sLASs: A Fully Automatic Auto-tuned Linear Algebra Library based on OpenMP extensions implemented in OmpSs

(LASs Library)

Pedro Valero-Lara[a,*], Sandra Catalán[a], Xavier Martorell[b], Tetsuzo Usui[c], Jesús Labarta[b]

[a]*Barcelona Supercomputing Center (BSC), Barcelona, Spain*
[b]*Universitat Politècnica de Catalunya, Barcelona, Spain*
[c]*Next Generation Technical Computing Unit, Fujitsu Limited, Kawasaki, Japan*

Abstract

In this work we have implemented a novel Linear Algebra Library on top of the task-based runtime OmpSs-2. We have used some of the most advanced OmpSs-2 features; weak dependencies and regions, together with the final clause for the implementation of auto-tunable code for the BLAS-3 TRSM routine and the LAPACK routines NPGETRF and NPGESV. All these implementations are part of the first prototype of sLASs library, a novel library for auto-tunable codes for linear algebra operations based on LASs library. In all these cases, the use of the OmpSs-2 features presents an improvement in terms of execution time against other reference libraries such as, the original LASs library, PLASMA, ATLAS and Intel MKL. These codes are able to reduce the execution time in about 18% on big matrices, by increasing the IPC on GEMM and reducing the time of task instantiation. For a few medium matrices, benefits are also seen. For small matrices and a subset of medium matrices, specific optimizations that allow to increase the degree of parallelism in both, GEMM and TRSM tasks, are applied. This strategy achieves an increment in performance of up to 40%.

Keywords: LASs, OmpSs, Linear Algebra, Auto-tuning

*Corresponding author
Email address: `pedro.valero@bsc.es` (Jesús Labarta)

2010 MSC: 00-01, 99-00

1. Introduction

Nowadays computer architectures present a highly variable number of cores, ranging from a few to thousands in high performance platforms, that need to be used appropriately in order to attain high performance. This situation is even more relevant in linear algebra operations, where input data parameters often change during the operation (e.g. LU factorization). For this reason, auto-tuned codes that are able to adapt the amount of workload depending on the input parameters to the architecture characteristics are essential in order to make good use of the available resources.

Following the idea of auto-tunable codes but targeting an approach through the use of tasks, we present in this work the auto-tunable implementation of three kernels included in LASs [1], a novel linear algebra library based on OmpSs [1]. There exist several task-based parallel programming models such as OpenMP, Cilk++ or Wool, among others. We make use of OmpSs programming model because it is based on tasks and also extends OpenMP directives to give support to asynchronous parallelism and device heterogeneity. The main reasons to choose OmpSs are: i) This model presents efficient management of the threads based on the use of queues, without the need of dealing with the overhead found in other models, such as the fork-join model used in OpenMP. ii) Unlike other programming models, in OmpSs we find some features, not found in other programming models, such as weak dependencies and regions, which are necessary for the implementation of our auto-tunable codes. iii) OmpSs is also especially well integrated with the tools used for the performance evaluation Extrae, which allows to obtain execution traces, and Paraver [2], a visualization and analysis tool for the traces.

In this work, we explore a set of algorithmic and programming optimizations

[1]`https://pm.bsc.es/mathlibs/lass`

on BLAS-3 and LAPACK level routines based on tasking. We have focused on TRSM and LU factorization since these are the most characteristic and widely used routines for this kind of operations on dense matrices. We start analyzing the BLAS 3 TRSM routine. Although this is not a LAPACK routine, it is used after the factorization of the most LAPACK routines, such as GETRF or POTRF, in their corresponding solve, such as GESV or POSV. We focus on those programming optimizations, which allow to implement auto-tunable codes by adapting the execution, according to the features of the architecture and the parameters of the problem, in particular, the size of the matrix/matrices to be computed. We compare our performance results against PLASMA [3], the reference linear algebra library implemented in OpenMP, and other reference libraries: ATLAS and Intel MKL, to analyze the potential benefits of the presented approach.

In order to implement the optimizations proposed in [4] for TRSM and make the code adaptable in order to apply them when they are suitable, we use some of the latest features provided by OmpSs-2 [1]. In those scenarios where optimizations are not appropriate, the auto-tunable code is able to run the non-optimized code, which works better in these scenarios. The auto-tunable code is created through the use of nesting, weak dependencies, regions, and the final clause. Weak dependencies [5] and regions [6] are specific of OmpSs-2 programming model (regions syntax is also included in OpenMP), while nesting and the final clause [7] are included in both, OmpSs-2 and OpenMP. With the combination of nesting and weak dependencies mechanism we are able to balance computations better thanks to the relaxation of dependencies among parent and children tasks, which turns into a higher parallelization degree of the tasks. On the other hand, the final clause allows us to choose the most appropriate version of the code to be executed regarding the input matrix size. The different optimizations code are automatically selected at run time.

All the results presented in this paper are obtained on a node of the Marenostrum 4 supercomputer. Each node features 2 sockets Intel Xeon Platinum 8160 with 24 cores each at 2.10GHz for a total of 48 cores per node. Moreover, each core has a private L1 cache of 32KB, a private L2 cache of 1MB and all the cores in the socket share an L3 cache of 33MB. The peak performance for a single node is 2300 GFLOPS. We use MKL for the single-threaded BLAS and LAPACK routines. All tests are run with 48 threads and use OmpSs-2 2018.11 version. In addition, mercurium 2.2.0 compiler (icc version 17.0.4) is used with the following flags *-DLASs_WITH_MKL -O3 -Wall –ompss-2 –openmp-compatibility*.

The paper is structured as follows; Section 2 discusses related work, while in Section 4 we revisit the optimizations proposed for TRSM in [4] and analyze the benefits of the auto-tuned version of LASs. In Sections 5 and 6, we present the auto-tuned version for NPGETRF and NPGESV respectively. Finally, Section 7 presents the conclusions of the work.

2. State of the art

PLASMA [3] is the reference library for Dense Linear Algebra. Based on OpenMP, PLASMA parallelizes BLAS and LAPACK level operations targeting homogeneous multi-core and multi-socket platforms. As in our libraries, LASs and sLASs, PLASMA makes use of tiled algorithms in order to distribute the workload among the cores in the platform, using tasking. Other relevant linear algebra libraries that implement dense, sparse or both types of linear algebra operations are libFLAME [8], Intel MKL [2] and OpenBLAS [3], among others.

Another example in the Linear Algebra field is ATLAS [9], a software package that provides a complete BLAS [10] collection of kernels along with a subset of LAPACK [11] operations that deliver high performance thanks to its auto-tunable approach. ATLAS exploits low level features such as the size of the different memory hierarchy levels in order to customize required parameters (e.g. the block size) and consequently making better use of the resources to improve performance.

Finally, other approaches try to adapt the number of computations to the resources available on the platform, but the adaption is not done automatically

[2]https://software.intel.com/en-us/articles/intel-math-kernel-library-documentation
[3]http://www.openblas.net/

or its implementation requires major changes in the code. Among these strategies we find those described in [12, 13, 14]. In the first work, a batched GEMM is proposed in order to exploit better the resources of the platform. The idea is that a GEMM is decomposed in batches that contain thousands of smaller independent GEMM in order to maximize the use of the system. In the second work, a similar idea is proposed but varying the number of operations per batch aiming to balance the workload among the batches. Finally, in the last work, the authors focus on malleability in order to assign the appropriate amount of resources in each stage of an LU decomposition. In addition, they present a strategy (Early Termination) in order to automatically tune the block size used in the factorization when the workload between the panel factorization and the update tasks is unbalanced.

All the commented references show improvements for specific scenarios; however, they present one or more drawbacks: the tuning is not applicable for all the input cases, it is not easily implementable or it is not automatically done. This work presents an auto-tunable version of LASs library based on tasks and easily implemented thanks to the features provided by OmpSs-2. In addition, performance results show the capability of adapting the code at each execution point in order to attain high performance.

3. OmpSs

OmpSs is a task-based programming model that targets parallelism at node level by combining ideas used in OpenMP and StarSs. As shown in Figure 1, OmpSs mainly differs from OpenMP in the execution model; while OpenMP follows a fork-join approach, OmpSs implements thread-pool parallelism. However, in both cases applications are parallelized by annotating the source code (i.e. pragmas).

Parallelism is extracted at task level (the minimum execution entity), taking into account the existent data dependencies among them. These dependencies define a Directed Acyclic Graph (DAG) that is used by the runtime to decide

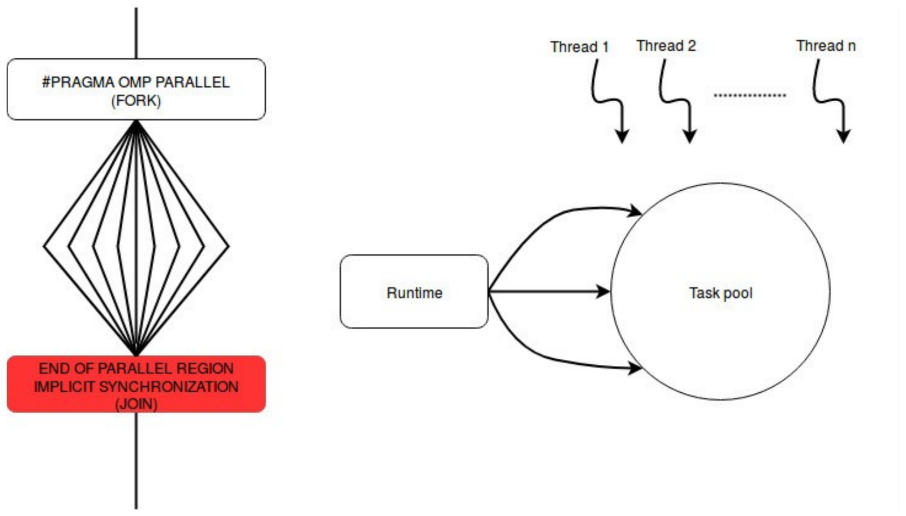


Figure 1: OpenMP and OmpSs programming models.

which tasks can be issued to execution. Once the data dependencies are satisfied, the task will be executed by a single thread.

4. trsm in sLASs

TRSM is the BLAS-3 routine that solves the equation:

$$A \cdot X = \alpha \cdot B \quad (1)$$

where α is a scalar, X and B are dense matrices, and A is a unit/non-unit upper/lower triangular matrix.

This routine involves the use of two kernels: TRSM and GEMM. TRSM is used in the diagonal blocks of the input matrix (dark blue blocks in Figure 2), while GEMM operates on the remaining blocks (light blue blocks in Figure 2).

