con $\mathcal{H}$-Matrices.

- La siguiente contribución importante es la paralelización de la factorización $\mathcal{H}$-LU proporcionada por la biblioteca H2Lib. La principal conclusión de dicho trabajo es que la anotación de dependencias de un modo tradicional, tal como se plantea con OpenMP y OmpSs, presenta algunas limitaciones que nos obligan a utilizar *barreras*, lo cual restringe la eficiencia paralela. Sin embargo, el modelo de programación OmpSs-2 tiene algunas funcionalidades o características nuevas que permiten alcanzar una eficiencia paralela mayor, combatiendo parte de dichas limitaciones.

- La última contribución se refiere al diseño e implementación de $\mathcal{H}$-Chameleon. La conclusión clave que se deriva de este trabajo es que es posible aprovechar conjuntamente las ventajas de utilizar $\mathcal{H}$-Matrices (tanto en términos de ahorro de almacenamiento como de cálculo) a la vez que se simplifica o estabiliza la construcción de la estructura de la matriz asociada. Utilizando Tile $\mathcal{H}$-Matrices, fuimos capaces de combinar las bibliotecas Chameleon y Hmat-oss para crear $\mathcal{H}$-Chameleon, la cual presenta resultados competitivos (es decir, una buena eficiencia) en arquitecturas multinúcleo, comparables a los de la librería privada Hmat.

En las subsecciones que siguen proporcionamos más detalles sobre las contribuciones y conclusiones mencionadas.

8.1.1 Los primeros prototipos: requisitos especiales de las $\mathcal{H}$-Matrices

Tal como se describe en el Capítulo 4, el primer objetivo de esta tesis fue evaluar si las estrategias de paralelismo basadas en tareas eran apropiadas para paralelizar aquellos algoritmos que operan con $\mathcal{H}$-Matrices. Con este objetivo, diseñamos e implementamos prototipos de $\mathcal{H}$-Matrices, en los que mantuvimos la estructura anidada de bloques característica de los particionados de las $\mathcal{H}$-Matrices, pero no incluimos bloques de rango bajo, con el fin de simplificar el diseño y la implementación. Con el fin de reproducir el desbalanceo de carga que típicamente existe entre las distintas operaciones que se realizan sobre las $\mathcal{H}$-Matrices, según si operan sobre bloques de rango bajo o densos, incluimos algunos bloques nulos en nuestra estructura, generando, gracias a esto, varias matices prototipo con distintos niveles de *dispersión* (basada en la cantidad de bloques nulos).

Durante el proceso de diseño de cómo construir, almacenar y actualizar nuestras $\mathcal{H}$-Matrices prototipo, descubrimos que la estructura necesaria para almacenarlas presenta los siguientes requisitos:

- La jerarquía de la matriz debe estar definida de un modo eficiente para que sea sencillo encontrar los bloques padres/descendientes (si los hay) de cada bloque.

- Los bloques hoja (es decir, los no particionados) requiren almacenar información diferente a los densos. Mientras que los primeros guardan su dimensión y valores, los segundos tienen que almacenar una lista de punteros relativos a cada uno de sus sub-bloques.

- Como consecuencia de los requisitos anteriores, para el almacenamiento del prototipo necesitamos definir tres estructuras diferentes con las que representar respectivamente a los bloques no hoja, a los nulos y a los densos. Adicionalmente, también fueron necesarias algunas estructuras auxiliares para definir la jerarquía de la matriz, tales como vectores de punteros para representar las relaciones entre los distintos bloques de la estructura.

- Gracias a haber simplificado nuestra estructura de $\mathcal{H}$-Matrices, almacenamos los valores de la matriz en un vector *pre-ajustado*. El ajuste realizado consiste en preprocesar los datos realizando una factorización simbólica (es decir, sin llegar a calcular las operaciones en sí) para detectar qué bloques nulos se rellenarán durante el cálculo la $\mathcal{H}$-LU, pasando a ser densos, con el objetivo de *hacer hueco* a los valores correspondientes, almacenando para ello tantos ceros como sean necesarios antes de iniciar el cálculo de dicha operación. Este ajuste facilita la paralelización, dado que permite que todos los datos estén almacenados en posiciones contiguas de memoria. No obstante, cuando se opera con $\mathcal{H}$-Matrices *puras*, esto no es posible hacerlo, lo cual hace necesario almacenar los valores de cada bloque hoja de un modo independiente respecto al resto, ocupando posiciones de memoria no contiguas.
- Además, necesitamos definir nuestro propio criterio de ordenación: el formato BDL, que puede entenderse como un almacenamiento en CMO que contempla la estructura anidada de bloques a la hora de ordenar los datos en la memoria.

Además de los requisitos mencionados con respecto a la representación de los datos, los algoritmos que resuelven la $\mathcal{H}$-LU y la $\mathcal{H}$-Cholesky también mostraron algunos aspectos a considerar. Teóricamente son equivalentes a sus homólogos para matrices densas a bloques que calculan dichas descomposiciones, salvo por el hecho de que ahora deben tenerse en cuenta diferentes tipos de bloques y dimensiones variadas. Sin embargo, las implementaciones prototipo de la $\mathcal{H}$-LU y $\mathcal{H}$-Cholesky revelaron algunos aspectos que aumentan la complejidad de los algoritmos mencionados:

- Dado que hay diferentes tamaños de bloque en las $\mathcal{H}$-Matrices y que los particionados no son regulares (es decir, no todos los bloques están subdivididos en la misma cantidad de niveles), cuando se opera con distintos tamaños de bloque, algunos de los bloques hoja necesitan ser tratados como si estuvieran sub-particionados. Como consecuencia, cada vez que se realiza una operación, todos los bloques sobre los que se opera deben particionarse tanto como sea necesario hasta alcanzar el menor tamaño de bloque con el que se opere en ese caso concreto.
- Una alternativa al particionado adicional descrito es reformular el almacenamiento de los datos, pero esto podría parecer artificial. De hecho, las representaciones de los datos utilizadas cuando se opera con $\mathcal{H}$-Matrices reales son similares a las descritas para nuestro prototipo, por lo que esta opción podría conllevar descuidar o rechazar la jerarquía que caracteriza a las $\mathcal{H}$-Matrices, lo cual consideramos mucho más contraproducente que realizar algunos particionados extra.

8.1.2 El paralelismo de tareas ofrece un buen rendimiento si se aplica a la $\mathcal{H}$-LU

Elegimos los modelos de programación OpenMP y OmpSs para evaluar la adecuación del paralelismo basado en tareas a escenarios en los que se opera con $\mathcal{H}$-Matrices. Pese a las limitaciones de nuestros prototipos de $\mathcal{H}$-LU y $\mathcal{H}$-Cholesky, logramos una eficiencia adecuada que evidenció los beneficios de aplicar esta estrategia en el mencionado contexto. Además, comparado con las alternativas clásicas tales como el paralelismo de bucles o la inclusión de las llamadas a las rutinas de BLAS de Intel MKL, el paralelismo basado en tareas era mucho más eficiente.

