The Journal of Supercomputing (2026) 82:271
https://doi.org/10.1007/s11227-026-08412-w

Towards a hierarchical approach for autotuning task-based libraries

Jesús Cámara[1] · Javier Cuenca[2] · Murilo Boratto[3]

Received: 15 October 2025 / Accepted: 1 March 2026

Abstract
This work proposes a hierarchical approach to reduce the training time of task-based routines by reusing previously obtained autotuning information. This approach has been integrated into a working prototype of Chameleon, a dense linear algebra software whose tile-based routines are executed on the available computational resources by means of a runtime system. The results show that this approach provides a high degree of scalability to the entire self-optimization process, achieving a reduction in training time of up to 80% and an appropriate selection of values for the adjustable parameters.

Keywords Hierarchical autotuning · Heterogeneous computing · Task-based scheduling

1 Introduction

Linear algebra routines are widely used as basic computational kernels in scientific software, so optimizing them for today's hardware platforms would significantly improve the resolution of important scientific and engineering challenges. In this regard, highly efficient linear algebra libraries are freely available, such as oneMKL [1], PLASMA [2], or MAGMA [3], whose routines base their performance on optimized implementations of basic tile-based kernels [4]. Current hardware platforms, meanwhile, are typically made up of a set of hybrid computing nodes composed

Jesús Cámara, Javier Cuenca and Murilo Boratto: contributed equally to this work.

✉ Jesús Cámara
jesus.camara@infor.uva.es

Javier Cuenca
jcuenca@um.es

Murilo Boratto
murilo.boratto@fieb.org.br

1 Department of Informatics, University of Valladolid, Valladolid, Spain

2 Department of Engineering and Technology of Computers, University of Murcia, Murcia, Spain

3 Supercomputing Center for Industrial Innovation, SENAI CIMATEC, Salvador, Bahia, Brazil

of multiple processing units, such as multicore CPUs and one or more accelerators (usually GPUs). The increasing heterogeneity of these platforms has led to the development of task-based libraries, in which part of the work is delegated to third-party software, such as a runtime system. As a result, these libraries often have multiple parameters whose values need to be carefully configured to achieve high performance.

In general terms, an autotuning methodology can be used to provide routines with self-optimization capabilities, so that they can search for the parameter values that offer the lowest execution time. However, in complex computing systems, the number of adjustable parameters and their possible values can be extremely large, making it unfeasible to exhaustively explore the search space. In this context, using a hierarchical approach can help overcome this problem by organizing the autotuning process into levels according to processing units and routines. Consequently, smaller sets of decisions should be made at each level of the hierarchy using information from the lower levels. That is, the autotuning of a routine for a given hardware platform is addressed in the same way as the coding of that routine, which relies on lower-level routines that have already been programmed with proven robustness. With this migration from the classic hierarchical software scheme to an autotuning engine, the aim is to inherit its well-known benefits, such as modularity, build efficiency, design robustness, scalability, and a certain degree of fault tolerance.

In this work, we propose the use of this approach, focusing particularly on task-based libraries. The main novel points covered are:

- Adaptation of the hierarchical autotuning methodology proposed in [5] to enable the selection of the best values for the adjustable parameters in task-based libraries.
- Integration of the proposed methodology into a working prototype of Chameleon [6], a task-based library that uses StarPU [7] to dynamically schedule tasks (the basic computational kernels) on the processing units available on the hardware platform.
- Reduction in the time spent on the training phase when the hierarchical approach is applied to find the best values for the adjustable parameters.

The rest of the paper is organized as follows. First, a brief summary of the main features of the Chameleon library is presented in Sect. 2. Next, the design and operation of the hierarchical autotuning methodology is described in Sect. 3. After that, a proof of concept is presented in Sect. 4 to illustrate how this methodology can be applied to Chameleon routines. The experimental results obtained are shown in Sect. 5. Finally, the main conclusions drawn from this work are outlined in Sect. 6.

2 The Chameleon Software

Chameleon is a task-based dense linear algebra software derived from the PLASMA library. Internally, this software uses a runtime system for dynamic task scheduling on heterogeneous systems. These tasks are executed using computational kernels from optimized linear algebra libraries, such as oneMKL for multicore CPUs and MAGMA or cuBLAS [8] for GPUs. These kernels, in turn, serve as building blocks for higher-level routines in Chameleon, which are designed to be executed on computing nodes comprising multicore CPUs and one or more GPUs.