First and for the sake of clarity, we review the optimizations presented in [4], the join of GEMM tasks and the modification of the tile size. The first optimization consists of joining the set of GEMM tasks which compute the tiles of one tile-column in one task (Figure 2 - right), instead of using one task per tile (Figure 2 - left). In this case, we can reduce the number of tasks to be computed,

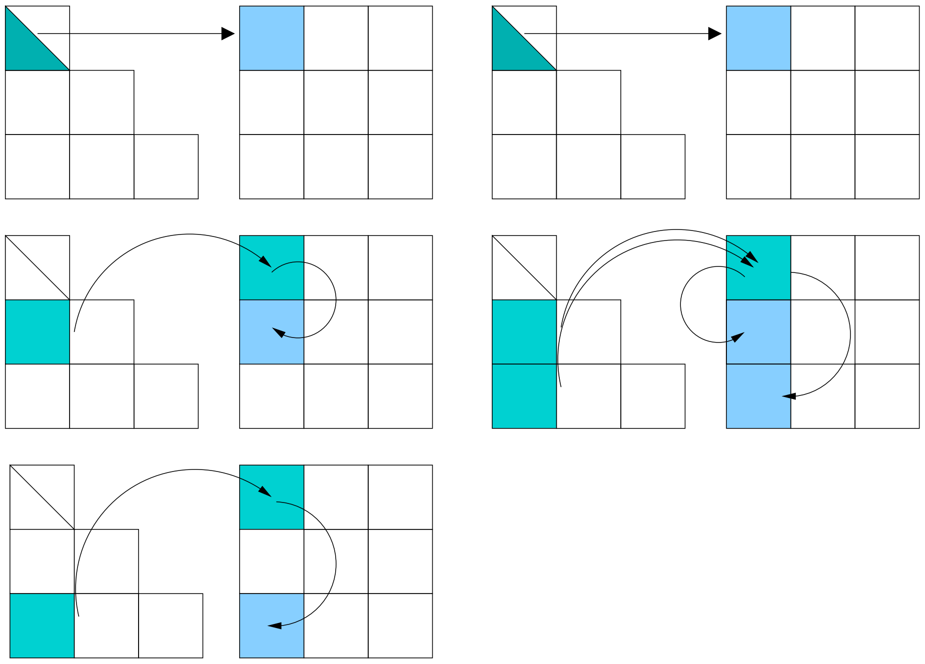


Figure 2: TRSM implementation in LASs (left) and LASs-opt (right).

as we can see in Figure 3; consequently the overhead of task management is reduced and better exploitation of the hierarchy memory may happen thanks to data locality improvement (direct consequence of keeping the data to be reused by GEMM tasks in the same core memory).

This optimization does not need big modifications in the code. Basically, this optimization consists of only moving the task instantiation and the modification of the data-dependence clauses to use the OmpSs regions, as we can see in Figures 4 and 5. In Figure 4 we present the original code included in LASs; Figure 5 shows the modifications in order to perform the join of GEMM tasks to create tile-columns.

To carry out this optimization efficiently, it is also necessary to apply the second optimization, modifying the tile size to ensure that we have as many tile-columns (*ct* in pseudo-codes) as number of cores. Due to this, we need to

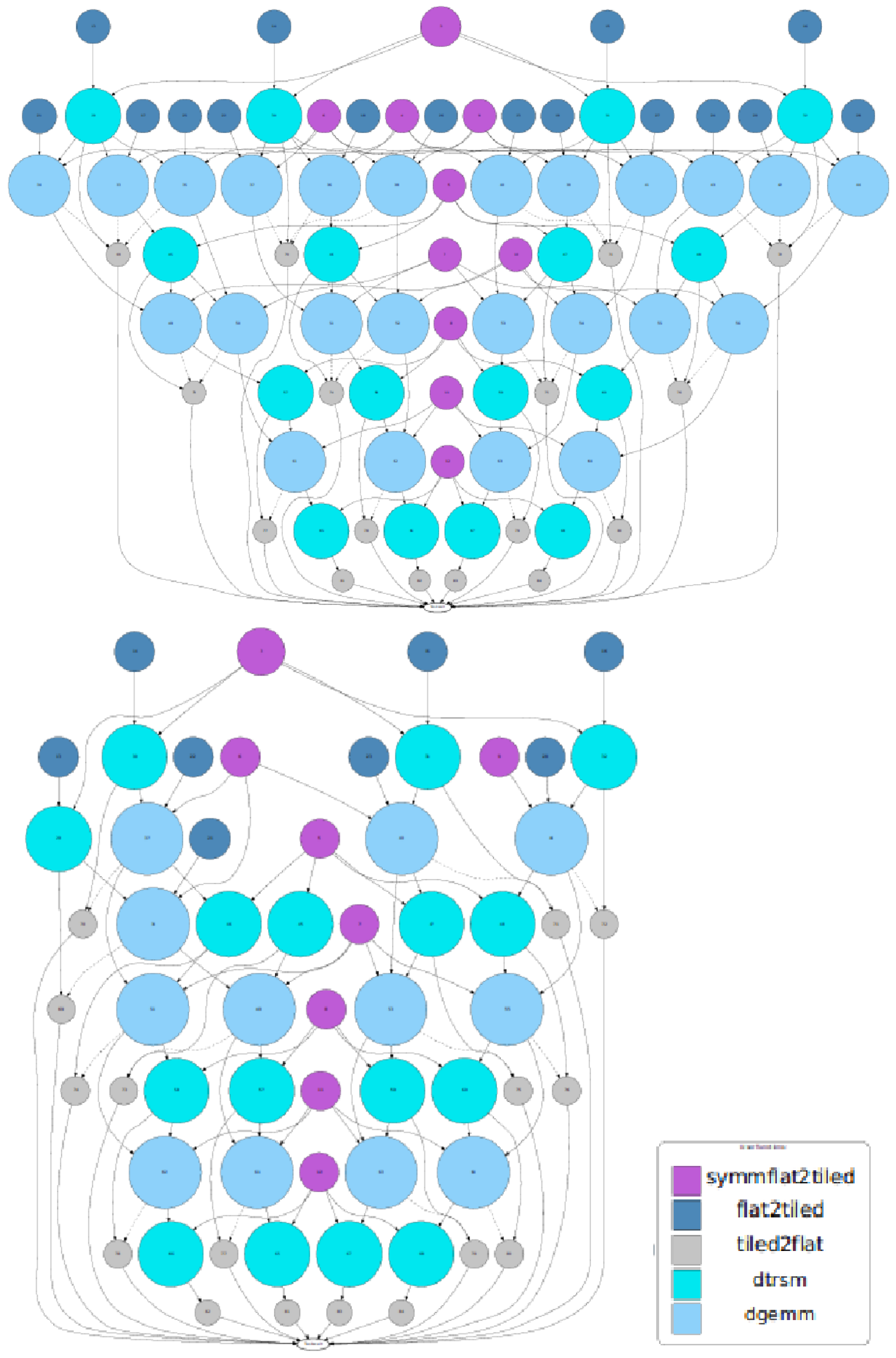


Figure 3: One task-gemm per tile (top) and one task computing many gemm per tile-column (bottom).

```
for ( d = 0; d < dt; d++) {
   for ( c = 0; c < ct; c++) {
      #pragma oss task in(TILE_A[d][d]) \
                       inout(TILE_B[d][c]) \
                       shared(TILE_A, TILE_B) \
                       firstprivate(d, c)
      dtrsm( ... );
   }
   for ( c = 0; c < ct; c++)  {
      for ( r = 0; r < rt; r++) {
         #pragma oss task in(TILE_A[d][d]) \
                          in(TILE_B[d][c]) \
                          inout(TILE_B[r][c]) \
                          shared(TILE_A, TILE_B) \
                          firstprivate(d, r, c)
         dgemm( ... );
       }
   }
}
```

Figure 4: Code for TRSM in LASs.

```
for ( d = 0; d < dt; d++) {
   for ( c = 0; c < ct; c++) {
      #pragma oss task in(TILE_A[d][d]) \
                       inout(TILE_B[d][c]) \
                       shared(TILE_A, TILE_B) \
                       firstprivate(d, c)
      dtrsm( ... );
   }
   for ( c = 0; c < ct; c++)  {
      #pragma oss task in(TILE_A[0:ct-1][d]) \
                       in(TILE_B[d][r]) \
                       inout(TILE_B[0:ct-1][c]) \
                       shared(TILE_A, TILE_B) \
                       firstprivate(d, r, c)
      for ( r = 0; c < rt; r++) {
         dgemm( ... );
       }
   }
}
```

Figure 5: Code for optimized TRSM in LASs.

change the size of the tiles according to the next equation:

$$tile\_size = N/\#cores \tag{2}$$

N being the number of columns of B and #cores the amount of available cores in the platform.

For example, for a given problem of N (number of columns of B) and M (number of rows of B) equal to 24576, we have a tile size equal to 512^2, $512 = 24576/48$.

Another important requirement necessary to carry out this optimization is the use of regions. Regions are an OmpSs feature, which allows programmers to define the data-dependencies in terms of a set or sets of elements instead of one particular variable or element of one array. This makes possible the implementation of the aforementioned optimizations, without a big effort from the programmer's point of view. This is illustrated in Figure 5.

We include some of our previous results (Figure 6) in order to show the performance results for LASs, the optimized version of LASs and also PLASMA, MKL and ATLAS as references. In the light of these results, we can see that in all cases MKL attains the highest performance. Nevertheless, for big matrices (for matrix sizes equal or greater than $24,576^2$) LASs-opt matches or is close to MKL performance. For small matrices, PLASMA followed by LASs could be an alternative to MKL, although the difference in performance is larger in this case. Finally, ATLAS provide the lowest GFLOPS.

This information is detailed in terms of execution time in Table 1. In these results, we can see that for big matrices, LASs_opt outperforms LASs 12%, while the benefits with respect to PLASMA reach 16% for the biggest matrix size.

Once we have summarized the main characteristics of the optimizations presented and evaluated in [4], we focus on presenting the effort carried out to implement auto-tunable codes. Based on the outcome of the previous study, we present in this work the implementation and analysis of the auto-tunable version of the operations, which are able to choose between the best already presented codes depending on the input matrix size. The main motivation be-

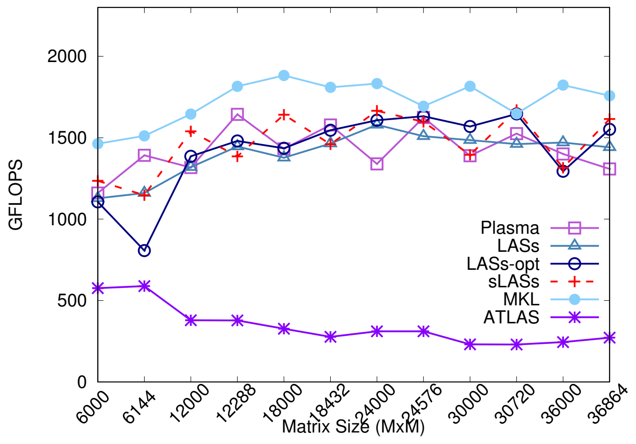


Figure 6: TRSM performance results for PLASMA, LASs, optimized LASs, sLASs, Intel MKL, and ATLAS.

Table 1: Execution time of TRSM for PLASMA, LASs and LASs with optimizations.

Time (s)	**6144**	**12288**	**18432**	**24576**	**30720**	**36864**
PLASMA	**0.16**	**1.12**	**3.96**	**9.20**	19.89	38.29
LASs	0.20	1.45	4.28	9.83	19.83	36.61
LASs_opt	0.29	1.50	4.06	9.44	**17.45**	**32.29**

hind this work is to unify both codes presented in [4], the original LASs and the LASs-opt. As shown, while the original LASs achieves the best performance on small and some medium matrices, LASs-opt consumes considerably less time than the original LASs on some medium matrices and big matrices. In order to unify both codes, we make use of the OmpSs features, weak dependencies and regions. We also make use of the OpenMP final clause. This OpenMP feature is included in OmpSs. For the sake of clarity and to help to understand how these codes are implemented, we include some pseudo-codes in Figure 7.

```
tune_dtrsm(ct);
is_final = smart_dtrsm(ct);
for ( d = 0; d < dt; d++){
   for ( c = 0; c < ct; c++){
      #pragma oss task ...
      dtrsm( ... );
    }
    for ( c = 0; c < ct; c++) {
       #pragma oss task \
       weakinout (TILEB[d:rt-1][c]) \
       final(is_final)
       for ( r = d; r < rt; r++) {
          #pragma oss task \
          inout(TILEB[r][c]) ...
          dgemm( ... );
       }
    }
}
```

Figure 7: Auto-tunable code (combination of codes included in Figures 4 and 5).