Una consecuencia destacable de los sub-particionados *artificiales* detallados en la subsección previa es el hecho de que se identifican tareas de grano más fino y, por tanto, se alcanza un mayor grado de concurrencia al usar enfoques basados en paralelismo de tareas cuando se anotan como tareas las suboperaciones, en lugar de las operaciones propias definidas por el algoritmo. Sin embargo, el hecho de que existan diferentes tamaños de bloque implica contemplar algunos aspectos adicionales:

- Los *Program Objects* asociados a cada porción de datos cuando se indican las dependencias no pueden utilizarse para diferentes tamaños. Si los prototipos no hubieran operado sobre escenarios tan simples como los descritos, este hecho nos habría obligado a particionar artificialmente todavía más los bloques hoja, no solamente hasta alcanzar el menor tamaño de bloque con el que se está operando en ese momento, sino hasta el menor tamaño de bloque existente en la $\mathcal{H}$-Matriz.
- La naturaleza recursiva de (los prototipos de) la $\mathcal{H}$-LU y $\mathcal{H}$-Cholesky nos obliga a incluir barreras al final de cada nivel de la jerarquía para asegurar que se cumplen correctamente las dependencias en las tareas anidadas. Esto limita el rendimiento, tal como se detalla en el Capítulo 5 pero no en el Capítulo 4, dado que, con nuestros prototipos, únicamente perseguíamos evaluar de un modo simple el paralelismo basado en tareas, sin llegar a optimizar el flujo de ejecución de las mismas. No obstante, este aspecto ya se observó en el diseño de los prototipos para la $\mathcal{H}$-LU y $\mathcal{H}$-Cholesky, por lo que merece la pena mencionarlo como *detectado* en este punto.

8.1.3 Aprovechando OmpSs-2 para paralelizar la $\mathcal{H}$-LU de H2Lib

Provistos de lo aprendido gracias a la evaluación de los prototipos, optamos por paralelizar la $\mathcal{H}$-LU del software H2Lib. El análisis inicial de las estructuras de datos para el almacenamiento de $\mathcal{H}$-Matrices en la librería reveló lo siguiente:

- Las estructuras diseñadas para almacenar $\mathcal{H}$-Matrices en H2Lib son similares a las utilizadas para nuestros prototipos, siendo la principal diferencia el hecho de que los valores de la matriz se almacenan en cada una de las estructuras definidas para los bloques hoja, lo cual significa que los valores no ocupan posiciones contiguas de memoria, salvo que pertenezcan al mismo bloque no particionado. Las estructuras clave tales como una lista de punteros para referenciar a los distintos sub-bloques de los bloques no hoja también se usan en la librería, de un modo similar a como hicimos con nuestros prototipos.
- Las similitudes con nuestra implementación prototipo nos llevaron a inferir que posiblemente nuestras conclusiones sobre el paralelismo basado en tareas también eran de aplicación en la $\mathcal{H}$-LU de H2Lib.
- Dado que los bloques de rango bajo y las operaciones de la Aritmética con $\mathcal{H}$-Matrices correspondientes sí se incluyen en H2Lib, comparado con nuestros prototipos, los algoritmos presentan una complejidad mayor en términos de la variedad de operaciones y combinaciones que puede darse cuando se llevan a cabo las operaciones que calculan la $\mathcal{H}$-LU.

- Se mantiene la naturaleza recursiva del algoritmo y, consecuentemente, el paralelismo de tareas *básico* que ofrecen OpenMP y OmpSs está limitado a aquello que puede ejecutarse en cada nivel de la jerarquía, debido a la necesidad de incluir barreras que garanticen el correcto cumplimiento de las dependencias de datos.

El modelo de programación OmpSs-2 nos brindó la posibilidad de aprovechar características novedosas que permiten evitar el uso de barreras en cada nivel de la recursión. Particularmente, pudimos ejecutar tareas pertenecientes a diferentes niveles gracias al *early release* y a la posibilidad de anotar dependencias *weak* y, su uso, nos permitió *informar* al runtime de que las tareas asociadas a las mismas no van a modificar los datos, sino que lo hará alguna de sus sub-tareas. Con dichas opciones, pudimos anticipar la ejecución de tareas pertenecientes a diferentes niveles o bloques *padre*, lográndose así mejorar el grado de concurrencia y ejecutar cada tarea tan pronto como se satisfacen sus dependencias.

La evaluación del rendimiento se llevó a cabo en el supercomputador Marenostrum del BSC, utilizando datos que provienen de BEM en 1D, 2D y 3D. Alcanzamos un *speedup* máximo de 42× usando hasta 48 núcleos.

8.1.4 $\mathcal{H}$-Chameleon: combinando Tile $\mathcal{H}$-Matrices, Chameleon y Hmat-oss

Estamos satisfechos con el rendimiento ofrecido por nuestra versión paralela de la $\mathcal{H}$-LU de la biblioteca H2Lib. Sin embargo, las funcionalidades que necesitamos para alcanzar la eficiencia mencionada solamente están disponibles en el modelo de programación OmpSs-2, y la complejidad asociada a anotar debidamente todas las dependencias (*weak/strong*) y recorrer la jerarquía de $\mathcal{H}$-Matrices puras es elevada. Por esta razón, consideramos explorar un enfoque que simplificara el almacenamiento de tal modo que las estrategias tradicionales para matrices a bloques se pudieran aplicar directamente y, así, el proceso de paralelización fuera más sencillo y rápido. Todo ello manteniendo una buena eficiencia, competitiva con la propia de otras soluciones que operan con $\mathcal{H}$-Matrices puras. Con este propósito diseñamos Tile $\mathcal{H}$-Matrices, que en esencia son matrices particionadas en bloques regulares, cada uno de los cuales contiene una $\mathcal{H}$-Matriz; es decir, matrices que están particionadas en bloques regulares (*tiles*) en el primer nivel, y donde cada bloque se convierte en una $\mathcal{H}$-Matriz.

En lo referente al diseño de la solución, decidimos aprovechar las librerías Chameleon y Hmat-oss para, respectivamente, abordar la parte de nuestra estructura a bloques (es decir, el primer nivel del particionado), y resolver las operaciones con $\mathcal{H}$-Matrices. Chameleon es un software para álgebra lineal densa que ha mostrado una alta eficiencia resolviendo problemas de aritmética densa en implementaciones a bloques. Por su parte, Hmat-oss está formado por el conjunto de rutinas secuenciales en las que se basa la biblioteca privada de Airbus Hmat, la cual es también altamente eficiente. $\mathcal{H}$-Chameleon se creó gracias a la definición de nuevas estructuras de datos, la inclusión de ciertos *wrappers* para comunicar ambas librerías y la definición de un nuevo algoritmo de *clustering* para definir el particionado en base al cual construir Tile $\mathcal{H}$-Matrices.