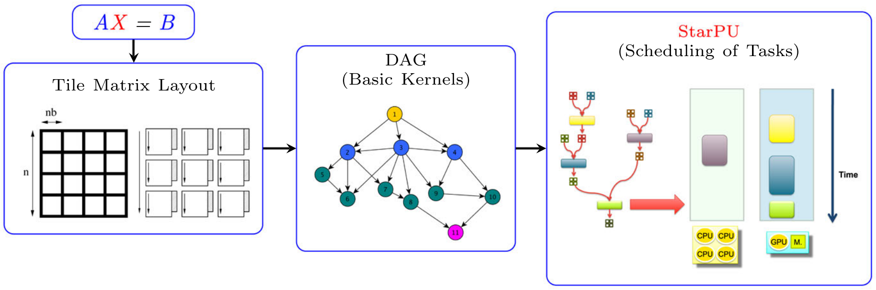

Fig. 1 Schematic representation of the execution of a linear algebra operation included in Chameleon using a task-based approach and a runtime scheduling system, such as StarPU

Figure 1 illustrates how a linear algebra operation included in Chameleon is decomposed into tasks to be scheduled and executed on the platform's processing units using the StarPU runtime system. First, the tasks required to compute the data tiles in the matrix layout are created based on the sequential task flow (STF) paradigm [9]. These tasks correspond to the basic computational kernels of the linear algebra routine to be executed. Next, a directed acyclic graph (DAG) is generated, which includes the dependencies between these tasks. After that, the tasks are scheduled using one of the scheduling policies provided by StarPU and, finally, they are executed on the processing units available in the hardware platform using optimized implementations of the basic kernels.

Recently, a proposal to improve this general approach has been published [10]. Its main objective is to adjust task sizes to the computing power of the processing units. This method focuses on transforming the static task graph, with a task grain of a specific pre-established size, into a dynamically adapted graph in which certain tasks can be decomposed, in turn, into subgraphs of child tasks.

3 Hierarchical Autotuning Methodology

In this section, the hierarchical autotuning methodology proposed to select the best values of the adjustable parameters that affect the performance of task-based libraries is described. An overview of this methodology was presented in [5]. Initially, it was focused on higher-level routines in which structuring into subtasks is carried out using lower-level routines whose execution is statically scheduled on the hardware platform. This static view of the workload distribution enables the searching for the optimal parameter values to be supported by theoretical–practical execution time models. On the other hand, a proposal for autotuning task-based linear algebra software was presented in [11]. In that work, an empirical approach based on a global search for the optimal values of all adjustable parameters in the routines was proposed, focusing solely on those related to high-level routines.

In this work, however, the methodology introduced in [5] has been adapted to the specific conditions of the framework composed by Chameleon and StarPU. Due to the dynamic nature of task scheduling in this framework, an approach based on theoretical

Table 1 Types of adjustable parameters in task-based libraries

AP	Meaning	Scope	Examples
RP	Routine Parameters	Routines	*nb* (block size), *ib* (inner-block size)
SchP	Scheduling Parameters	Runtime System	*ws, dmda, …* (scheduling policies)
SP	System Parameters	Processing Units	*#cores, #gpus*

models of the execution time of the routines is not appropriate. Therefore, it has been necessary to deal only with empirical search techniques, in the same way as in [11], but now with a hierarchical perspective on both software and hardware.

The entire self-optimization process is structured into different abstraction layers, which operate independently, taking advantage of the information exchanged between them. Basically, when a routine of a certain level has been self-optimized, it becomes a black box from which the optimal values of its adjustable parameters can be queried to be used directly in the autotuning process of any routine that internally uses it. As a result, the search space for the upper-level routine is greatly reduced as it is bounded only at its own level.

3.1 Operating mode

An overview of the hierarchical autotuning methodology adapted for task-based libraries is shown in Fig. 2. In these libraries, the performance of routines depends on the values selected for three subsets of adjustable parameters, AP. Table 1 summarizes the AP that can be considered when working with task-based libraries.

In order to find the best values for these AP, a search strategy (exhaustive, guided, hybrid, etc.) is selected to train each basic routine, R. For this purpose, R is executed in the system for each training problem size, n_t, varying the values of its AP according to the selected search strategy. In the case of an exhaustive search, for example, the routine is executed with all possible combinations for the values of the AP based on a fixed increment, thus obtaining an absolute optimum. However, if a guided search is chosen, the routine is executed starting with a specific combination of AP values and continues with successive combinations, using a specific heuristic, until a pseudo-local optimum is reached [11]. As a result of this training phase, the best AP values and the corresponding performance obtained (in GFlops) for each n_t are stored in a database. This database is implemented as a directory hierarchy with the following structure: $\{$platform $\rightarrow$ routine $\rightarrow$ parameter(s) $\rightarrow$ search strategy$\}$, where the last one stores the files containing the performance information for the best values obtained with each problem size.