In the pseudo-code of Figures 4 and 5, we present the LASs original and LASs-opt implementations respectively, in the last pseudo-code (Figure 7) we show the proposed auto-tunable code, which unifies the main characteristics of the previous ones. We need an extra function in order to determine the tile size regarding the number of available cores (*tune_dtrsm*) in the pseudo-code. In addition, we also need to compute if the optimization (join of GEMM) needs to be computed or not. This is done by *smart_dtrsm*. Basically, this function computes the parameter (boolean) of the final clause, *is_final*. As reported by performance results, the join of GEMM is only effective on big matrices, in particular on those matrices whose number of columns is equal to or greater than $\#cores \times default\_tile\_size$. Note that the *default_tile_size* used on our target platform is 512^2. Therefore, we need a matrix with a number of columns equal to or greater than 24576^2 to effectively use the aforementioned optimization. In that case, the *is_final* boolean should be true when computing matrices of such sizes. If the *is_final* boolean is false, then the nested tasks are not instantiated, and the weak dependencies are computed as they were "strong" dependencies, performing the join of GEMM thanks to the use of OmpSs regions. Otherwise, if the matrix is not big enough the *is_final* boolean is false. In that case, the tasks instantiated in the "for r" loop are created. It is important to highlight that this case presents an overhead not presented in the original LASs, since a bigger number of tasks are instantiated. When the join of GEMM is not effective, we instantiate both, the parent tasks which use weak dependencies and regions ("for c") and the nested tasks ("for r"). However, as we see in the rest of this work, this overhead is not important.

4.1. TRSM *Auto-tunable Performance Results.*

After explaining the main characteristics and programming strategies to implement auto-tunable codes, in this Section, we evaluate the performance of the novel code developed during this work. First, we analyze the time consumed by the three variants: LASs, LASs-opt. and sLASs (auto-tunable code). These results are presented in Table 2.

Table 2: Execution time for TRSM in LASs, LASs-opt and sLASs.

Time (s)	**6144**	**12288**	**18432**	**24576**	**30720**	**36864**
LASs	**0.20**	**1.45**	4.28	9.83	19.83	36.61
LASs_opt	0.29	1.50	**4.06**	**9.44**	17.45	32.29
sLASs	**0.20**	1.54	4.35	9.51	**17.36**	**30.63**

The auto-tunable code (sLASs) is able to adapt its execution depending on the size of the matrices involved in the execution. When small matrices are used, the execution time of this new variant is similar to the time consumed by the original LASs. When the matrices are big, the execution time of LASs-opt is similar to that consumed by the auto-tunable code.

For the sake of completeness, we also analyze the performance of the different approaches in terms of GFLOPS. Figure 6 also includes the performance results in terms of GFLOPS for the three versions of the codethat we developed (LASs, LASs-opt and sLASs). In the plot, it is more clearly illustrated that the adaptation of the auto-tunable code is able to perform on the two well-defined areas of interest, from 6144^2 to 12288^2 and from 24576^2 to 36864^2.

To evaluate more in detail the effectiveness of the presented strategies we compare first the original LASs code against the auto-tunable version on small matrices (12288^2). Execution traces for this input matrix are presented in Figure 8. We can see that the overhead presented in the proposed auto-tunable code does not show an important difference with respect to the original LASs code. In fact, the instantiation of the extra tasks of the auto-tunable code corresponds to about 0.82%.

Similarly to the previous analysis, we now compare the performance achieved using the original LASs, the LASs-opt code with the auto-tuned code (sLASs) on big matrices (36864^2). The execution trace for each case is graphically illustrated in Figure 9. For the sake of performance analysis in the LASs-opt trace some of the join of GEMM present a different color, but in the sLASs trace, we did not do this distinction. As we can see, there is no significant difference between LASs-opt and sLASs traces, which present a quite similar behavior and performance. However, the reduction in execution time is remarkable with respect to the original LASs version.

Finally, we analyzed performance results in terms of cache misses. When obtaining L1 cache misses for the kernels that are invoked from TRSM routine, TRSM and GEMM, we see a dramatic reduction in sLASs with respect to LASs. For TRSM tasks, we achieve a reduction cache miss ratio of about 68%. This is a direct consequence of the change of the tile size since no other optimization is

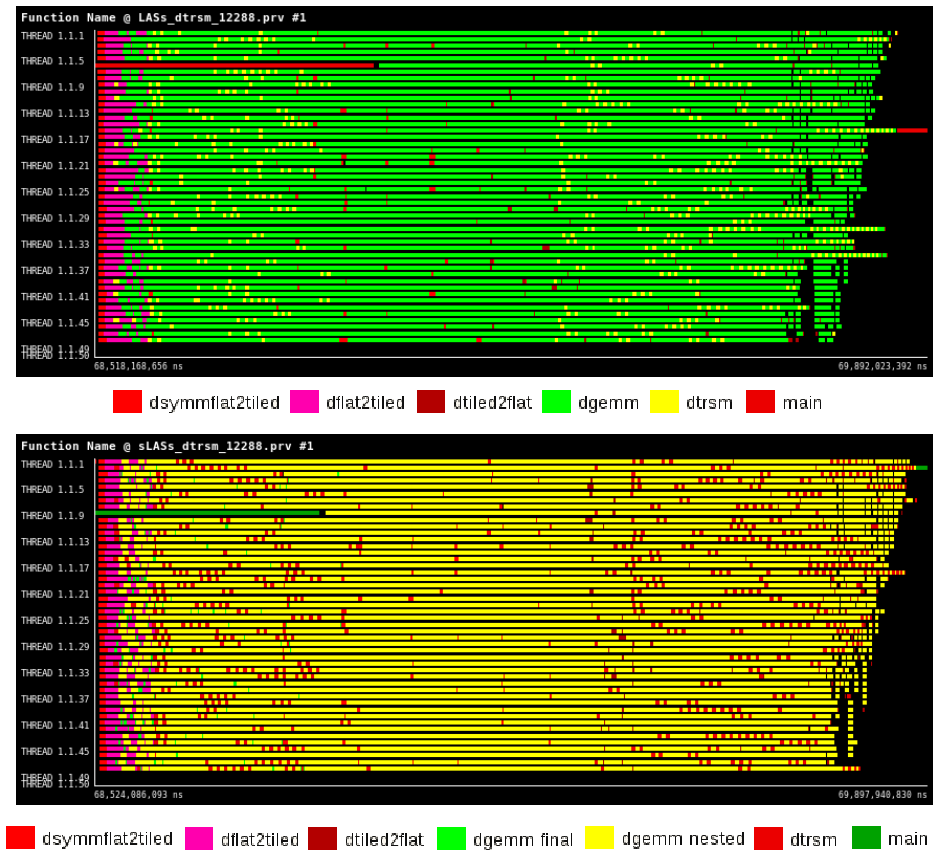


Figure 8: Execution trace for TRSM of LASs (top) and sLASs (bottom) for an input matrix size equal to 12288^2.

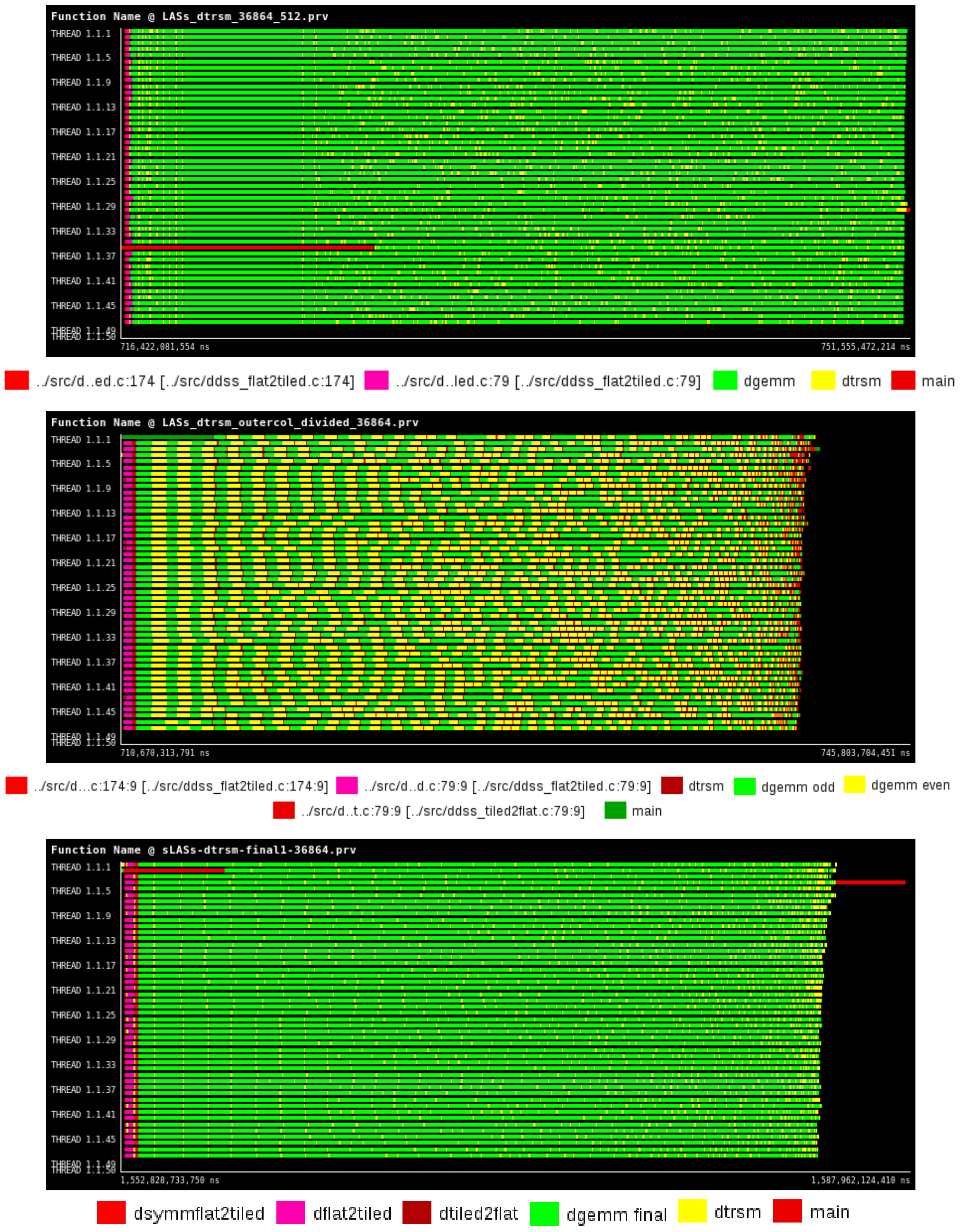


Figure 9: Execution trace for TRSM of LASs (top), LASs-opt (center) and sLASs (bottom) for an input matrix size equal to 36864^2.

applied to these tasks. For GEMM tasks, the reduction is around 95% thanks to the increase of the tile size and the join strategy that we apply.

4.2. Optimization for Small Matrices

After proposing and evaluating some optimizations, which increment the performance on big matrices, in this section, we explore a set of different algorithmic and programming strategies based on tasking to optimize the execution of small and medium size matrices.

4.2.1. Infra-utilization of computational resources

First of all, let us describe the principal problem found when computing relative small matrices, i.e. the infra-utilization. This happens when not all the computational resources are used when computing a problem. In linear algebra problems, this can happen if the matrix to be computed is not big enough with respect to the number of cores.