Nuestra evaluación de $\mathcal{H}$-Chameleon arroja las siguientes conclusiones:

- El ratio de compresión y precisión ofrecidos por las Tile $\mathcal{H}$-Matrices son prácticamente equivalentes a los propios de las $\mathcal{H}$-Matrices puras.
- Aprovechando el *runtime* de StarPU para llevar a cabo las ejecuciones paralelas en arquitecturas multinúcleo, pudimos ofrecer un resolutor eficiente para problemas que emergen de

BEM. Incluimos diferentes prioridades en las sub-operaciones que conforman la $\mathcal{H}$-LU y probamos diferentes modos de planificación para alcanzar la mejor eficiencia posible. La mejor estrategia fue la que tiene en cuenta las prioridades fijadas para cada tarea.

- Los experimentos llevados a cabo en una máquina multinúcleo, con casos reales, demostraron que $\mathcal{H}$-Chameleon es competitivo con respecto a la librería Hmat. Así, provee una de las primeras librerías de código abierto que es capaz de alcanzar una buena eficiencia usando $\mathcal{H}$-Matrices.
- Dos consecuencias destacables de utilizar las librerías Chameleon y Hmat-oss son que 1) $\mathcal{H}$-Chameleon se beneficiará directamente de todas las mejoras realizadas desde el punto de vista del *runtime*, dado que está integrada en la librería Chameleon; y 2) cualquier optimización llevada a cabo sobre los kernels para $\mathcal{H}$-Matrices también mejorará el rendimiento de $\mathcal{H}$-Chameleon.

8.2 Publicaciones relacionadas

En esta sección listamos las publicaciones asociadas a las contribuciones de esta disertación, las cuales incluyen artículos enviados a conferencias internacionales y revistas indexadas, siendo validadas mediante procesos de revisión por pares. Inicialmente enumeramos las contribuciones científicas directamente relacionadas con la tesis. A continuación, listamos otros trabajos que también se han realizado durante el desarrollo de la misma, si bien guardan una relación indirecta con las $\mathcal{H}$-Matrices, más centrada en aspectos de álgebra lineal y paralelismo de tareas en general, o bien reflejan resultados de investigación en educación. Junto con la investigación científica llevada a cabo con el objetivo de implementar algoritmos eficientes para operar con $\mathcal{H}$-Matrices, durante el desarrollo de la tesis se impartieron 170 horas en grados de ingeniería, y esta es la razón por la que se llevaron a cabo estudios relacionados con la investigación docente.

8.2.1 Publicaciones directamente relacionadas

Chapter 4. It all began with prototypes

(*Capítulo 4. Todo empezó con prototipos*)

El primer prototipo implementado en el contexto de esta tesis fue el que calculaba la $\mathcal{H}$-LU; posteriormente se implementó también el de la $\mathcal{H}$-Cholesky. Como se menciona en el Capítulo 4, el objetivo de estas implementaciones no era crear una nueva biblioteca para operar con $\mathcal{H}$-Matrices, sino evaluar si el paralelismo de tareas era apropiado para operar con este tipo de matrices tan especial. Las tres contribuciones científicas listadas a continuación evidencian que lo es y, de hecho, llega a ser mejor que otros enfoques clásicos. Además, mediante estos trabajos expusimos las dificultades que deben analizarse cuando se paralelizan algoritmos que operan con $\mathcal{H}$-Matrices.

Actas de conferencias [6]

Aliaga, José I., Carratalá-Sáez, Rocío, Kriemann, Ronald and Quintana-Ortí, Enrique S. Task-Parallel LU Factorization of Hierarchical Matrices using OmpSs. En *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPS)* (2017), pp. 1148–1157.

En este artículo investigamos la paralelización multinúcleo del prototipo de $\mathcal{H}$-LU descrito en el Capítulo 4, utilizando los modelos de programación paralela basados en tareas OpenMP y OmpSs. El estudio se centró en diseñar un modo de almacenamiento eficiente para este tipo de matrices, así como analizar las consecuencias que esta decisión implica respecto a la detección de dependencias de tareas, el esfuerzo de programación y el rendimiento de la solución. La evaluación del rendimiento mostró que el paralelismo de tareas es más eficiente que el de bucles o las llamadas a kernels multihilo de BLAS.

Aliaga, José I., Barreda, Maria, Carratalá-Sáez, Rocío, Catalán, Sandra and Quintana-Ortí, Enrique S. A Unified Task-Parallel Approach for Dense, Hierarchical and Sparse Linear Systems. En *XXV Congreso de Ecuaciones Diferenciales y Aplicaciones + XV Congreso de Matemática Aplicada (CEDYA+CMA17)* (2017), pp. 152–159. Actas de conferencias [5]

Este trabajo reflejó un enfoque unificado para resolver sistemas de ecuaciones lineales, ofreciendo tres análisis diferentes que respectivamente operan sobre matrices de coeficientes densas, jerárquicas y densas. El paralelismo de tareas intrínseco a los algoritmos se explotó en todos los casos para alcanzar una ejecución eficiente en procesadores multinúcleo. OpenMP y OmpSs fueron los modelos de programación utilizados, y el énfasis se puso en describir la mayor ventaja de las estrategias basadas en tareas: adoptar un *runtime* para la ejecución en paralelo de modo que se desacoplen los aspectos numéricos del método y de la aplicación (dejándolos en manos de los matemáticos expertos, físicos o científicos computacionales), de las dificultades asociadas con la computación de alto rendimiento (abordado de un modo más natural por científicos e ingenieros informáticos).

Aliaga, José I., Carratalá-Sáez, Rocío and Quintana-Ortí, Enrique S. Parallel Solution of Hierarchical Symmetric Positive Definite Linear Systems. En *Applied Mathematics and Nonlinear Sciences (AMNS)* (2017), Vol. 2(1), pp. 201–212. Revista [7]

En este artículo presentamos nuestro prototipo del algoritmo para la $\mathcal{H}$-Cholesky descrito en el Capítulo 4, así como la paralelización asociada utilizando los modelos de programación OpenMP y OmpSs. Su eficiencia se comparó con la obtenida con otros enfoques paralelos que explotan llamadas multihilo a kernels de BLAS o paralelismo de bucle vía un *runtime*. Del mismo modo que con la $\mathcal{H}$-LU, observamos que el paralelismo basado en tareas ofrece el mejor rendimiento.