In this way, to know which are the optimal (or near-optimal) values of its AP to solve a specific problem n_r, it is only necessary to read the j-th element from the corresponding file, being n_j the training problem size closest to n_r. If n_r is located at the same distance between two problem sizes, the AP values for the immediately higher problem size are considered. Since the file is sorted by problem size, a binary search is performed to find n_j. Therefore, the query time does not depend neither on

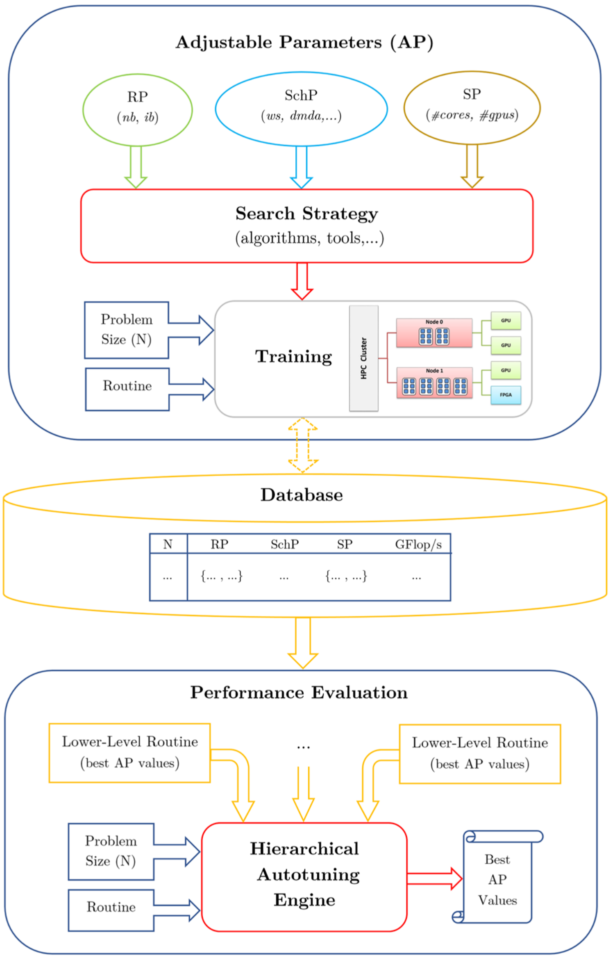

Fig. 2 Overview of the hierarchical autotuning methodology for task-based linear algebra libraries

how many AP are managed for R, nor on the search space that has been used at R training time for each adjustable parameter.

From this point onwards, when the training phase for a higher-level routine, R', is performed, the search space will be reduced to only its own AP. The best AP values for all basic routines used by R' are directly queried from the database. Likewise, all the automatic optimization information of these higher-level routines is also stored in the database to be used for autotuning routines of even higher levels.

Other reasons also justify why this approach significantly reduces the total training time required for optimization, offering high scalability and modularity. Firstly, the training of basic routines can be performed simultaneously on the different processing units. Secondly, when a new processing unit is added to the hardware platform, it is only necessary to train the basic routines on this unit because the performance information previously stored in the database for the rest of the processing units remains useful.

Finally, when a new higher-level routine needs to be self-optimized, no additional training of the basic routines it uses is required, as the performance information stored in the database for each of them also remains useful.

It should be noted that this methodology can be a useful complement to one of the latest improvements in the management of the Chameleon task graph [10], where certain tasks can be decomposed hierarchically into a subgraph of child tasks. Given a task, the choice of its most appropriate subgraph can be made using subgraph performance models. A lower-level autotuning of each basic routine would allow these models to be created and managed with a reduced impact on overall time.

4 A working prototype

In this section, the behaviour of a working prototype, built as a proof of concept, is described. This prototype consists of the Cholesky, LU, and QR factorization routines from Chameleon, properly adapted to have autotuning capability in accordance with the proposed methodology. Reference implementations of these routines for systems with hierarchical levels of memory are part of the LAPACK library [12]. Basically, they consist of a succession of panel (or block-column) factorizations followed by updates of the trailing submatrix. In Chameleon, these routines are arranged in a tile-based scheme, i.e. the matrix to be factorized is split into multiple submatrices, or tiles [13]. In this way, the routine is divided into smaller tasks, which correspond to the computational kernels (lower-level routines) involved in performing the factorization.

Figure 3 shows the tasks involved in the matrix decomposition carried out in the Cholesky routine (`POTRF_TILE`), and the task DAG generated before executing them on the processing units. These tasks correspond to the computational kernels used to perform this matrix operation: `POTRF`, `TRSM`, `SYRK`, and `GEMM`. The complexity of scheduling these tasks and solving data dependencies is delegated to StarPU. By default, it uses the *eager* simple greedy scheduler, which provides correct load balancing. However, other schedulers, such as *ws* or *dmda*, can be specified using the `STARPU_SCHED` environment variable. The selection of the best $SchP$ values using the proposed hierarchical approach exceeds the scope of this work, but is considered for future work. Nevertheless, a preliminary study for the Cholesky routine using a global autotuning process with a reduced set of problem sizes is presented in [11].

