Int J Parallel Prog (2014) 42:408–434
DOI 10.1007/s10766-013-0249-6

Empirical Installation of Linear Algebra Shared-Memory Subroutines for Auto-Tuning

Jesús Cámara · Javier Cuenca · Domingo Giménez · Luis Pedro García · Antonio M. Vidal

Received: 11 October 2012 / Accepted: 30 April 2013 / Published online: 23 May 2013

Abstract The introduction of auto-tuning techniques in linear algebra shared-memory routines is analyzed. Information obtained in the installation of the routines is used at running time to take some decisions to reduce the total execution time. The study is carried out with routines at different levels (matrix multiplication, LU and Cholesky factorizations and linear systems symmetric or general routines) and with calls to routines in the LAPACK and PLASMA libraries with multithread implementations. Medium NUMA and large cc-NUMA systems are used in the experiments. This variety of routines, libraries and systems allows us to obtain general conclusions about the methodology to use for linear algebra shared-memory routines auto-tuning. Satisfactory execution times are obtained with the proposed methodology.

J. Cámara (✉) · D. Giménez
Departamento de Informática y Sistemas, University of Murcia, Murcia, Spain
e-mail: jcm23547@um.es

D. Giménez
e-mail: domingo@um.es

J. Cuenca
Departamento de Ingeniería y Tecnología de Computadores,
University of Murcia, Murcia, Spain
e-mail: jcuenca@um.es

L. P. García
Servicio de Apoyo a la Investigación Tecnológica,
Technical University of Cartagena, Cartagena, Spain
e-mail: luis.garcia@sait.upct.es

A. M. Vidal
Departamento de Sistemas Informáticos y Computación,
Technical University of Valencia, Valencia, Spain
e-mail: avidal@dsic.upv.es

Keywords Linear algebra libraries · Linear algebra routines · Empirical installation · Shared-memory · Auto-tuning

1 Introduction

The appearance of multicore and cc-NUMA systems has led to the need to develop optimized software for this type of systems. Software optimization techniques are used in parallel routines to decide how to execute them with low execution times. Decisions are taken at running time as a result of a learning phase which is carried out when the routine is installed in the system. During installation the values of some system parameters in a theoretical model of the execution time of the routines are estimated, or some empirical study of the behavior of the routine in the system is carried out experimentally. The decisions taken at running time depend on the type of computational system used. Some could be: selecting the appropriate number of threads to use at each level of parallelism; how to assign processes to processors; or selecting the correct block size in algorithms by blocks.

Linear algebra routines are the basic computational kernels used in many scientific applications, and it is especially interesting to optimize and auto-optimize them, because any improvement will produce a reduction of the execution time of the high-level scientific applications where they are used. Furthermore, the auto-tuning techniques used for linear algebra routines can be extended to routines in different fields [4,15].

In this work, previous ideas for installing multithreaded basic linear algebra routines in large cc-NUMA systems [6] are combined and extended. In [6], auto-tuning is carried out by applying installation techniques to the BLAS-3 matrix multiplication routine (`dgemm`), which constitutes the basic subroutine for many computational routines. Here, these ideas are extended for higher-level routines. The LU and Cholesky factorizations and their multithread implementations in LAPACK [3] and PLASMA [24] are considered, along with symmetric and general linear systems routines based on LDL^T and LU factorizations. This variety of routines and systems allows us to draw general conclusions about the installation and auto-tuning of shared-memory linear algebra subroutines.

The rest of the paper is organized as follows. Section 2 discusses the background of auto-tuning and its application to parallel linear algebra routines. Section 3 summarizes the auto-tuning methodology we use for linear algebra routines in NUMA systems. Section 4 analyzes the application of the auto-tuning methodology to MKL [14] implementations of the matrix multiplication and matrix factorizations. In Sect. 5 the methodology is extended to higher level routines (solution of multiple linear systems), and Sect. 6 shows some experiments of the application of the methodology to PLASMA routines. Finally, conclusions and future research lines are offered in Sect. 7.