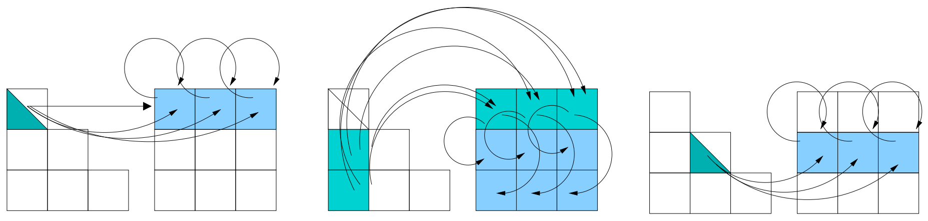


Figure 10: TRSM scheme.

Let us present a simple example for the sake of clarity. For instance, if we have to compute the BLAS-3 level routine TRSM (Figure 10) on a matrix of size 2048^2 and our tile size is equal to 512^2, we will have only 12 GEMM (tasks) to be computed after computed the first iteration of TRSM ("for loop d" of Figure 7). If the target platform has 48 cores, this means that we are only using the 25% of the total available resources.

The infra-utilization presented above can be seen in the next trace, where we execute TRSM on a matrix of size 4096^2 using one node of the MareNostrum supercomputer.

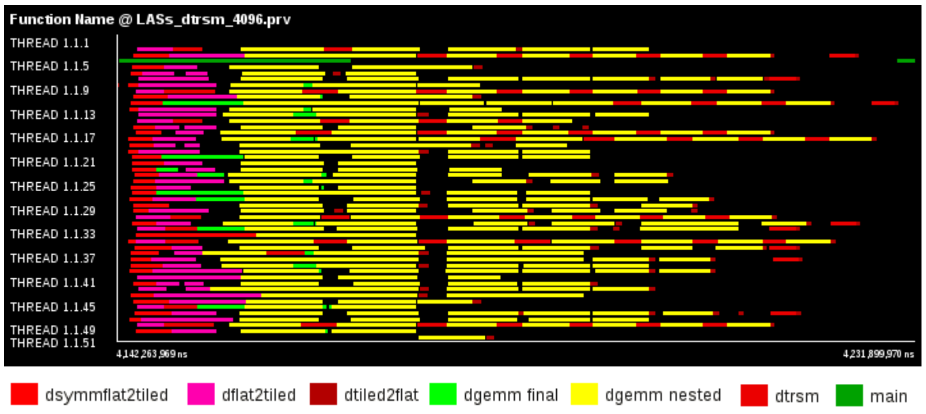
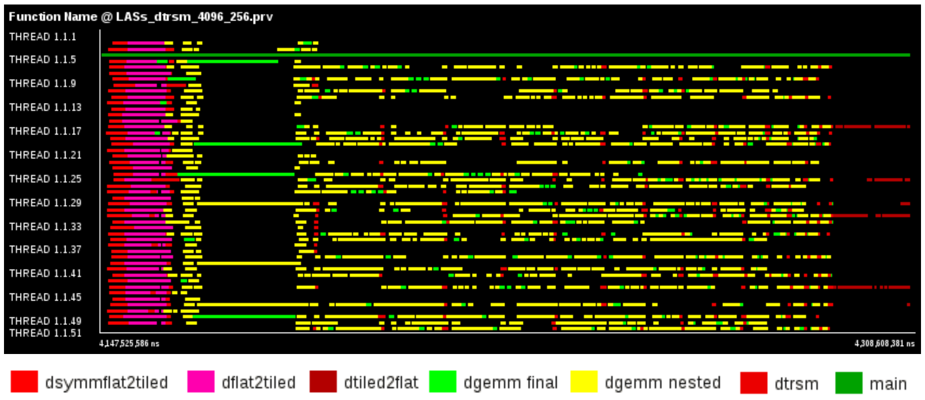


Figure 11: Trace of TRSM on a matrix of size 4096^2.

As shown at the end of the trace, we see that only a few cores are being used. This is something that it is not possible to eliminate completely, since in TRSM, as in most of the BLAS-3 and LAPACK operations, the parallelism (number of tasks) is smaller and smaller at the end of the process. At this point, we propose a set of optimizations to mitigate the infra-utilization found at the end of the trace. This final part of the trace does not suppose a big time in those operations that involve big matrices. However, this part is important on small and medium matrices, as we can see in the trace shown in Figure 11.

One could think that a possible solution is to minimize the tile size (in LASs, the default tile size used in MareNostrum nodes is 512^2). However, with smaller tiles, the tasks take less time, which makes it difficult that the runtime can give work to all cores. Also, reducing the tile size, the IPC per task may be reduced. All this is graphically illustrated in Figure 12 (both traces share the same time scale).

As we see in Figure 12 a smaller tile does not mitigate the infra-utilization problem at the end of the trace, but it increases the problem, due to the consequences aforementioned. In fact, when a big tile size (512^2) is used, we see an important reduction in time.



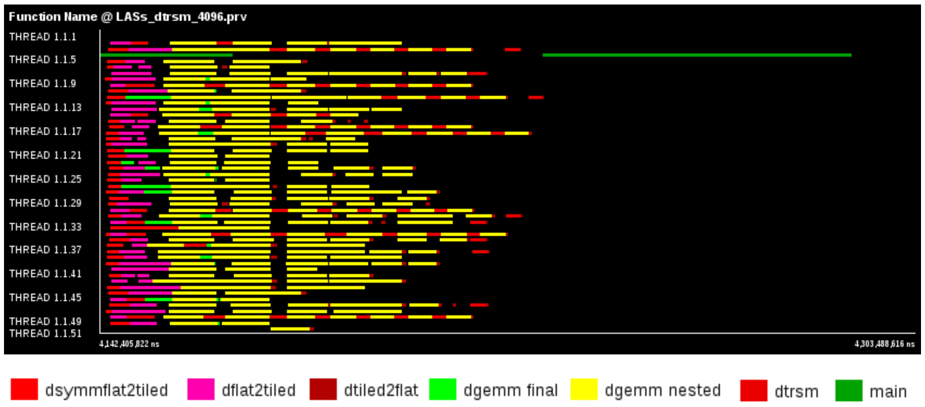


Figure 12: Traces of TRSM on a matrix of 4096^2, using a tile size of 256^2 (top) and 512^2 (bottom)

4.2.2. Approaches

At this point, we present two different approaches that attempt to reduce the infra-utilization on small and medium matrices. Both consist of dividing the computation of one GEMM task into smaller tasks. We focus on GEMM since this represents more than 90% of the total execution time. There are mainly two different ways to do this decomposition, that we call *fine* and *coarse* decomposition. In both, we do not use extra memory to do the decomposition, as it is done at the beginning of the process to pass from flat data-layout to tiled data-layout. We access directly to the tiles.

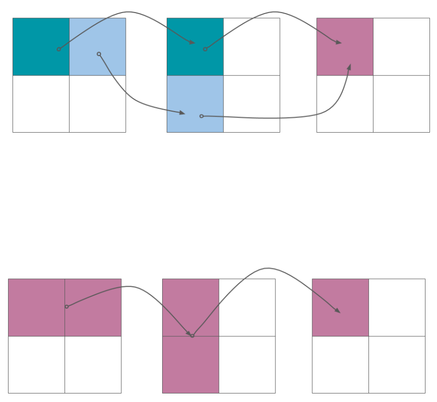


Figure 13: GEMM fine (top) and coarse (bottom) decomposition.

While in the fine-decomposition (Figure 13-top) we have more tasks (16) and the use of dependencies is necessary to guarantee the data-dependencies, the coarse-decomposition (Figure 13-bottom) is in need of a lower number of tasks (only 4) and the dependencies are not necessary. Basically, the main difference, in terms of programming, is found in the location, where the tasks are instantiated and the absence of dependencies in the coarse-decomposition (see Figure 14).

Although the fine-decomposition presents higher parallelism in terms of num-

```
for ( i = 0; i < it; i++) {
    for ( j = 0; j < jt; j++) {
        for ( k = 0; k < kt; k++) {
            #pragma oss task in( TILE_A[i][k] ) \
                             in( TILE_B[k][j] ) \
                             inout( TILE_C[i][j] ) \
                             shared( TILE_A, TILE_B,TILE_C ) \
                             firstprivate( i, j, k )
            dgemm( ... );
        }
    }
}
```

```
for ( i = 0; i < it; i++) {
    for ( j = 0; j < jt; j++) {
        #pragma oss task \
                    shared( TILE_A, TILE_B,TILE_C ) \
                    firstprivate( i, j )
        for ( k = 0; k < kt; k++) {
            dgemm( ... );
        }
    }
}
```

Figure 14: GEMM fine (top) and coarse (bottom) decomposition.

ber of tasks, the truth is that this parallelism cannot be effectively exploited. In fact, due to the data dependencies, we can have only 4 tasks being executed in parallel. Also, the use of dependencies supposes a non-negligible cost for the runtime at this level. Furthermore, as we see in Figure 12-top, the computation of GEMM (tasks) of size 256^2 presents a problem for the runtime, since the granularity of these tasks is too fine, as well as the IPC can be reduced.

Because of all this, we propose and implement the coarse-decomposition. This approach, unlike the fine one, does not have to deal with dependencies. We create 4 tasks per GEMM, which can be executed totally in parallel. As we see in Figure 13-bottom, every task computes two matrices of size 256x512 and 512x256, storing the result in a matrix of 256^2. Therefore, the tasks are more expensive computationally against the tasks executed in the fine-decomposition, which helps the runtime to have bigger tasks and to not reduce the IPC so much.

4.2.3. Implementation

Once the decomposition has been described in the previous section, the other important point about the implementation consists of identifying where the decomposition must be carried out. We previously mentioned that we focus on GEMM, discarding the rest of the routines for now, since the execution of GEMM represents more than the 90% of the execution time.

The infra-utilization happens when we have fewer tasks than number of cores. We can easily compute this, obtaining the number of tasks as follows:

$$(rt(d+1)) \times ct \tag{3}$$

Being rt the number of tiles in the vertical (row) dimension, ct the number of tiles in the horizontal (column) dimension, and d the corresponding level (diagonal block) to be computed (see Figure 15).

This equation must be computed at every level of the execution (every "for loop d" iteration), before executing the set of GEMM.

For the sake of clarity, in Figure 15 we present a pseudo-code of our implementation. It is similar to the code of our original LASs. The differences fall

```
for ( d = 0; d < dt; d++) {
    for ( c = 0; c < ct; c++) {
        #pragma oss task in( TILE_A[d][d] ) \
                         inout( TILE_B[d][c] ) \
                         shared( TILE_A, TILE_B ) \
                         firstprivate( d, C )
        dtrsm( ... );
}
for ( r = d+1; r < rt; r++) {
    for ( c = 0; c < ct; c++) {
        compute_if_nesting(nesting);
        #pragma oss task in( TILE_A[d][r] ) \
                         in( TILE_B[d][c] ) \
                         inout( TILE_B[r][c] ) \
                         shared( TILE_A, TILE_B ) \
                         firstprivate( d, r, c )
        final_dgemm( ..., nesting);
        }
    }
}
```

```
final_dgemm( ..., NESTING){
    if ( NESTING == false ){
        dgemm( ... );
    }
    else{ // Do decomposition via nesting
        for ( i = 0; i < it; i++) {
            for ( j = 0; j < jt; j++) {
                #pragma oss task \
                            shared( TILE_A, TILE_B,TILE_C ) \
                            firstprivate( i, j )
                for ( k = 0; k < kt; k++) {
                    dgemm( ... );
                }
            }
        }
    }
}
```

Figure 15: TRSM pseudo-code with coarse-decomposition.

into two main changes: i) instead of calling to the standard GEMM API after instantiating the task, we call to one intermediate function called final_dgemm. In this function, we pass as parameter the number of tiles (tasks) to be computed in the given d level, following the equation above introduced. ii) into the final_dgemm function, we compute first if we have more tasks than cores or vice-versa. If we have enough number of tasks, the decomposition is not necessary, we compute the GEMM standard API, otherwise if we do not have enough tasks, we compute the coarse-decomposition using nesting.