Chapter 5. Parallelizing the $\mathcal{H}$-LU in H2Lib
(*Capítulo 5. Paralelizando la $\mathcal{H}$-LU de H2Lib*)

Gracias a lo aprendido en los trabajos previamente mencionados, pudimos confirmar que el paralelismo de tareas era apropiado para los algoritmos que operan sobre $\mathcal{H}$-Matrices, pero algunas consideraciones especiales debían tenerse en cuenta con respecto a su naturaleza recursiva, así como a la estructura anidada asociada, que requiere un almacenamiento y trato particular. Dado que nunca pretendimos *reinventar la rueda*, optamos por paralelizar la $\mathcal{H}$-LU proporcionada por el software H2Lib, y ese es justamente el trabajo que refleja la siguiente publicación.

Revista [34] Carratalá-Sáez, Rocío, Christophersen, Sven, Aliaga, José I., Beltran, Vicenç, Börm, Steffen and Quintana-Ortí, Enrique S. Exploiting Nested Task-Parallelism in the H-LU Factorization. En *Journal of Computational Science* (2019), Vol. 33, pp. 20–33.

En este trabajo describió se describieron los esfuerzos llevados a cabo para implementar una versión paralela basada en tareas del algoritmo para la $\mathcal{H}$-LU incluido en la biblioteca H2Lib. Comprende una exhaustiva descripción de los beneficios que surgen de utilizar OmpSs-2 frente a OpenMP u OmpSs, los cuales hemos descrito en el Capítulo 5. La posibilidad de anotar dependencias *weak*, junto con la opción *early release*, nos permitieron alcanzar un buen rendimiento paralelo que logró un *speedup* de 42× al evaluar nuestra $\mathcal{H}$-LU paralela con problemas que surgen de BEM, y utilizando hasta 48 núcleos del supercomputador Marenostrum del BSC.

Chapter 6. Moving further: $\mathcal{H}$-Chameleon, a parallel $\mathcal{H}$-library

(*Capitulo 6. Un paso más allá: $\mathcal{H}$-Chameleon, una $\mathcal{H}$-biblioteca paralela*)

Paralelizando la $\mathcal{H}$-LU de H2Lib logramos una alta eficiencia. Sin embargo, esto no habría sido posible sin las funcionalidades especiales presentes en OmpSs-2, las cuales no están disponibles en otros modelos de programación. La complejidad de la estructura de las $\mathcal{H}$-Matrices es la responsable de las dificultades que convierten dichas funcionalidades exclusivas en vitales, y es esa precisamente la razón por la que decidimos explorar si podíamos plantear un diseño de estructura alternativo al de las $\mathcal{H}$-Matrices *puras* que simplificara la complejidad. Con este objetivo diseñamos las Tile $\mathcal{H}$-Matrices, con el propósito de incorporar patrones más regulares en el nivel más alto de la jerarquía (en términos de tamaño de bloque), manteniendo a la vez el ahorro de almacenamiento y cómputo característico de las $\mathcal{H}$-Matrices. Implementamos $\mathcal{H}$-Chameleon para operar con Tile $\mathcal{H}$-Matrices, cuyos detalles se describen en la publicación que sigue.

Actas de conferencias [35] Carratalá-Sáez, Rocío, Faverge, Mathieu, Pichon, Grégoire, Sylvand, Guillaume and Quintana-Ortí, Enrique S. Tiled Algorithms for Efficient Task-Parallel $\mathcal{H}$-Matriz Solvers. En *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2020), pp. 757–766.

Por último, en este artículo se proporcionó una descripción de la biblioteca $\mathcal{H}$-Chameleon, incluyendo los cambios necesarios para *llamar* a los kernels de Hmat-oss (y así operar con $\mathcal{H}$-Matrices) desde las funciones disponibles en Chameleon, así como el algoritmo de particionado nuevo diseñado para construir las Tile $\mathcal{H}$-Matrices y las nuevas estructuras para almacenar los datos correspondientes. Como resultado, creamos un resolutor eficiente de sistemas lineales que surgen de BEM. Uno de los aspectos a destacar es que se mantuvieron los beneficios del uso de $\mathcal{H}$-Matrices (en términos tanto de almacenamiento como de cómputo), a la vez que se alcanzaba un alto grado de concurrencia gracias a aprovechar los algoritmos a bloques clásicos. La eficiencia mostrada en la evaluación fue alta y, de este modo, logramos presentar el software más eficiente (hasta el momento) para resolver sistemas lineales densos comprimibles en arquitecturas de memoria compartida.

8.2.2 Publicaciones indirectamente relacionadas

Colaboraciones en otras investigaciones

Gracias a lo aprendido durante la evaluación de los diferentes modelos y estrategias de programación basada en tareas, así como los conocimientos obtenidos sobre la SVD como consecuencia de analizar cómo operar con $\mathcal{H}$-Matrices, realicé una colaboración en una investigación indirectamente relacionada con mi tesis, la cual dio lugar a la publicación que sigue.

Tomás, Andrés E., Rodríguez-Sánchez, Rafael, Catalán, Sandra, Carratalá-Sáez, Rocío and Quintana-Ortí, Enrique S. Dynamic look-ahead in the reduction to band form for the singular value decomposition. En *Parallel Computing* (2019), Vol. 81, pp. 22–31. Revista [112]

> Este artículo ilustró los beneficios de una reducción alternativa a la basada en el subproducto intermedio tras el primer paso en el algoritmo de dos pasos para la SVD. Contrariamente a la propuesta convencional, la cual produce una matriz banda triangular superior, nuestra propuesta genera una matriz banda con el mismo ancho de banda superior e inferior. Gracias a esto, se pudo aplicar una estrategia *look-ahead* con pequeñas restricciones con respecto a la relación entre el tamaño de bloque y el ancho de banda, lográndose una implementación de altas prestaciones en servidores equipados con tecnología multinúcleo y procesadores gráficos.

Publicaciones relacionadas con la investigación docente

Las 170 horas de docencia en diferentes grados de ingeniería durante la realización de la tesis, reflejaron la falta de contenido relacionado con la Computación de Altas Prestaciones (CAP, el inglés HPC) en los currículos de ingeniería. Como consecuencia de ello, ofrecí junto con otros investigadores y profesores algunos cursos no oficiales para estudiantes de grado, con el fin de salvar la brecha existente entre el CAP y los futuros ingenieros. Las publicaciones que siguen se realizaron gracias a los cursos mencionados y la retroalimentación proporcionada por los estudiantes que asistieron.