In Chameleon, the scheduling policies do not allow the selection of the best values for the adjustable parameters of the computational kernels. Thus, this library has been adapted so that each routine can automatically select the best values for the RP based on the problem size to be solved, n. Algorithm 1 shows the adaptation made for the Cholesky routine. Since the performance of this routine depends mainly on the tile size, nb, only this adjustable parameter is considered in the RP. The call to `chameleon_tune` (line 1) allows the routine to automatically obtain the best value of nb for the input problem size, n.

Likewise, Fig. 4 shows the decomposition into tasks for the LU factorization routine (`GETRF_TILE`), as well as the DAG generated according to Algorithm 2. In this algorithmic scheme, in addition to nb (line 1), the parameter ib (line 2) also appears, which corresponds to the inner-block size used by the `GETRF` basic kernel to solve a

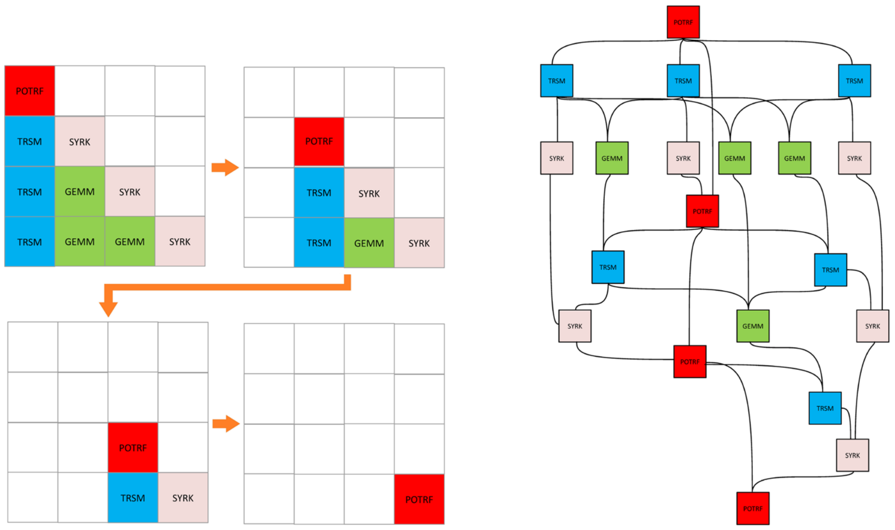

Fig. 3 Decomposition of a tile-based matrix into tasks with the computational kernels involved in the Cholesky routine (left) and DAG used to execute these tasks (right)

Algorithm 1 : POTRF_TILE. Tiled Cholesky routine of Chameleon adapted to be autotuned hierarchically.

```
1: $nb$ = chameleon_tune(POTRF_TILE, $n$);
2:
3: for $k = 0 \ldots (mt - 1)$ do
4:    INSERT_TASK_POTRF(options, $nb$, A[$k$][$k$]);
5:    for $j = (k + 1) \ldots (mt - 1)$ do
6:       INSERT_TASK_TRSM(options, $nb$, A[$k$][$k$], A[$j$][$k$]);
7:    end for
8:    for $i = (k + 1) \ldots (nt - 1)$ do
9:       INSERT_TASK_SYRK(options, $nb$, A[$i$][$k$], A[$i$][$i$]);
10:      for $j = (i + 1) \ldots (mt - 1)$ do
11:         INSERT_TASK_GEMM(options, $nb$, A[$j$][$k$], A[$i$][$k$], A[$j$][$i$]);
12:      end for
13:   end for
14: end for
```

subproblem of size nb (line 5). Thus, it has been autotuned and its training information has been stored in the database previously. (This situation was described in Sect. 3.) Consequently, to perform an automatic optimization of GETRF_TILE, the search space has been reduced by having a smaller set of adjustable parameters, from $\{nb, ib\}$ to only $\{nb\}$. Given a training problem size n, for each tile size nb, the optimal ib value is just queried from the database.

On the other hand, Fig. 5 shows the decomposition into tasks for the QR routine (GEQRF_TILE), as well as the DAG generated according to the execution of Algorithm 3. In this case, the algorithmic scheme has more adjustable parameters (lines 2-5) than the LU and Cholesky routines. These parameters correspond to the inner-block size of the lower-level routines (lines 8, 10, 13, and 15) called by this tile-based