4.2.4. Performance Evaluation

At this point, we carry out the performance evaluation of the strategies presented in the previous sections. First, we compare the performance (trace) of computing TRSM on a matrix of 4096^2, using only tasks and dependencies (LASs) against a code that makes use of the changes presented in Figure 15.

As graphically illustrated in Figure 16, the use of coarse-decomposition achieves an important reduction in time (close to 20%). In the bottom trace of Figure 16, we can see two dominant colors, the red and the green. While the red color corresponds to GEMM tasks where we do not apply any optimization, the green one corresponds to those tasks where we apply the coarse-decomposition.

It is important to highlight that the use of coarse-decomposition, although it is able to reduce the time by minimizing the effect of the infra-utilization, it achieves a lower IPC per GEMM due to the computation of smaller GEMM, which is masked by better use of the resources.

We extend this study by testing more matrix sizes. In the next graph, we see the performance, in terms of GFLOPS (left axis) and the percentage of performance gain with respect to the original LASs (right axis).

In Figure 17, we see the benefit of using coarse-decomposition (sLASs) with respect to not use it (LASs). We note that a matrix of size 512^2 (default tile-size in our tests) is too small to obtain any benefit using coarse-decomposition. However, on bigger matrices, for instance between 1024^2 – 4096^2, the benefit is important, being almost 40% faster in some cases. The benefit is increasing

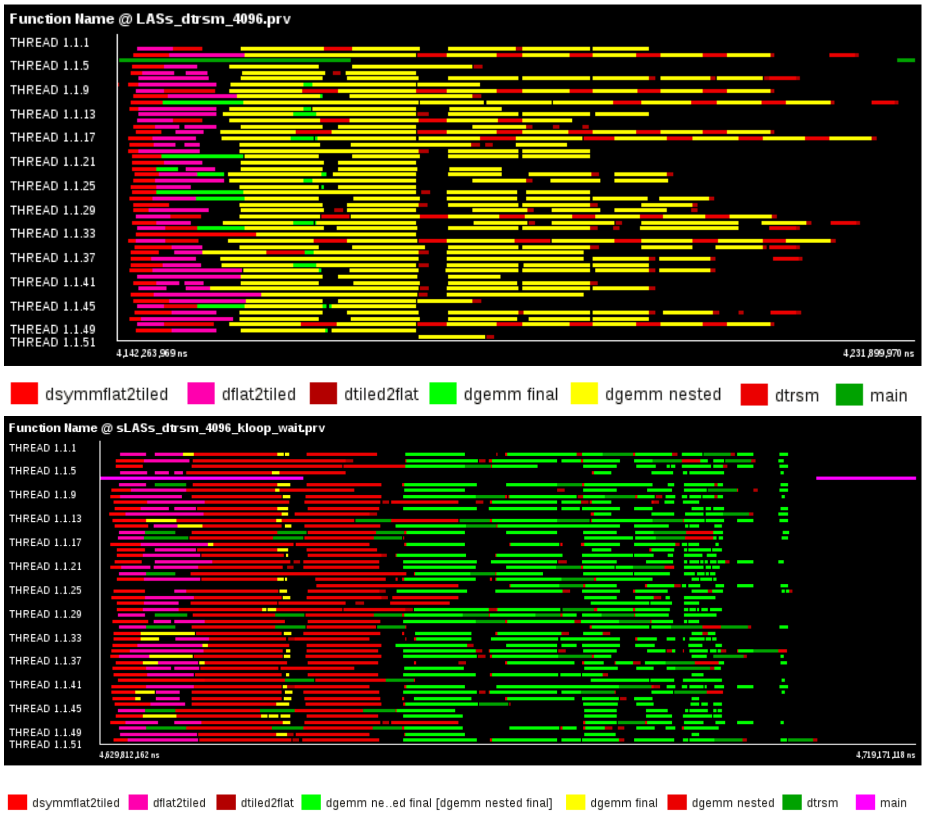


Figure 16: TRSM traces using LASs (top) and the proposed strategies (bottom) on a matrix of 4096^2.

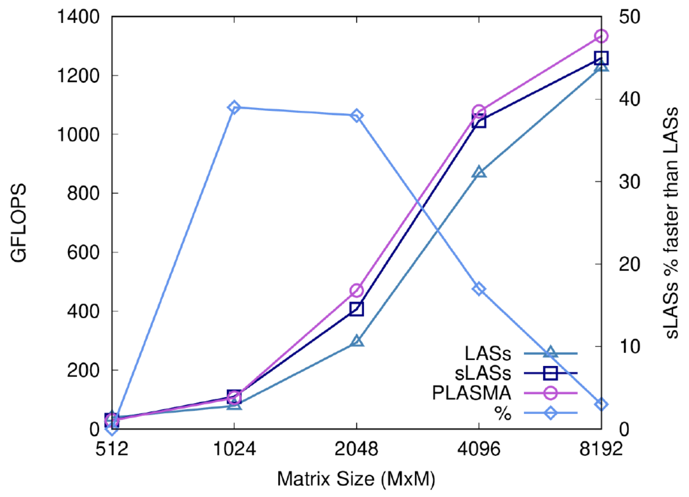


Figure 17: TRSM performance in GFLOPS and percentage of performance gain with respect to the original LASs.

from 2018^2 to 8192^2. The size of the latest one is too big to see any benefit, since the part where we find infra-utilization of the resources is very small compared with the overall time. We have also included the performance of PLASMA (OpenMP) for the same test cases. As shown, although LASs achieves clearly less performance than PLASMA, the strategies presented evaluated in this section are competitive with respect to the GFLOPS attained by PLASMA.

4.2.5. TRSM *decomposition*

In this section, we extend the optimizations for TRSM on small and medium matrices. Although a good performance was achieved with the aforementioned optimizations presented for GEMM tasks, we noted that at the end of the execution, the TRSM tasks became an important bottleneck. These tasks are in the critical path, so, although important improvements were presented by decomposing GEMM tasks into smaller tasks, the dependencies between both type of tasks, TRSM and GEMM, does not allow us to reduce the time at the end of the execution, which is of vital importance to achieve high performance for this kind of matrices.

To deal with these constraints, we implemented an optimization similar to the one implemented on GEMM tasks, but on TRSM tasks. The main idea is the same, decompose the tasks in those parts of the execution where we have an infra-utilization of the computational resources (when the number of tasks to be computed is smaller than the number of cores available). Although the idea is exactly the same as the one implemented for GEMM tasks, the implementation (decomposition) is different (see Figure 18). The TRSM routine can be seen as a set of independent TRSV routines that computes the same input matrix A on different vectors, which compose the matrix B. So the decomposition consists of partitioning the matrix B into sub-matrices. For instance, if we have a matrix A and B of size 512^2, and we want to compute TRSM on these two matrices, we could compute two TRSM in parallel, where the first task computes TRSM using A and the first 256 columns of B and the second task computes TRSM using the same matrix A and the last 256 columns. As commented, both tasks use the same matrix A, this helps us to make an efficient memory hierarchy exploitation during the decomposition.

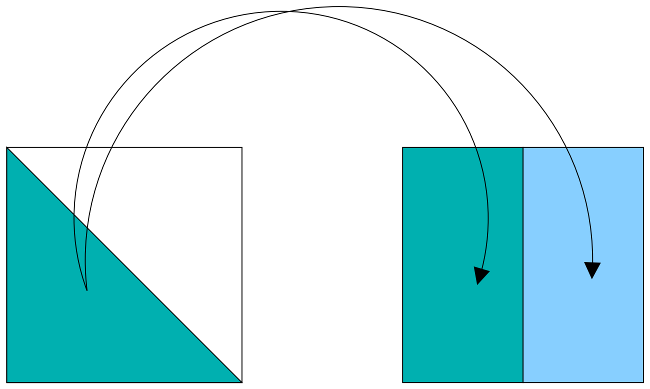


Figure 18: TRSM decomposition

The implementation is simple, when one infra-utilization state is detected, one TRSM task is decomposed into smaller tasks by using nesting. In the nested tasks, it is not necessary the use of dependencies, since these nested tasks are completely independent among them.

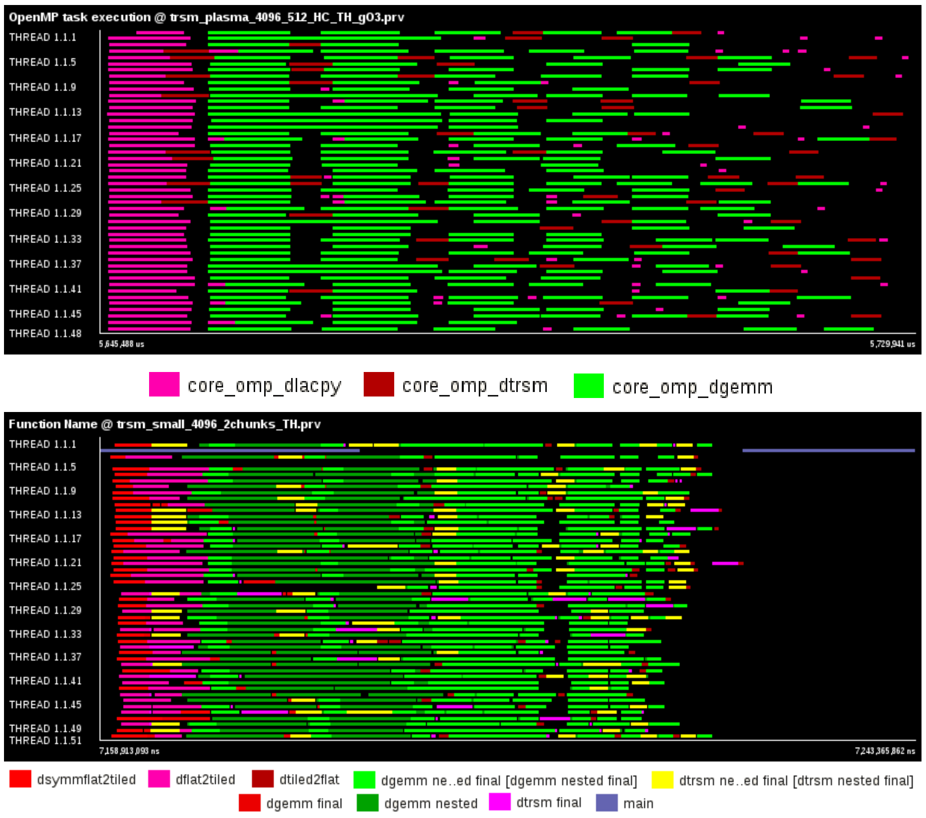


Figure 19: TRSM PLASMA (top) and sLASs (bottom) on a matrix of 4096^2.

To analyze these optimizations, we obtained some traces for a matrix A and B of size 4096^2. For the sake of performance comparison, we also include one trace for PLASMA for the same routine (TRSM) and parameters (Figure 19).

Unlike the results presented for GEMM (Figure 16), and as it is shown in Figure 19, the use of the decomposition of TRSM presented above not only helps to optimize and accelerate the performance of the previous version of sLASs, but it improves the performance presented by PLASMA.