Carratalá-Sáez, Rocío, Catalán, Sandra and Iserte, Sergio. Teaching on Demand: an HPC Experience. En *2019 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)* (2017), pp. 32–41. Actas de conferencias [33]

> El curso “Construye tu propio supercomputador con Rasperry Pi” se ofreció como un taller no obligatorio, con el objetivo de introducir a los alumnos de grado de la Universitat Jaume I (UJI) en el CAP. El interés a nivel de investigación docente reside no solamente en los resultados de aprendizaje, sino también en la experiencia de aprendizaje *personalizado* innovador ofrecido a los estudiantes, gracias a la cual cada uno de ellos pudo satisfacer su curiosidad con respecto a aquellos temas específicos (presentados y comentados durante las sesiones del curso o no), con independencia del resto de alumnos. Las dos encuestas rellenadas por los estudiantes al inicio y fin del taller, respectivamente, reflejaron que los alumnos adquirieron conocimientos sobre CAP a la vez que disfrutaban de la experiencia, y especialmente de la parte práctica en la que construyeron el *supercomputador* con Rasperry Pi, así como de la sección *bajo demanda*,

la cual les permitió guiar ellos mismos el aprendizaje de un modo personalizado según aquello que les motivara en mayor medida.

Revista Catalán, Sandra, Carratalá-Sáez, Rocío and Iserte, Sergio. Leveraging Teaching on Demand: Approaching HPC to Undergrads. Enviado a *Journal of Parallel and Distributed Computing* (2020).

Este trabajo es una extensión del anterior [33]. Gracias a la realización de una segunda edición del curso, pudimos llevar a cabo un análisis más profundo de los resultados de aprendizaje, así como del desarrollo del taller. Además, se cubrieron objetivos de aprendizaje adicionales, de modo que las conclusiones y lecciones aprendidas de la primera edición sirvieron para enriquecer el programa ofrecido.

8.3 Líneas de investigación abiertas

El trabajo descrito en esta disertación evidencia que el objetivo planteado al inicio de la tesis se ha logrado. No obstante, existen algunos aspectos que pueden explorarse para mejorar o extender ciertos aspectos de las contribuciones descritas:

- En nuestro trabajo abordamos ejecuciones paralelas en arquitecturas multinúcleo. Hay estudios recientes que tratan de ejecutar operaciones sobre $\mathcal{H}$-Matrices en sistemas distribuidos [74, 116]. De hecho, nosotros hemos desarrollado una implementación de la biblioteca $\mathcal{H}$-Chameleon para memoria distribuida. Sin embargo, las particularidades propias de la estructura anidada que caracteriza a las $\mathcal{H}$-Matrices, junto con el alto grado de dependencias de datos entre las sub-operaciones de la $\mathcal{H}$-LU, constriñen severamente la eficiencia que puede alcanzarse. Las mejoras que deben aplicarse sobre la versión para memoria distribuida de $\mathcal{H}$-Chameleon forman parte del trabajo futuro. Posiblemente, combinando lo aprendido en esta disertación y una implementación eficiente en los sistemas mencionados, podría aprovecharse también para considerar estrategias similares con las que diseñar una implementación para memoria distribuida de la $\mathcal{H}$-LU de H2Lib. Esta es una línea de investigación abierta que se centraría en encontrar algo que a día de hoy es incierto: cómo calcular operaciones de $\mathcal{H}$-Matrices de un modo eficiente en sistemas distribuidos.
- En referencia a las implementaciones paralelas de la $\mathcal{H}$-LU en H2Lib, observamos que la eficiencia que pudimos alcanzar en 1D y 2D no es tan buena como la que se consigue en los casos 3D. Con el objetivo de mejorar el rendimiento en los casos mencionados, podrían analizarse las razones que se esconden tras esta diferencia, lo cual posiblemente también ayudaría a operar mejor sobre datos procedentes de entornos 3D e incluso sobre problemas pertenecientes a aplicaciones diferentes.
- Con respecto a las evaluaciones, nosotros siempre hemos basado los tests en operar con datos procedentes de BEM y, particularmente, hemos considerado la ecuación de Laplace en 1D, 2D y 3D. Podrían explorarse contextos alternativos, como por ejemplo aquellos basados en la ecuación de Helmotz en BEM. Además, en la evaluación de $\mathcal{H}$-Chameleon utilizamos datos procedentes de la superficie de un cilindro cuyas dimensiones podrían modificarse (e incluso el propio cilindro podría sustituirse por otro cuerpo) para explorar el impacto de dichos cambios sobre la manera en la que se comprimen los datos para construir las $\mathcal{H}$-Matrices.

- En relación a los modelos de programación empleados, OmpSs-2 y StarPU fueron la clave para alcanzar una buena eficiencia paralela. Otros modelos de programación y estrategias alternativas podrían explorarse con el fin de analizar si presentan alguna característica que aborde las limitaciones que todavía se mantienen en nuestros desarrollos, o si, de algún modo, mejoran el rendimiento de los mismos.
- Existen trabajos [30, 113] que exploran el uso de GPUs para mejorar el rendimientos de algunas operaciones con $\mathcal{H}$-Matrices. Analizar la integración de soporte para GPUs en H2Lib y/o $\mathcal{H}$-Chameleon también podría ser una línea de investigación futura interesante.

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84.

[2] Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S., and Tomov, S. Faster, Cheaper, Better, a Hybridization Methodology to Develop Linear Algebra Software for GPUs. Research report, CEA/DAM; Total E&P; Université de Bordeaux, 2010.

[3] Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., and Thibault, S. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. Research report, CEA/DAM ; Total E&P ; Université de Bordeaux, October 2017.

[4] Aliaga, J. I., Badia, R. M., Barreda, M., Bollhöfer, M., and Quintana-Ortí, E. S. Leveraging task-parallelism with OmpSs in ILUPACK's preconditioned CG method. In *26th Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD 2014)* (2014), pp. 262–269.

[5] Aliaga, J. I., Barreda, M., Carratalá-Sáez, R., Catalán, S., and Quintana-Ortí, E. S. A unified task-parallel approach for dense, hierarchical and sparse linear systems. In *XXV Congreso de Ecuaciones Diferenciales y Aplicaciones + XV Congreso de Matemática Aplicada (CEDYA+CMA17)* (2017), pp. 152–159.

[6] Aliaga, J. I., Carratalá-Sáez, R., Kriemann, R., and Quintana-Ortí, E. S. Task-Parallel LU Factorization of Hierarchical Matrices Using OmpSs. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2017), pp. 1148–1157.

[7] Aliaga, J. I., Carratalá-Sáez, R., and Quintana-Ortí, E. S. Parallel solution of hierarchical symmetric positive definite linear systems. *Applied Mathematics and Nonlinear Sciences 2*, 1 (2017), 201–212.

[8] Ambikasaran, S., Li, J. Y., Kitanidis, P. K., and Darve, E. Large-scale stochastic linear inversion using hierarchical matrices. *Computational Geosciences 17*, 6 (2013), 913–927.