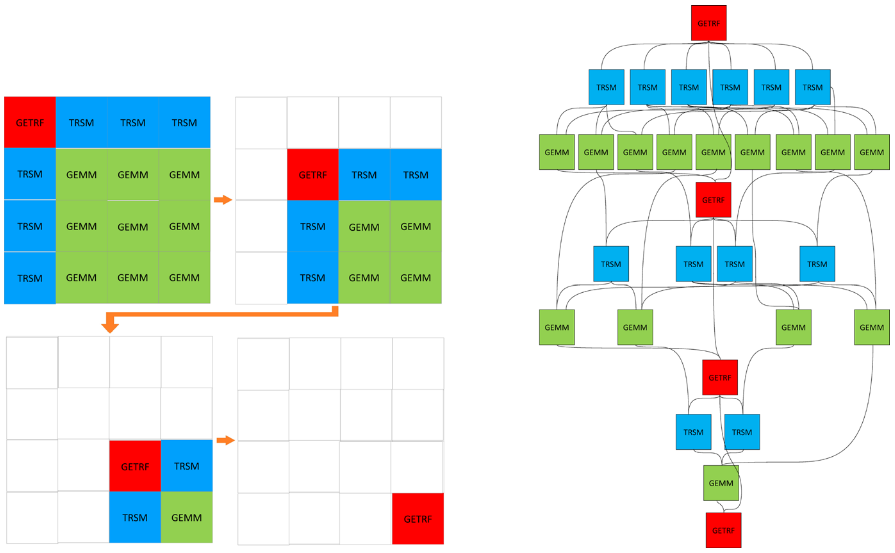

Fig. 4 Decomposition of a tile-based matrix into tasks with the computational kernels involved in the LU routine (left) and DAG used to execute these tasks (right)

Algorithm 2 : GETRF_TILE. Tiled LU routine of Chameleon adapted to be autotuned hierarchically.

```
1:  $nb$ = chameleon_tune(GETRF_TILE, $n$);
2:  $ib$ = chameleon_tune(GETRF, $nb$);
3:
4:  for $k = 0 \ldots (min(mt, nt) - 1)$ do
5:      INSERT_TASK_GETRF(options, $ib$, $nb$, A[$k$][$k$]);
6:      for $j = (k + 1) \ldots (mt - 1)$ do
7:          INSERT_TASK_TRSM(options, $nb$, A[$k$][$k$], A[$j$][$k$]);
8:      end for
9:      for $i = (k + 1) \ldots (nt - 1)$ do
10:         INSERT_TASK_TRSM(options, $nb$, A[$k$][$k$], A[$k$][$i$]);
11:         for $j = (k + 1) \ldots (mt - 1)$ do
12:             INSERT_TASK_GEMM(options, $nb$, A[$j$][$k$], A[$k$][$i$], A[$j$][$i$]));
13:         end for
14:     end for
15: end for
```

routine. As in previous higher-level routines, information about the best values for these parameters is obtained in advance by training the lower-level routines involved in this factorization. Subsequently, in the `GEQRF_TILE` autotuning process, these values are queried from the database, with no additional experimental cost.

Finally, it should be noted that all the automatic optimization information of these three routines (`POTRF_TILE`, `GETRF_TILE`, and `GEQRF_TILE`) is also stored in the database to autotune routines of even higher levels.

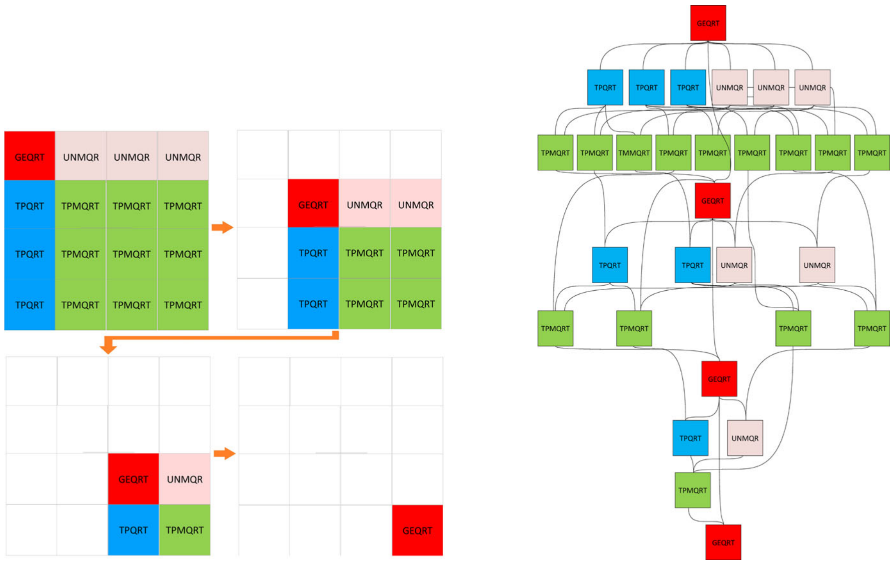

Fig. 5 Decomposition of a tile-based matrix into tasks with the computational kernels involved in the QR routine (left) and DAG used to execute these tasks (right)