5. npgetrf (non-pivoting LU factorization) routine

Despite the state-of-the-art routine for LU factorization (GETRF) involves pivoting, we have developed the non-pivoting version (NPGETRF) mainly for two reasons: i) on well-conditioned matrices the pivoting is not necessary and ii) for the sake of performance analysis, we want to analyze first the performance of the proposed optimizations without the influence of pivoting. In addition, although the use of pivoting for the solving of linear systems of equations is commonly accepted, we can find multiple problems where the matrices to be solved are well-conditioned, and so computationally expensive operations like pivoting are not necessary. Due to this, it is possible to find multiple implementations in reference libraries, which do not make use of such a technique. Examples of this are: MAGMA library [4], Intel MKL [5], NVIDIA cuSolver [6], NVIDIA cuSparse [7] [15, 16], just to mention a few. Also, the BLKTRI routine of the open-source FISHPACK package [8], makes use of non-pivoting algorithms to [17, 18].

Regarding PLASMA we cannot compare our implementation against this library because this algorithm is not provided. For this reason, results in this sec-

[4]`http://icl.cs.utk.edu/projectsfiles/magma/doxygen/group__group__gesv__nopiv.html`

[5]`https://software.intel.com/en-us/mkl-developer-reference-c-mkl-getrfnpi`

[6]`https://docs.nvidia.com/cuda/cusolver/index.html`

[7]`https://docs.nvidia.com/cuda/cusparse/`

[8]`https://www.netlib.org/fishpack/`

tion are obtained for the LASs and sLASs implementations of the non-pivoting LU factorization.

The implementation of NPGETRF consists of 3 routines: the non-pivoting LU factorization, TRSM and GEMM. For the non-pivoting LU factorization, we make use of the *LAPACKE_mkl_dgetrfnpi* code when the library is linked with MKL. If the library is not linked with MKL, we use NPGETRF, a code implemented by us, which computes the LU factorization with non-pivoting.

The LU factorization on a tiled matrix consists of i) factorizing the first tile of the diagonal, obtaining the L (dark-green in the figure) and U (light-green) matrices on such tile, ii) once L and U are obtained, we compute several TRSM (light-blue) in the corresponding row, using the L matrix, and in the corresponding column, using the U matrix, iii) finally, we compute the so-called "update" step (dark-blue), which consists of multiplying (GEMM) the result of the set of TRSM before mentioned, updating the tiles in the rest of the matrix. We compute on the next tile of the diagonal and the next two steps until all the matrix is computed.

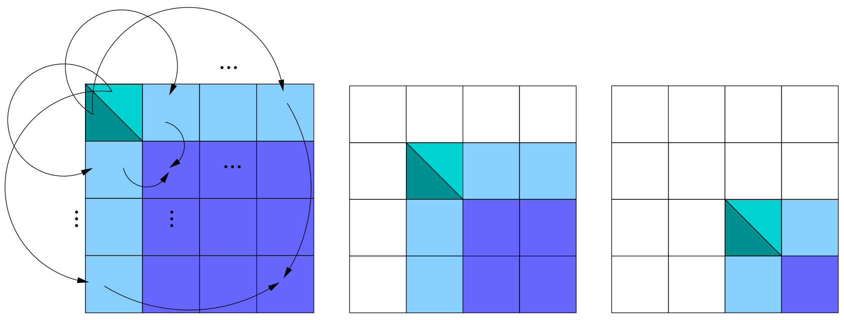


Figure 20: NPGETRF schema.

This process is graphically illustrated in the scheme of Figure 20. As we can see, the number of tasks (tiles) to be computed is reduced along the factorization, in particular, the number of GEMM is reduced step by step. This behavior forces us to use a different strategy than the strategy presented in the previous section for TRSM to implement auto-tunable codes. Unlike TRSM, where the number of columns of tiles to be computed does not change along the execution, in NPGETRF the number of columns of tiles is different depending on the step of the execution. It is because of this that we have to evaluate if there is enough parallelism ($\#columns\ of\ tiles \geq \#cores$) every step. This is illustrated in the next pseudo-code presented in Figure 21.

```
tune_dnpgetrf(dt);
for ( d = 0; d < dt; d++){
    #pragma oss task ...
    dnpgetrf( ... );
    for ( r = d+1; r < rt; r++){
        #pragma oss task ...
        dtrsm( ... );
    }
    for ( c = d+1; c < ct; c++){
        #pragma oss task ...
        dtrsm( ... );
    }
    for ( c = d+1; c < ct; c++){

        is_final = smart_dnpgetrf(c, d, ct);

        #pragma oss task \
        weakin(TILEA[d+1:rt-1][d]) \
        final(is_final)
        for ( r = d+1; r < rt; r++) {
            #pragma oss task \
                in(TILEA[r][d]) ...
            dgemm( ... );
        }
    }
}
```

Figure 21: sLASs code for NPGETRF.

As we can see in Figure 21, *smart_dnpgetrf* is computed in every "for c" iteration. This function computes the *is_final* boolean, which, as in the TRSM implementation, is the parameter of the OpenMP final clause. Depending on the *is_final* boolean, the column of tiles will be computed using one task, when *is_final* is equal to true and then no more tasks are instantiated using the OmpSs weak dependencies and regions, or using multiples tasks (one task per GEMM), when *is_final* is equal to false and so the weak dependencies and regions are not considered (omitted). In this way, we apply a dynamic auto-tuning to LU adapting the number of tasks to be created in each iteration.

We have also explored other optimizations. The first one consists of computing the first column of tiles of the "update" step with *is_final* equal to false. This helps to increase the number of "ready to execute" tasks in the queue of tasks, since as soon as the GEMM task computes the next tile of the diagonal, this tile can be factorized, increasing the number of tasks ready to be computed. The other optimization consists of using priorities. The use of priorities helps to balance the execution. We use the highest priorities on NPGETRF and TRSM tasks. The GEMM tasks involved in the update use a different priority depending on how near are of the left (first) column of the update step. The closer to the left the higher priority. These two optimizations can be seen as an implementation of the well known look-ahead optimization by using priorities and nesting.

Similarly to the analysis performed for TRSM, we evaluate the performance of our auto-tunable NPGETRF code by comparing this with the performance achieved by the implementation of the NPGETRF code of the original LASs, which does not make use of OmpSs weak dependencies, regions and OpenMP final. The auto-tunable code is part of our sLASs library.

First, we present the time consumed by both approaches, LASs and sLASs in Table 3

Table 3: Execution time for NPGETRF in LASs and sLASs.

Time (s)	**6144**	**12288**	**18432**	**24576**	**30720**	**36864**
LASs	**0.19**	**0.93**	**2.90**	**6.63**	13.10	22.40
sLASs	**0.19**	**0.93**	2.96	7.04	**11.45**	**20.28**

As shown, our auto-tunable code is able to achieve a similar performance with respect to the original LASs when the optimizations are not carried out (from 6144^2 to 18432^2), however, when the optimizations are done using weak dependencies, regions and the final clause, the same code is able to adapt the execution to take advantage of the higher parallelism found in these bigger matrices (from 24576^2 to 36864^2), improving performance around 10%.

We also analyzed the performance in terms of GFLOPS to visualize the benefit of using the auto-tunable code more clearly. Results for NPGETRF performance are reported in Figure 22. This plot also includes MKL results as a reference (ATLAS is not included because it does not provide an implementation for the LU factorization without pivoting). Results show that sLASs outperforms or matches MKL performance in all cases.

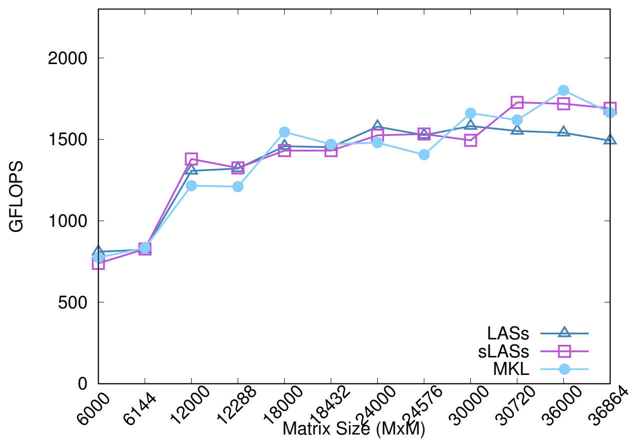


Figure 22: NPGETRF performance in GFLOPS.

To perform a deeper analysis, we take advantage of using the BSC tools Extrae + Paraver to visualize the traces corresponding to two characteristic cases, one for a matrix size equal to 12288^2 and one for a matrix size of 36864^2. We start analyzing the case corresponding to a matrix size of 12288^2 in the traces of Figure 23.

Although the sLASs code (bottom trace) for the NPGETRF routine is in need of computing *smart_dnpgetrf* every iteration of the "for c" loop, as well as, a higher number of tasks must be instantiated, this does not present an important overhead with respect to the original LASs when the optimizations (*is_final* = *false*) are not computed.

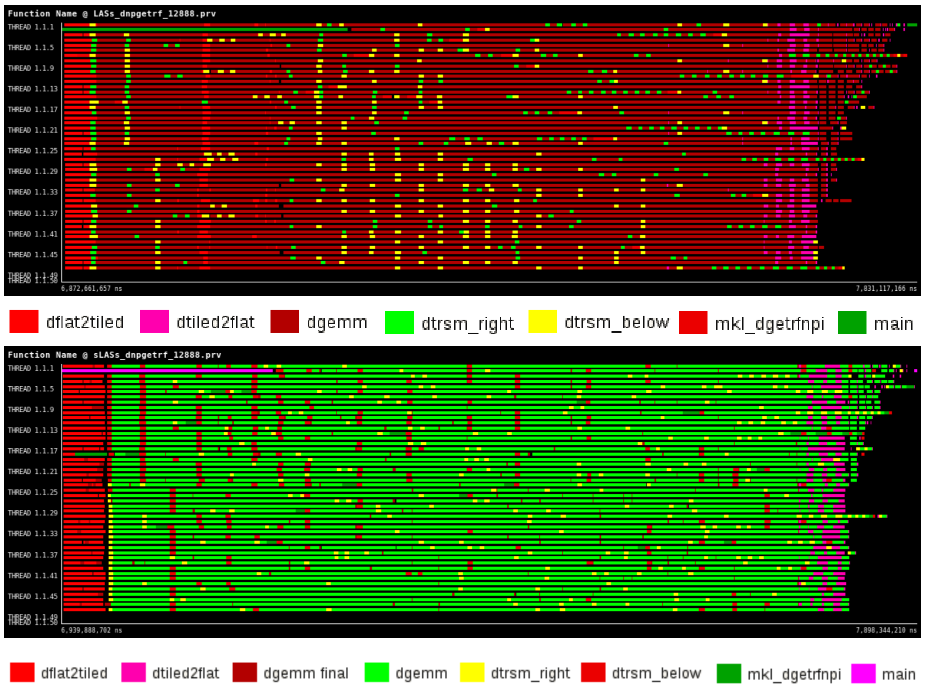


Figure 23: NPGETRF execution traces for LASs and sLASs. Input matrix size is 12288^2.

However, as we see in the traces of Figure 24, the benefit of the same code (sLASs) is notable when applying the optimizations (*is_final* $=$ *true*), using weak dependencies, regions and final.