[9] Ambikasaran, S., Singh, K. R., and Sankaran, S. S. Hodlrlib: A library for hierarchical matrices. *Journal of Open Source Software 4*, 34 (2019), 1167.

[10] Amestoy, P., Ashcraft, C., Boiteau, O., Buttari, A., L'Excellent, J. Y., and Weisbecker, C. Improving multifrontal methods by means of block low-rank representations. *SIAM Journal on Scientific Computing 37*, 3 (2015), A1451–A1474.

[11] Amestoy, P., Buttari, A., L'Excellent, J., and Mary, T. Bridging the gap between flat and hierarchical low-rank matrix formats: the multilevel BLR format. vol. 41, pp. A1414–A1442.

[12] Aminfar, A., Ambikasaran, S., and Darve, E. A fast block low-rank dense solver with applications to finite-element matrices. *Journal of Computational Physics 304* (2016), 170–188.

[13] Another software library on Hierarchical Matrices for Elliptic Differential equations (AHMED). `https://github.com/xantares/ahmed`. Accessed: February 2021.

[14] NASA Apollo Lunar Surface Journal - "Gimbal Angles, Gimbal Lock, and a Fourth Gimbal for Christmas". `https://history.nasa.gov/alsj/gimbals.html`. Accessed: February 2021.

[15] NASA Apollo 11 Mission Overview website. `https://www.nasa.gov/mission_pages/apollo/missions/apollo11.html`. Accessed: February 2021.

[16] NASA History website - 50 Years Ago: "Houston, We've Had a Problem". `https://www.nasa.gov/feature/50-years-ago-houston-we-ve-had-a-problem`. Accessed: February 2021.

[17] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience 23*, 2 (2011), 187–198.

[18] Avetisov, V. A., Bikulov, A. K., and Nechaev, S. K. Random hierarchical matrices: spectral properties and relation to polymers on disordered trees. *Journal of Physics A: Mathematical and Theoretical 42*, 7 (jan 2009), 075001.

[19] Avetisov, V. A., Bikulov, A. K., Vasilyev, O. A., Nechaev, S. K., and Chertovich, A. V. Some physical applications of random hierarchical matrices. *Journal of Experimental and Theoretical Physics 109*, 3 (2009), 485–504.

[20] Badia, R. M., Herrero, J. R., Labarta, J., Pérez, J. M., Quintana-Ortí, E. S., and Quintana-Ortí, G. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience 21* (2009), 2438–2456.

[21] Bai, Z., Hiraishi, T., Nakashima, H., Ida, A., and Yasugi, M. Parallelization of matrix partitioning in construction of hierarchical matrices using task parallel languages. *Journal of Information Processing 27* (2019), 840–851.

[22] Barcelona Supercomputing Center. `https://www.bsc.es/`. Accessed: February 2021.

[23] Bathe, K. *Finite Element Procedures.* No. pt. 2 in Finite Element Procedures. Prentice Hall, 1996.

[24] Benedetti, I., Aliabadi, M., and Davì, G. A fast 3d dual boundary element method based on hierarchical matrices. *International Journal of Solids and Structures 45*, 7 (2008), 2355–2376.

[25] Benner, P., and Mach, T. Locally optimal block preconditioned conjugate gradient method for hierarchical matrices. *PAMM 11*, 1 (2011), 741–742.

[26] Benner, P., and Mach, T. The preconditioned inverse iteration for hierarchical matrices. *Numerical Linear Algebra with Applications 20*, 1 (2013), 150–166.

[27] Benner, P., and Mach, T. The LR Cholesky algorithm for symmetric hierarchical matrices. *Linear Algebra and its Applications 439* (08 2013), 1150–1166.

[28] Berendorf, M. *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems.* Lecture Notes in Computational Science and Engineering (LNCSE), Springer-Verlag, 2008.

[29] Basic Linear Algebra Subprograms (BLAS). `http://www.netlib.org/blas/`. Accessed: February 2021.

[30] Boukaram, W., Turkiyyah, G., and Keyes, D. Hierarchical matrix operations on gpus: Matrix-vector multiplication and compression. *ACM Trans. Math. Softw. 45*, 1 (Feb. 2019).

[31] Boyd, S., and Vandenberghe, L. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares.* Cambridge University Press, 2018.

[32] Butenhof, D. R. *Programming with POSIX Threads.* Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[33] Carratalá-Sáez, R., Catalán, S., and Iserte, S. Teaching on demand: an hpc experience. In *Proceedings of the 2019 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)* (2019), pp. 32–41.

[34] Carratalá-Sáez, R., Christophersen, S., Aliaga, J. I., Beltran, V., Börm, S., and Quintana-Ortí, E. S. Exploiting nested task-parallelism in the $\mathcal{H}$-LU factorization. *Journal of Computational Science 33* (2019), 20–33.

[35] Carratalá-Sáez, R., Faverge, M., Pichon, G., Sylvand, G., and Quintana-Ortí, E. S. Tiled algorithms for efficient task-parallel $\mathcal{H}$-Matrix solvers. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2020), pp. 757–766.

[36] Casenave, F. *Combler l'écart entre $\mathcal{H}$-Matrices et méthodes directes creuses pour la résolution de systèmes linéaires de grandes tailles.* Ph.D. Dissertation, Université Paris-Est, 2013.

[37] Casenave, F. *Méthodes de réduction de modèles appliquées à des problèmes d'aéroacoustique résolus par équations intégrales.* Ph.D. Dissertation, Université de Bordeaux, 2019.

[38] Chameleon software home page. `https://solverstack.gitlabpages.inria.fr/chameleon/`. Accessed: February 2021.

[39] Chandrasekaran, S., Gu, M., and Pals, T. A fast ulv decomposition solver for hierarchically semiseparable representations. *SIAM Journal on Matrix Analysis and Applications 28* (2006), 603–622.

[40] Chávez, G., Turkiyyah, G., Zampini, S., Ltaief, H., and Keyes, D. Accelerated cyclic reduction: A distributed-memory fast solver for structured linear systems. *Parallel Computing 74* (2018), 65 – 83. Parallel Matrix Algorithms and Applications (PMAA'16).

[41] Chen, Y., Davis, T. A., Hager, W. W., and Rajamanickam, S. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Trans. Math. Softw. 35*, 3 (Oct. 2008).

[42] Cheng, H., Gimbutas, Z., Martinsson, P. G., and Rokhlin, V. On the compression of low rank matrices. *SIAM Journal on Scientific Computing 26*, 4 (2005), 1389–1404.

[43] Demmel, J. W. *Applied Numerical Linear Algebra.* Society for Industrial and Applied Mathematics, 1997.

[44] Fan, Y., Lin, L., Ying, L., and Zepeda-Nú nez, L. A multiscale neural network based on hierarchical matrices. *Multiscale Modeling & Simulation 17*, 4 (2019), 1189–1213.