Algorithm 3 : GEQRF_TILE. Tiled QR routine of Chameleon adapted to be autotuned hierarchically.

```
1: $nb$ = chameleon_tune(GEQRF_TILE, $n$);
2: $ib1$ = chameleon_tune(GEQRT, $nb$);
3: $ib2$ = chameleon_tune(UNMQR, $nb$);
4: $ib3$ = chameleon_tune(TPQRT, $nb$);
5: $ib4$ = chameleon_tune(TPMQRT, $nb$);
6:
7: for $k = 0 \ldots (min(mt, nt) - 1)$ do
8:     INSERT_TASK_GEQRT(options, $ib1$, $nb$, A[$k$][$k$]), T[$k$][$k$];
9:     for $j = (k + 1) \ldots (nt - 1)$ do
10:        INSERT_TASK_UNMQR(options, $ib2$, $nb$, D[k], T[$k$][$k$], A[$k$][$j$]);
11:    end for
12:    for $i = (k + 1) \ldots (mt - 1)$ do
13:        INSERT_TASK_TPQRT(options, $ib3$, $nb$, A[$k$][$k$], A[$i$][$k$], T[$i$][$k$]);
14:        for $j = (k + 1) \ldots (nt - 1)$ do
15:            INSERT_TASK_TPMQRT(options, $ib4$, $nb$, A[$i$][$k$], T[$i$][$k$], A[$k$][$j$], A[$i$][$j$]);
16:        end for
17:    end for
18: end for
```

5 Experimental Results

This section summarizes the experimental results obtained with the hierarchical autotuned routines of the prototype described in Sect. 4. As mentioned, only RP has been considered as the AP to be selected when applying hierarchical autotuning. The experiments focus on analysing both the training time required to find the best RP values using the proposed approach and the performance obtained with the selected values.

Table 2 Training times (in seconds) and best values of $\{nb, ib\}$ obtained for the **LU** routine on **jupiter** when both global autotuning ($Global\_AT$) and hierarchical autotuning ($Hier\_AT$) are applied for a set of problem sizes. The reduction obtained (in %) with respect to global autotuning is also shown

n	*Auto_AP (nb, ib)*	*Global_AT (Sec.)*	*Hier_AT (Sec.)*	*Reduction (%)*
1000	(128, 64)	208	44	79
2000	(128, 96)	479	99	79
4000	(256, 256)	1337	298	78
6000	(256, 64)	3411	800	77
8000	(384, 192)	8332	2019	76
10000	(512, 512)	18348	4526	75
12000	(768, 384)	35106	8737	75
14000	(768, 192)	63182	15792	75
16000	(768, 384)	106291	26636	75

The results are compared with those obtained when using the Chameleon library with the default RP values and with those obtained when applying a global autotuning [11].

The hardware platform used for the experiments is a heterogeneous cluster that consists of a set of computing nodes of different characteristics connected via a Gigabit Ethernet network. The nodes selected for the experimental study are:

- **jupiter**: with 2 Intel Xeon E5-2620 (hexa-core) CPUs at 2.0 GHz and 6 GPUs: 4 GeForce GTX 590 and 2 Tesla C2075.
- **venus**: with 2 Intel Xeon E5-2620 (hexa-core) CPUs at 2.40 GHz and 2 GPUs: a NVIDIA GeForce GT 640, and a NVIDIA PNY Quadro P2200.

The experiments have been carried out with the QR and LU autotuned routines on the specified computing nodes. Tables 2, 3, 4, and 5 show a comparison of the training time needed (in seconds) to obtain the best values of the adjustable parameters (tile size, *nb*, and inner-block size, *ib*) depending on whether a global autotuning method is used (*Global_AT*) or the proposed hierarchical autotuning method (*Hier_AT*). As can be seen in the last column, there is a very noticeable reduction in the training time for both routines and in the two computing nodes when the reuse of the autotuning information from lower levels within the hierarchy of routines is applied (Fig. 2). More specifically, it can be stated that global autotuning is forced to perform a joint search for the optimal values of all AP. As a result, in many cases, the search for low-level AP values is repeated for different combinations of high-level AP values. These redundant combinations can be avoided by using the hierarchical approach, reducing the amount of experimentation required to train the higher-level routine.

On the other hand, a performance analysis has been carried out for the LU and QR routines using the $\{nb, ib\}$ values selected by hierarchical autotuning ($Auto\_AP$ column in Tables 2, 3, 4, and 5) and with the default values configured in Chameleon for these AP (nb = 320, ib = 48). As shown in Fig. 6, in both computing nodes (jupiter and venus) and for both routines, a higher gain is obtained for small problem sizes, ranging from 20% to 70%. This improvement slows down for medium problem sizes, ranging from 1% to 4%. However, for large problem sizes, the gain obtained

Table 3 Training times (in seconds) and best values of $\{nb, ib\}$ obtained for the **QR** routine on **jupiter** when both global autotuning ($Global\_AT$) and hierarchical autotuning ($Hier\_AT$) are applied for a set of problem sizes. The reduction obtained (in %) with respect to global autotuning is also shown

n	*Auto_AP (nb, ib)*	*Global_AT (Sec.)*	*Hier_AT (Sec.)*	*Reduction (%)*
1000	(128, 32)	220	45	80
2000	(128, 32)	616	121	80
4000	(320, 48)	4342	749	83
6000	(320, 48)	19388	3136	84
8000	(384, 96)	62371	9373	85
10000	(640, 160)	146163	23356	84
12000	(640, 160)	244307	46671	81
14000	(640, 160)	386037	85569	78
16000	(896, 224)	636379	146342	77