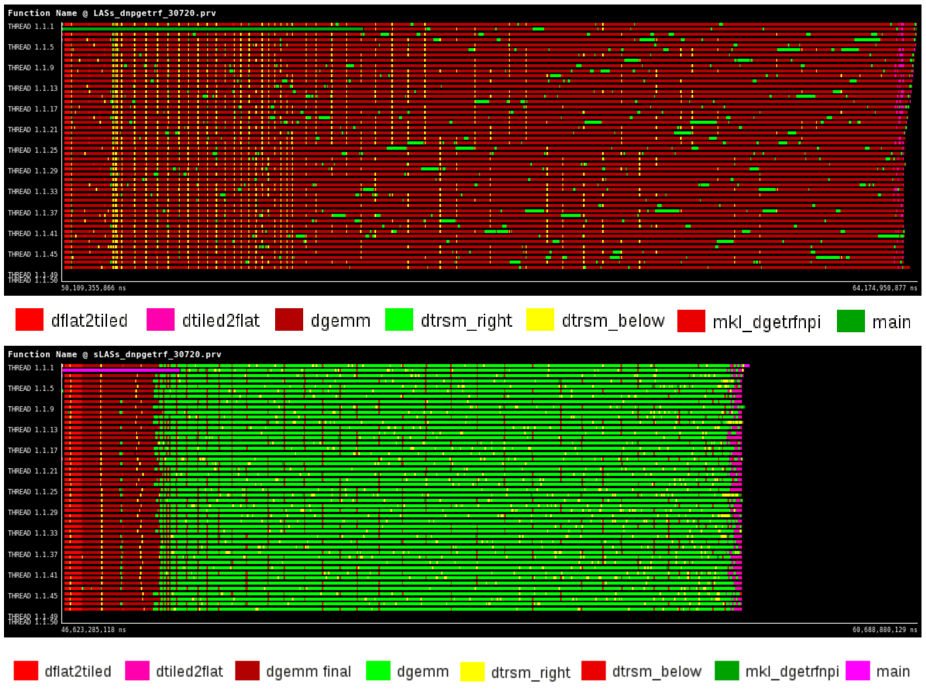


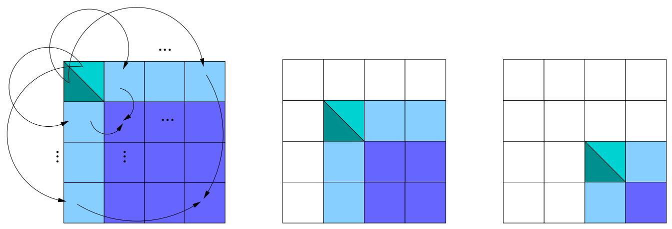
Figure 24: NPGETRF execution traces for LASs and sLASs. Input matrix size is 30720^2.

In the trace which corresponds to sLASs-NPGETRF (bottom trace), we can distinguish two dominant colors, red and green. While the first one corresponds to those tasks which make use of the OmpSs weak dependencies and regions, the green represents the rest of tasks that do not use these OmpSs features. The optimizations only can be carried out on those regions with enough parallelism (first steps of the LU factorization). Due to this, we see the "red" tasks only at the beginning of the trace.

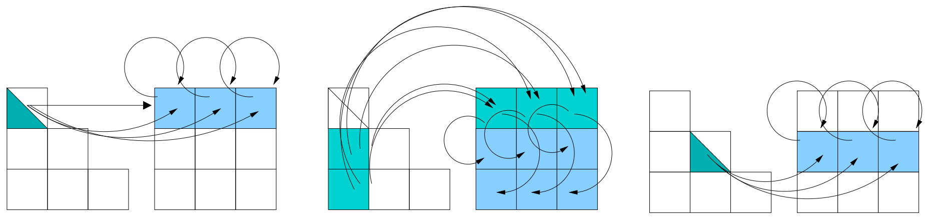
6. npgesv (non-pivoting LU solve) routine

This routine basically consists of joining the two routines described in the previous points, NPGETRF and TRSM, to solve a linear system of equations, $AX = B$. First, we compute the NPGETRF to decompose the matrix A into L and U, $A = LU$. After this, we solve the next equation using the lower triangular matrix L, $LY = B$. Finally, once Y is computed, we solve the next equation $UX = Y$ to obtain the final solution X. This is graphically illustrated in Figure 25.

A=LU:



LY=B:



UX=Y:

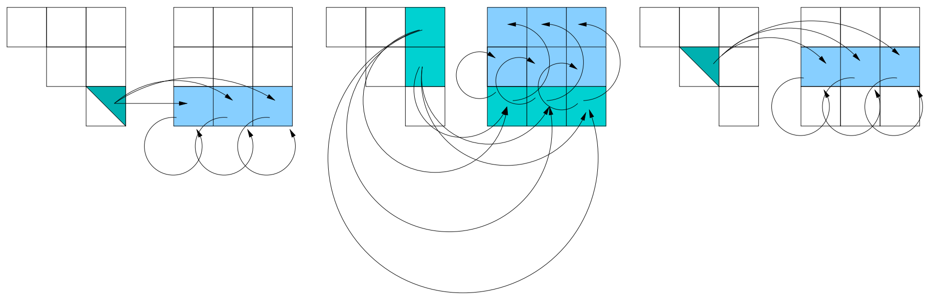


Figure 25: NPGESV schema.

It is important to note that the last TRSM ($UX = Y$) cannot be computed until the previous TRSM ($LY = B$) has been completely computed. This does not allow us to overlap the execution of these two consecutive TRSM tasks.

Like in the two previous routines, we have also implemented an auto-tunable code for NPGESV. This code is part of the sLASs library. The implementation of NPGESV in the sLASs consists basically of combining the two previous codes. Probably the most interesting contribution of this code is the use of static tuning (TRSM) and dynamic tuning (NPGETRF) in one single code. This is implemented by using the two "smart" functions, *smart_dtrsm* and *smart_dnpgetrf*. While *smart_dnpgetrf* must be computed every step of the factorization, as described in the previous point, the *smart_dtrsm* is only necessary to be computed once at the beginning, since the parallelism (number of tile-columns) in TRSM does not change along the execution, as it is graphically shown in Figure 25.

For the sake of clarity, we also include a simple pseudo-code (Figure 26) which can help to understand the main characteristics of the NPGESV code.

We evaluate the performance of our auto-tunable code, which makes use of the OmpSs weak dependencies and regions and the OpenMP final with respect to a code that does not make use of these features, only using tasks + data dependencies. This last code is part of the LASs library [9]. Like in NPGETRF code, in NPGESV we keep the use of priorities for better execution. First, we analyze the time consumed by both codes, LASs (without using weak dependencies, regions, and final) and sLASs (using weak dependencies, regions, and final). Table 4 presents execution time results for NPGESV. We can see that execution time is reduced in all cases, being remarkable for big matrices, where reduction reaches 15% with respect to the original LASs.

The benefit of sLASs with respect to LASs is shown in all the matrices tested, however the bigger matrix size the higher benefit, achieving a reduction of about 12 seconds on the biggest matrix evaluated. This is also in concordance with the analysis in terms of GFLOPS, as can be seen in Figure 27.

[9]https://pm.bsc.es/mathlibs/lass

```
tune_dnpgesv(dt);
// DNPGETRF
for ( d = 0; d < dt; d++){
    #pragma oss task ...
    dnpgetrf( ... );
    for ( r = d+1; r < rt; r++){
        #pragma oss task ...
        dtrsm( ... );}
    for ( c = d+1; c < ct; c++){
        #pragma oss task ...
        dtrsm( ... );}
    for ( c = d+1; c < ct; c++){
        is_final = smart_dnpgetrf(c, d, ct);
        #pragma oss task \
        weakin(TILEA[d+1:rt-1][d]) ...
        final(is_final)
        for ( r = d+1; r < rt; r++) {
            #pragma oss task \
            in(TILEA[r][d]) ...
            dgemm( ... );
        }
    }
}
// DTRSM-Lower
is_final = smart_dtrsm(ct);
for ( d = 0; d < dt; d++){
    for ( c = 0; c < ct; c++){
       #pragma oss task ...
        dtrsm( ... );
    }
    for ( c = 0; c < ct; c++) {
        #pragma oss task \
        weakinout (TILEB[d:rt-1][c]) ...
        final(is_final)
        for ( r = d; r < rt; r++){
           #pragma oss task \
           inout(TILEB[r][c]) ...
           dgemm( ... );
        }
    }
}
// DTRSM-Upper
for ( d = dt-1; d >= 0; d--){
    for ( c = 0; c < ct; c++){
        #pragma oss task ...
        dtrsm( ... );}
    for ( c = 0; c < ct; c++) {
        ...
    }
}
```

Figure 26: sLASs code for NPGESV.

Table 4: Execution time for NPGESV in LASs and sLASs.

Time (s)	**6144**	**12288**	**18432**	**24576**	**30720**	**36864**
LASs	0.46	3.32	11.67	27.12	53.90	94.43
sLASs	**0.44**	**3.29**	**11.45**	**26.69**	**46.04**	**82.74**

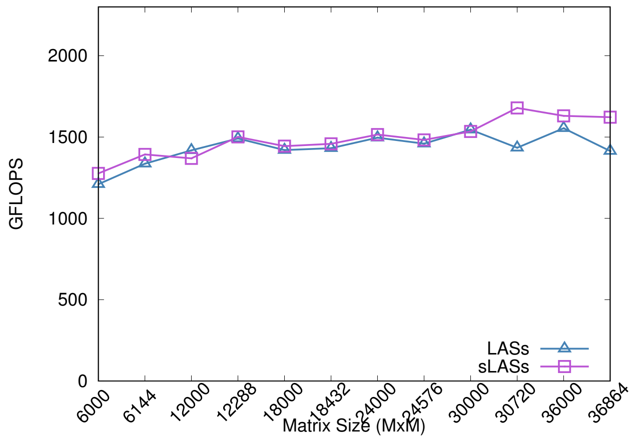
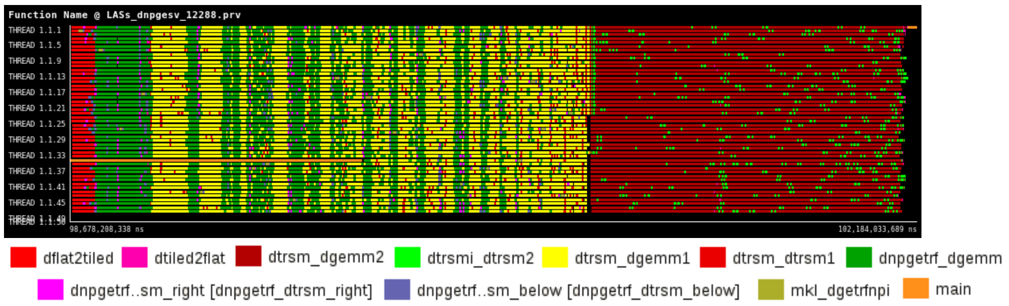


Figure 27: NPGESV performance in GFLOPS for LASs and sLASs.



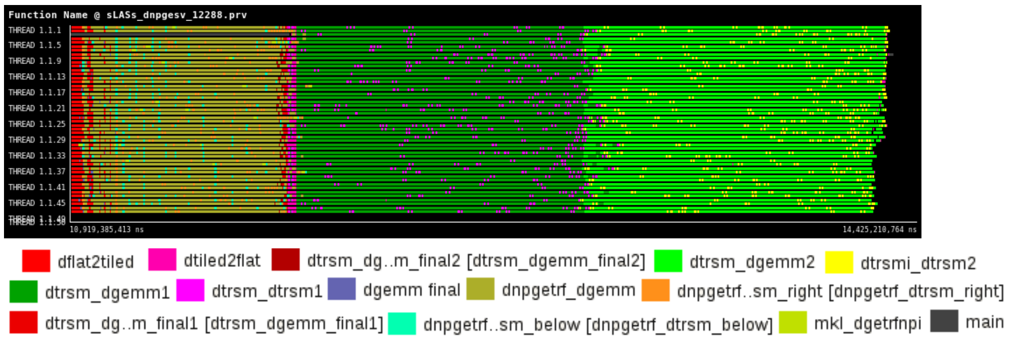


Figure 28: NPGESV execution traces for LASs and sLASs. Input matrix size is 12288^2.