[45] Frigo, M., Leiserson, C. E., and Randall, K. H. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not. 33*, 5 (May 1998), 212–223.

[46] Fast User/Kernel Tracing (FxT) software. `http://savannah.nongnu.org/projects/fkt`. Accessed: February 2021.

[47] Geoga, C. J., Anitescu, M., and Stein, M. L. Scalable gaussian process computations using hierarchical matrices. *Journal of Computational and Graphical Statistics 29*, 2 (2020), 227–237.

[48] Ghysels, P., Li, X. S., Rouet, F.-H., Williams, S., and Napov, A. An efficient multicore implementation of a novel hss-structured multifrontal solver using randomized sampling. *SIAM Journal on Scientific Computing 38*, 5 (2016), S358–S384.

[49] Gillman, A. *Fast direct solvers for elliptic partial differential equations.* Ph.D. Dissertation, University of Colorado, 2011.

[50] Gillman, A., Young, P., and Martinsson, P. G. A direct solver with o(n) complexity for integral equations on one-dimensional domains. *Frontiers of Mathematics in China 7*, 2 (2012), 217–247.

[51] Golub, G. H., and Van Loan, C. F. *Matrix Computations (3rd Ed.).* Johns Hopkins University Press, USA, 1996.

[52] Gómez-Iglesias, A., Cheng, F., Huan, L., Liu, H., Liu, S., and Rosales, C. Benchmarking the Intel Xeon Platinum 8160 processor. Tech. Rep. TR-17-01, Texas Advanced Computing Center, 2017. Available at `https://repositories.lib.utexas.edu/handle/2152/61472`.

[53] Grasedyck, L., and Hackbusch, W. Construction and arithmetics of $\mathcal{H}$-Matrices. *Computing 70*, 4 (2003), 295–334.

[54] H2lib Package. `http://www.h2lib.org/`. Accessed: February 2021.

[55] Hackbusch, W. A sparse matrix arithmetic based on hmatrices. part i: Introduction to $\mathcal{H}$-Matrices. *Computing 62*, 2 (1999), 89–108.

[56] Hackbusch, W. *Hierarchical Matrices: Algorithms and Analysis*, vol. 49. 12 2015.

[57] Hackbusch, W., and Boerm, S. H2-matrix approximation of integral operators by interpolation. *Applied Numerical Mathematics 43*, 1 (2002), 129 – 143. 19th Dundee Biennial Conference on Numerical Analysis.

[58] Hackbusch, W., Khoromskij, B., and Kriemann, R. Hierarchical matrices based on a weak admissibility criterion. *Computing 73* (2004), 207–243.

[59] Hackbusch, W., Khoromskij, B., and Sauter, S. *On $\mathcal{H}^2$-Matrices.* 08 2000, pp. 9–29.

[60] HiCMA. https://github.com/ecrc/hicma. Accessed: February 2021.

[61] High-Performance $\mathcal{H}^2$ Matrix Package (H2Pack). https://www.cc.gatech.edu/~echow/pubs/h2pack.pdf. Accessed: February 2021.

[62] Hiraishi, T., Yasugi, M., Umatani, S., and Yuasa, T. Backtracking-based load balancing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2009), PPoPP '09, Association for Computing Machinery, p. 55–64.

[63] HLib Package. http://www.hlib.org/. Accessed: February 2021.

[64] HMAT-Oss. https://github.com/jeromerobert/hmat-oss. Accessed: February 2021.

[65] Hoag, D. Apollo guidance and navigation considerations of apollo imu gimbal lock. *NASA Apollo Lunar Surface Journal, MIT Instrumentation Laboratory Document E-1344* (1963).

[66] Hoque, R., Herault, T., Bosilca, G., and Dongarra, J. Dynamic task discovery in parsec: A data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems* (2017), pp. 1–8.

[67] Hu, X., Rodrigo, C., and Gaspar, F. J. Using hierarchical matrices in the solution of the time-fractional heat equation by multigrid waveform relaxation. *Journal of Computational Physics 416* (2020), 109540.

[68] Ida, A. Lattice $\mathcal{H}$-Matrices on distributed-memory systems. In *Proceedings of the International Parallel and Distributed Processing Symposium* (Rio de Janeiro, Brasil, May 2018), pp. 389–398.

[69] Ida, A., Iwashita, T., Mifune, T., and Takahashi, Y. Parallel hierarchical matrices with adaptive cross approximation on symmetric multiprocessing clusters. *Journal of Information Processing 22*, 4 (2014), 642–650.

[70] Ida, A., Iwashita, T., Mifune, T., and Takahashi, Y. Variable preconditioning of krylov subspace methods for hierarchical matrices with adaptive cross approximation. *IEEE Transactions on Magnetics 52*, 3 (2016), 1–4.

[71] IEEE 754 Standard for floating-point arithmetic. https://standards.ieee.org/standard/754-2019.html. Accessed: February 2021.

[72] Katsikadelis, J. T. *Boundary Elements Theory and Applications.* Elsevier, 2002.

[73] Kazempour, M., and Gürel, L. Solution of low-frequency electromagnetics problems using hierarchical matrices. In *CEM'13 Computational Electromagnetics International Workshop* (2013), pp. 9–10.

[74] Keyes, D., Ltaief, H., and Turkiyyah, G. Hierarchical algorithms on hierarchical architectures. *Philosophical Transactions of the royal society A 378* (2020).

[75] Khoromskij, B. N., Litvinenko, A., and Matthies, H. G. Application of hierarchical matrices for computing the karhunen–loève expansion. *Computing 84*, 1 (2009), 49–67.

[76] Knuth, D. E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, third ed. Addison-Wesley, Reading, Mass., 1997.

[77] Koch, O., Ede, C., Jordan, G., and Scrinzi, A. Hierarchical matrices in computations of electron dynamics. *J. Sci. Comput. 42*, 3 (March 2010), 447–455.

[78] Kriemann, R. Parallel $\mathcal{H}$-Matrix arithmetics on shared memory systems. *Computing 74*, 3 (2005), 273–297.

[79] Kriemann, R. $\mathcal{H}$-LU factorization on many-core systems. *Computing and Visualization in Science 16* (06 2013), 105–117.

[80] Linear Algebra PACKage (LAPACK). `http://www.netlib.org/lapack/`. Accessed: February 2021.

[81] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw. 5*, 3 (Sept. 1979), 308–323.

[82] Le Borne, S. Hierarchical matrices for convection-dominated problems. In *Domain Decomposition Methods in Science and Engineering* (Berlin, Heidelberg, 2005), T. J. Barth, M. Griebel, D. E. Keyes, R. M. Nieminen, D. Roose, T. Schlick, R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O. Widlund, and J. Xu, Eds., Springer Berlin Heidelberg, pp. 631–638.