Table 4 Training times (in seconds) and best values of $\{nb, ib\}$ obtained for the **LU** routine on **venus** when both global autotuning ($Global\_AT$) and hierarchical autotuning ($Hier\_AT$) are applied for a set of problem sizes. The reduction obtained (in %) with respect to global autotuning is also shown

n	*Auto_AP (nb, ib)*	*Global_AT (Sec.)*	*Hier_AT (Sec.)*	*Reduction (%)*
1000	(128, 64)	52	28	46
2000	(128, 64)	118	32	73
4000	(256, 256)	432	115	73
6000	(384, 384)	1390	355	75
8000	(384, 96)	3817	974	75
10000	(768, 192)	8918	2229	75
12000	(768, 576)	17339	4374	75
14000	(1152, 1152)	31602	7951	75
16000	(1152, 288)	53405	13371	75

Table 5 Training times (in seconds) and best values of $\{nb, ib\}$ obtained for the **QR** routine on **venus** when both global autotuning ($Global\_AT$) and hierarchical autotuning ($Hier\_AT$) are applied for a set of problem sizes. The reduction obtained (in %) with respect to global autotuning is also shown

n	***Auto_AP (nb, ib)***	*Global_AT (Sec.)*	*Hier_AT (Sec.)*	*Reduction (%)*
1000	(128, 32)	55	16	71
2000	(128, 32)	206	52	75
4000	(320, 48)	2943	639	78
6000	(320, 48)	16434	3737	77
8000	(320, 48)	55132	11171	80
10000	(320, 48)	126603	24138	81
12000	(320, 48)	211798	44509	79
14000	(320, 48)	328901	84250	74
16000	(320, 48)	531519	140979	74

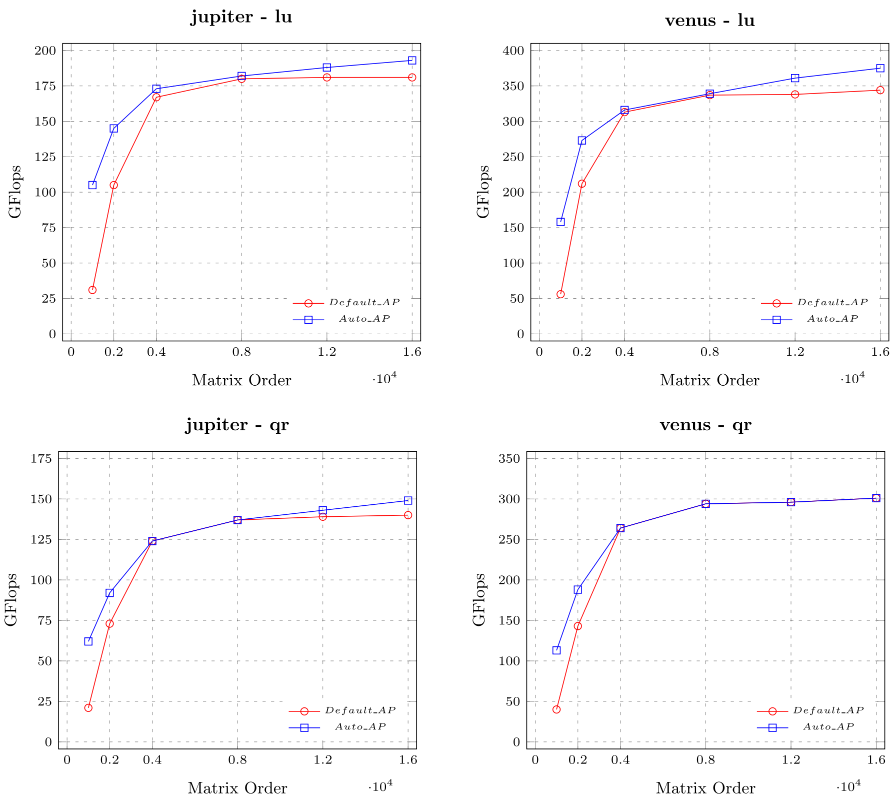

Fig. 6 Performance obtained on **jupiter** (left) and **venus** (right) for the **LU** (top) and **QR** (bottom) routines when using, on the one hand, the default values in Chameleon ($Default\_AP$) for $\{nb, ib\}$ and, on the other hand, those selected by hierarchical autotuning ($Auto\_AP$)

increases again up to 8%. This variability is due to the fact that, for intermediate sizes, the default values in Chameleon for nb and ib are optimal or near-optimal for the LU and QR routines. Therefore, the methodology also selects these values. Nevertheless, when the problem size differs from these intermediate sizes, the decisions made by hierarchical autotuning are crucial for achieving maximum performance.