To carry out a deeper analysis we have studied two characteristic test cases, one for a matrix size equal to 12288^2 and one for a size of 36864^2. We start with the smallest matrix size, 12288^2. In Figure 28, we see the traces corresponding to the LASs (top) and the sLASs (bottom). In both traces, we note that the second TRSM (red in the top trace and light-red in the bottom trace) is not overlapped with computation of the first TRSM or NPGETRF. This is due to the data dependencies of the LU solve, since the second TRSM cannot be computed until the first TRSM has been completely computed. In the bottom trace, we also note that the first TRSM (in dark-green) is not overlapped with the computation of the NPGETRF (in dark-yellow). This behavior is not shown in the LASs trace. This is the consequence of using priorities, since the tasks involve in the NPGETRF have a higher priority than the tasks involved in the execution of the two TRSM. This difference has not important consequences in terms of performance, being the sLASs execution slightly better than the LASs counterpart for small and medium matrices.

Finally we analyze the LASs (top) and sLASs (bottom) traces for a matrix size of 36864^2. We see the same behavior shown in the previous traces, i. e. while the tasks of NPGETRF and the first TRSM are overlapped in the top trace, these are not overlapped in sLASs, since this last uses priorities. It is also possible to see the notable reduction in time between both traces. In the bottom trace (sLASs), we are able to see the two different phases of the NPGETRF, where the join of GEMM is computed (in dark-green) and not computed (in purple).

6.1. Improving NPGESV *for small matrices*

As expected, when applying the previous optimizations, the performance increase is mainly observed for big matrices. In order to improve performance for small matrices, we also apply the same optimizations that were carried out for TRSM; that is, creating smaller nested tasks when parallelism is not enough to feed all the available cores in both cases, for GEMM and TRSM tasks.

Figure 30 shows performance for small matrices (up to 6144^2) when running NPGESV. Results show that performance is improved in all cases between 2%

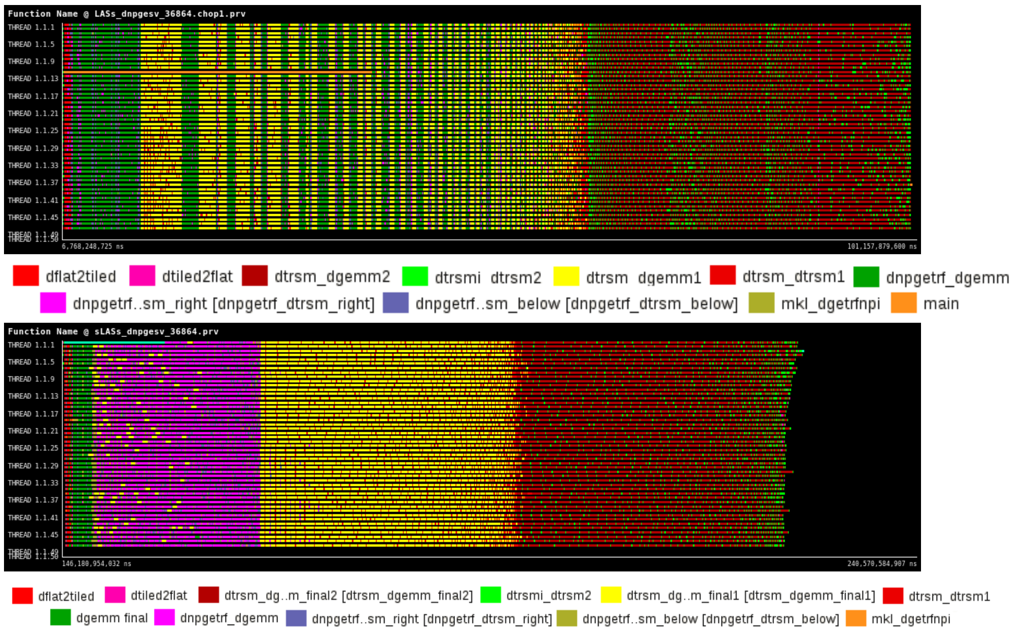


Figure 29: NPGESV execution traces for LASs and sLASs. Input matrix size is 36864^2.

and 36%. Improvements are less effective for very small matrices (e.g. 512^2), where the overhead introduced by the runtime has a higher impact. This effect is also seen for the largest matrix in this set (6144^2). In this case, the source of the reduction in performance is the fact that the percentage of code where the optimizations designed for small matrices can be applied is smaller than in the previous cases, so no effect is seen.

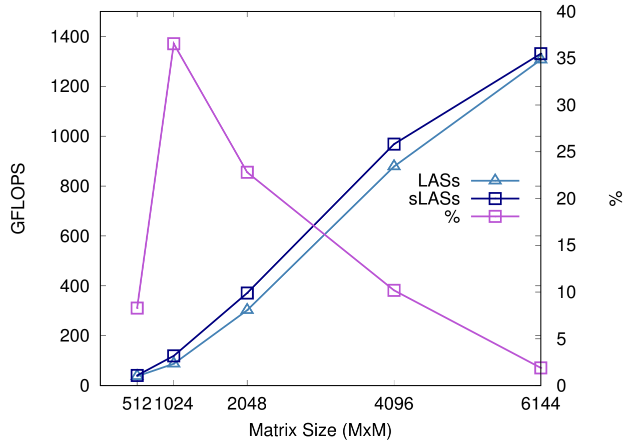


Figure 30: NPGESV performance results for LASs and sLASs when input matrices are small (512^2 to 6144^2)

7. Conclusions

In this work we have presented auto-tunable versions of the BLAS-3 TRSM routine and the LAPACK routines NPGETRF and NPGESV included in LASs library. These new versions of the code, implemented by means of OmpSs-2 features (weak dependencies, regions and the final clause), are part of the first prototype of sLASs library, a novel library for auto-tunable codes for linear algebra operations based on LASs library. At the sight of the results, the use of the OmpSs-2 features presents an improvement in terms of execution time against the original LASs library and the OpenMP reference library PLASMA. The improvement is especially remarkable with respect to ATLAS library and

Intel MKL. As overview, these codes are able to reduce the execution time in about 18% on big matrices (in comparison to LASs), by increasing the IPC on GEMM and reducing the time of task instantiation. For a few medium matrices, benefits are also seen. For small matrices and a subset of medium matrices, specific optimizations allow to increase the degree of parallelism in both, GEMM and TRSM tasks. These strategies achieve an increment in performance of up to 40%. When comparing with Intel MKL, the optimized codes achieve similar performance for TRSM and outperform it for NPGETRF.

Acknowledgment

This project has received funding from the Spanish Ministry of Economy and Competitiveness under the project Computación de Altas Prestaciones VII (TIN2015- 65316-P), the Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programació i Entorns d'Execució Parallels (2014-SGR-1051), and the Juan de la Cierva Grant Agreement No IJCI-2017- 33511. We also acknowledge the funding provided by Fujitsu under the BSC-Fujitsu joint project: Math Libraries Migration and Optimization.

References

[1] OmpSs project home page, `http://pm.bsc.es/ompss`.

[2] G. Llort, H. Servat, J. González, J. Giménez, J. Labarta, On the usefulness of object tracking techniques in performance analysis, in: 2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2013, pp. 1–11. `doi:10.1145/2503210.2503267`.

[3] PLASMA project home page, `http://icl.cs.utk.edu/plasma`.

[4] P. Valero-Lara, S. Catalán, X. Martorell, J. Labarta, 27th euromicro international conference on parallel, distributed and network-based processing,

PDP 2019, pavia, italy, february 13-15, 2019, IEEE, 2019.
URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8663972

[5] J. M. Perez, V. Beltran, J. Labarta, E. Ayguadé, Improving the integration of task nesting and dependencies in OpenMP, in: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017, pp. 809–818. doi:10.1109/IPDPS.2017.69.

[6] J. M. Perez, R. M. Badia, J. Labarta, Handling task dependencies under strided and aliased references, in: Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10, ACM, New York, NY, USA, 2010, pp. 263–274. doi:10.1145/1810085.1810122.
URL http://doi.acm.org/10.1145/1810085.1810122

[7] R. Vargas, E. Quinones, A. Marongiu, OpenMP and timing predictability: A possible union?, in: Proceedings of the 2015 Design, Automation; Test in Europe Conference; Exhibition, DATE '15, EDA Consortium, San Jose, CA, USA, 2015, pp. 617–620.
URL http://dl.acm.org/citation.cfm?id=2755753.2755893

[8] J. A. Gunnels, F. G. Gustavson, G. M. Henry, R. A. van de Geijn, Flame: Formal linear algebra methods environment, ACM Trans. Math. Softw. 27 (4) (2001) 422–455. doi:10.1145/504210.504213.
URL http://doi.acm.org/10.1145/504210.504213

[9] R. C. Whaley, J. Dongarra, Automatically Tuned Linear Algebra Software, in: Ninth SIAM Conference on Parallel Processing for Scientific Computing, 1999, cD-ROM Proceedings.

[10] Netlib.org, Blas, http://www.netlib.org/blas.

[11] Netlib.org, Lapack, http://www.netlib.org/lapack.

[12] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, M. Zounon, The design and performance of batched BLAS on

modern high-performance computing systems, Procedia Computer Science 108 (2017) 495 – 504, international Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland. doi:https://doi.org/10.1016/j.procs.2017.05.138.
URL http://www.sciencedirect.com/science/article/pii/S1877050917307056

[13] P. Valero-Lara, I. Martnez-Prez, S. Mateo, R. Sirvent, V. Beltran, X. Martorell, J. Labarta, Variable batched DGEMM, in: 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), 2018, pp. 363–367. doi:10.1109/PDP2018.2018.00065.

[14] S. Catalán, J. R. Herrero, E. S. Quintana-Ortí, R. Rodríguez-Sánchez, R. A. van de Geijn, A case for malleable thread-level linear algebra libraries: The LU factorization with partial pivoting, CoRR abs/1611.06365. arXiv:1611.06365.
URL http://arxiv.org/abs/1611.06365

[15] P. Valero-Lara, I. Martínez-Perez, R. Sirvent, X. Martorell, A. J. Peña, NVIDIA gpus scalability to solve multiple (batch) tridiagonal systems implementation of cuThomasBatch, in: Parallel Processing and Applied Mathematics - 12th International Conference, PPAM 2017, Lublin, Poland, September 10-13, 2017, Revised Selected Papers, Part I, 2017, pp. 243–253.

[16] P. Valero-Lara, I. Martínez-Pérez, R. Sirvent, X. Martorell, A. J. Peña, cuThomasBatch and cuThomasVBatch, CUDA routines to compute batch of tridiagonal systems on NVIDIA GPUs, Concurrency and Computation: Practice and Experience 30 (24).

[17] P. Valero-Lara, A. Pinelli, J. Favier, M. P. Matias, Block tridiagonal solvers on heterogeneous architectures, in: IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, ISPA '12, 2012, pp. 609–616.

[18] P. Valero-Lara, A. Pinelli, M. Prieto-Matias, Fast finite difference Poisson solvers on heterogeneous architectures, Computer Physics Communications 185 (4) (2014) 1265 – 1272. `doi:10.1016/j.cpc.2013.12.026`.