[83] Lehmann, L., and Rüberg, T. Application of hierarchical matrices to the simulation of wave propagation in fluids. *Communications in Numerical Methods in Engineering 22*, 5 (2006), 489–503.

[84] Litvinenko, A., Sun, Y., Genton, M. G., and Keyes, D. E. Likelihood approximation with hierarchical matrices for large spatial datasets. *Computational Statistics & Data Analysis 137* (2019), 115–132.

[85] Liu, X., Wu, H., and Jiang, W. A boundary element method based on the hierarchical matrices and multipole expansion theory for acoustic problems. *International Journal of Computational Methods 15*, 03 (2018), 1850009.

[86] Lizé, B. *Résolution directe rapide pour les éléments finis de frontière en électromagnétisme et acoustique: $\mathcal{H}$-Matrices. Parallélisme et applications industrielles.* Ph.D. Dissertation, École doctorale Galilée, Université Sorbonne Paris Nord and Airbus, 2014.

[87] LoRaSp. `https://bitbucket.org/hadip/lorasp`. Accessed: February 2021.

[88] Marenostrum Supercomputer from the Barcelona Supercomputing Center (BSC). `https://www.bsc.es/marenostrum/marenostrum`. Accessed: February 2021.

[89] Mary, T. *Block Low-Rank multifrontal solvers: complexity, performance, and scalability.* Ph.D. Dissertation, Université de Toulouse, 2017.

[90] Massei, S., Robol, L., and Kressner, D. Hm-toolbox: MATLAB software for HODLR and HSS matrices. *SIAM J. Sci. Comput. 42*, 2 (2020), c43–c68.

[91] $\mathcal{H}$-Lib^{pro} Library. `https://www.hlibpro.com`. Accessed: February 2021.

[92] Mercurium compiler. `https://pm.bsc.es/mcxx`. Accessed: February 2021.

[93] Intel Math Kernel Library (MKL). `https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html`. Accessed: February 2021.

[94] Moore, G. Cramming more components onto integrated circuits. *Electronics 38* (04 1965).

[95] Moore, G. Progress in digital integrated electronics [technical literaiture, copyright 1975 IEEE. reprinted, with permission. technical digest. international electron devices meeting, IEEE, 1975, pp. 11-13.]. *Solid-State Circuits Newsletter, IEEE 20* (10 2006), 36 – 37.

[96] Nanos++ runtime. `https://pm.bsc.es/nanox`. Accessed: February 2021.

[97] Ohtani, M., Hirahara, K., Takahashi, Y., Hori, T., Hyodo, M., Nakashima, H., and Iwashita, T. Fast computation of quasi-dynamic earthquake cycle simulation with hierarchical matrices. *Procedia Computer Science 4* (2011), 1456–1465. Proceedings of the International Conference on Computational Science, ICCS 2011.

[98] OmpSs programming model. `https://pm.bsc.es/ompss`. Accessed: February 2021.

[99] OpenMP programming model. `https://www.openmp.org/`. Accessed: February 2021.

[100] Paraver performance analysis tool. `https://tools.bsc.es/paraver`. Accessed: February 2021.

[101] Perez, J. M., Beltran, V., Labarta, J., and Ayguadé, E. Improving the integration of task nesting and dependencies in OpenMP. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2017), pp. 809–818.

[102] Pichon, G. *On the use of low-rank arithmetic to reduce the complexity of parallel sparse linear solvers based on direct factorization techniques.* Ph.D. Dissertation, Université de Bordeaux, 2019.

[103] Pichon, G., Darve, E., Faverge, M., Ramet, P., and Roman, J. Sparse supernodal solver using block low-rank compression: Design, performance and analysis. *Journal of Computational Science 27* (2018), 255–270.

[104] PlaFRIM home page. `https://www.plafrim.fr/en/home/`. Accessed: February 2021.

[105] Reinders, J. *Intel Threading Building Blocks*, first ed. O'Reilly &amp; Associates, Inc., USA, 2007.

[106] Rodejohann, W. Hierarchical matrices in the see-saw mechanism, large neutrino mixing and leptogenesis. *The European Physical Journal C - Particles and Fields 32*, 2 (2004), 235–234.

[107] Saibaba, A. K., Miller, E. L., and Kitanidis, P. K. Fast kalman filter using hierarchical matrices and a low-rank perturbative approach. *Inverse Problems 31*, 1 (jan 2015), 015009.

[108] Saibaba, A.K., Ambikasaran, S., Yue Li, J., Kitanidis, P.K., and Darve, E.F. Application of hierarchical matrices to linear inverse problems in geostatistics. *Oil Gas Sci. Technol. - Rev. IFP Energies nouvelles 67*, 5 (2012), 857–875.

[109] Sergent, M. *Passage à l'échelle d'un support d'exécution à base de tâches pour l'algèbre linéaire dense.* Ph.D. Dissertation, Université de Bordeaux, 2016.

[110] StarPU Runtime System home page. `https://starpu.gitlabpages.inria.fr/`. Accessed: February 2021.

[111] Test_FEMBEM library github repository. `https://gitlab.inria.fr/hiepacs/papers/hmat/hmat-comparison/`. Accessed: February 2021.

[112] Tomás, A. E., Rodríguez-Sánchez, R., Catalán, S., Carratalá-Sáez, R., and Quintana-Ortí, E. S. Dynamic look-ahead in the reduction to band form for the singular value decomposition. *Parallel Computing 81* (2019), 22–31.

[113] Vater, K. *A GPU-Accelerated Galerkin BEM Based on Hierarchical Matrices for the Solution of Elliptic Partial Differential Equation Problems.* Ph.D. Dissertation, University College London, 2017.

[114] Visual Trace Explorer software. `http://vite.gforge.inria.fr/`. Accessed: February 2021.

[115] Xia, J., Chandrasekaran, S., Gu, M., and Li, X. S. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications 17* (2010), 953–976.

[116] Yamazaki, I., Ida, A., Yokota, R., and Dongarra, J. Distributed-memory lattice $\mathcal{H}$-Matrix factorization. *The International Journal of High Performance Computing Applications 33*, 5 (2019), 1046–1063.

[117] Yarkhan, A., Kurzak, J., and Dongarra, J. Quark users' guide. *Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tennessee 268* (2011).

[118] Yuan, N., Li, J., Hu, J., and Bhardwaj, A. High-performance computing and engineering applications in electromagnetics. *International Journal of Antennas and Propagation 2012* (03 2012).

[119] Zechner, J., and Beer, G. A fast elasto-plastic formulation with hierarchical matrices and the boundary element method. *Computational Mechanics 51* (2013), 443–453.

Index