In conclusion, these results highlight the usefulness of the proposed hierarchical approach, as they show, on the one hand, the considerable reduction in training time compared to a global autotuning methodology and, on the other hand, that decision-making remains effective in terms of the AP values selected and performance obtained.

6 Conclusions

In this work, the challenge of setting up an autotuning system for a non-trivial execution environment is described. This environment consists of linear algebra routines that operate on data organized in tiles, which are computed using specific lower-level routines. These routines are dynamically scheduled by a runtime system for execution on the processing units available on the hardware platform (Sect. 2).

In this context, a global autotuning system that attempts to search for the best values across the entire set of adjustable parameters may exceed a reasonable training time. This situation becomes even more challenging when the routine or the hardware platform increases in complexity and/or the set of adjustable parameters grows. For all these reasons, a hierarchical autotuning methodology for task-based libraries is proposed (Sect. 3). A working prototype has been built integrating this methodology into several routines of Chameleon (Sect. 4). Next, an experimental study has been carried out to show, on the one hand, that hierarchical autotuned routines enable better performance than the basic implementation provided by the library and, on the other hand, that training time can be substantially reduced compared to what would be obtained if a global autotuning process were applied (Sect. 5).

As future work, the set of adjustable parameters supported by this prototype will be extended, including the task scheduling policy to be used by the runtime system and the selection of the number and type of processing units on the hardware platform.

Finally, it can be noted that this methodology can be a useful tool for taking advantage of one of the latest improvements in Chameleon [10]. It is focused on transforming the static task graph of each routine into a dynamically adapted graph, where some tasks can be decomposed into subgraphs of child tasks. Thus, given a task, the choice of its most suitable subgraph could be based on a lower-level autotuning of each basic routine.

Acknowledgements This work is supported by Grant PID2022-136315OB-I00 and Grant PID2022-142292NB-I00, both funded by MCIN/AEI/10.13039/501100011033/ and by “ERDF A way of making Europe”, EU.

Author Contributions J.C. helped to write the manuscript, conducted the experimental study, and prepared Figures 1-2. J.C. wrote the main manuscript text and prepared Figures 3-5. M.B. helped to write the manuscript. All authors reviewed the manuscript.

Funding Open access funding provided by FEDER European Funds and the Junta de Castilla y León under the Research and Innovation Strategy for Smart Specialization (RIS3) of Castilla y León 2021-2027.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare no conflict of interest.

References

1. Intel oneAPI Math Kernel Library (oneMKL). https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. [Online; Accessed 20.02.2026]
2. PLASMA. https://icl.utk.edu/plasma. [Online; Accessed 20.02.2026]
3. Agullo E, Demmel J, Dongarra J, Hadri B, Kurzak J, Langou J, Ltaief H (2009) Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. J Phys: Conf Ser 180(1)
4. Golub G, Loan CFV (2013) Matrix Computat, 4th edn. The John Hopkins University Press, Baltimore
5. Cámara J, Cuenca J, Giménez D (2020) Integrating software and hardware hierarchies in an autotuning method for parallel routines in heterogeneous clusters. J Supercomput 76(12):9922–9941
6. Chameleon: A dense linear algebra software for heterogeneous architectures. https://solverstack.gitlabpages.inria.fr/chameleon. [Online; Accessed 20.02.2026]
7. Augonnet C, Thibault S, Namyst R, Wacrenier P (2011) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurren Computat Pract Exper 23(2):187–198
8. CUDA Basic Linear Algebra Subroutine library (cuBLAS). https://docs.nvidia.com/cuda/cublas/index.html. [Online; Accessed 20.02.2026]
9. Pei Y, Bosilca G, Dongarra J (2022) Sequential task flow runtime model improvements and limitations. In: 2022 IEEE/ACM international workshop on runtime and operating systems for supercomputers (ROSS), pp 1–8
10. Faverge M, Furmento N, Guermouche A, Lucas G, Namyst R, Thibault S, Wacrenier P-A (2023) Programming heterogeneous architectures using hierarchical tasks. In: Euro-Par 2022: Parallel Processing Workshops, pp 97–108
11. Cámara J, Cuenca J, Boratto M (2023) Improving the performance of task-based linear algebra software with autotuning techniques on heterogeneous architectures. In: Proceedings of the 23rd International conference on computational science (ICCS 2023). Lecture Notes in Computer Science, vol 14073, pp 668–682
12. Anderson E, Bai Z, Bischof CH, Blackford LS, Demmel J, Dongarra JJ, Croz JD, Greenbaum A, Hammarling S, McKenney A, Sorensen DC (1999) LAPACK Users' Guide, 3rd edn. Environments and Tools. SIAM, Philadelphia, Software
13. Agullo E, Augonnet C, Dongarra J, Ltaief H, Namyst R, Thibault S, Tomov S (2010) Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In: GPU Computing Gems vol 2. Morgan Kaufmann