2 Background

Advanced computational systems have a complex structure which complicates their efficient use. Scientists and engineers use these advanced systems to solve problems of high computational cost, and they need optimized routines with which the solution times of their problems are reduced. The work for optimizing the code by hand can take several weeks or months and it requires deep knowledge of several disciplines, like computer architecture, programming and debugging tools, and mathematical software. Furthermore, the optimization task performed for a specific platform may not necessarily be suitable for other platforms. Such a diverse working environment has triggered important changes in the traditional way of optimizing the software for scientific calculations, with the goal of following the pace of both the user needs and the new hardware developments.

Automatic optimization techniques have emerged as valuable tools that provide scientific software with environment adaptation capacity [21]. The techniques are applied to routines in different fields like discrete Fourier transform [11], digital signal processing transforms [25], Fast Wavelet transform [4] or quantum chemistry [26]. In many scientific problems, the basic computational components are linear algebra routines, and so auto-tuning techniques are especially interesting in linear algebra, and there are numerous projects in this field: ATLAS [29] optimizes computational kernels for dense linear algebra, SPARSITY [13] works with sparse linear algebra kernels, ABCLib-DRSSED [17] with routines for obtaining eigenvalues, etc.

In parallel linear algebra the auto-tuning is carried out in different ways and for different computational environments. The libraries are optimized in the installation process for shared memory machines [29] or for message-passing systems [8,9]. The routines can adapt to the conditions of the system at a particular moment [23]. Poly-algorithms [20] and poly-libraries [2] can be used.

Auto-tuning is usually through theoretical models of the execution time of the routine or through empirical analysis of the behavior of the routine based on exhaustive testing. The approach in FAST [7] is an extensive benchmark followed by a polynomial regression to find optimal parameters for different routines in homogeneous and heterogeneous environments. Polynomial regression is used in [28] to decide the most appropriate version from variants of a routine, and a black-box pruning method is introduced to reduce the enormous implementation space. In the FIBER approach [16] the execution time of a routine is approximated by fixing one parameter (problem size) and varying the other (unrolling depth for an outer loop); a set of polynomial functions of degrees 1 to 5 is generated and the best is selected; the values provided by these functions for different problem sizes are used to generate another function where the second parameter is now fixed and the first one is varied. The Incremental Performance Parameter Estimation is introduced in [27], in which the estimation of the theoretical model by polynomial regression is started from the least sampling points and incremented dynamically to improve accuracy. In [19] the number of sampling points is reduced by using a predetermined shape of the curve that represents the execution time, and the concept of "speed band" is introduced to represent the inherent fluctuations in the speed due to changes in load.

Nowadays, with the increased complexity of computational environments, which are hierarchical and heterogeneous in different ways, it is necessary to adapt the techniques previously developed for simpler environments to these more complex systems. Thus, basic linear algebra routines are being redesigned for platforms based on multicore processors [5] and for heterogeneous/hybrid architectures [1], including GPUs

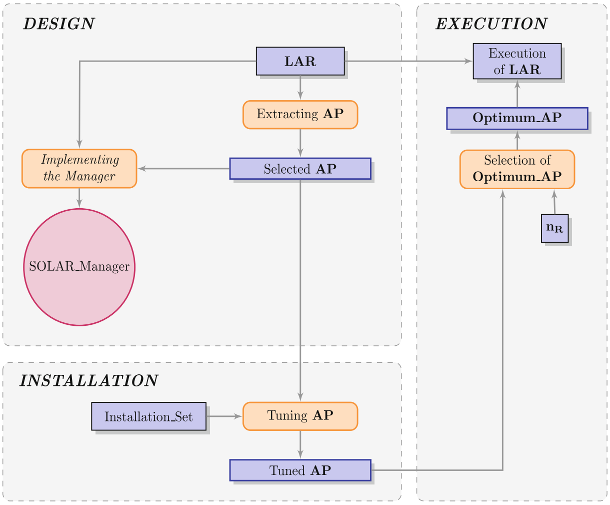


Fig. 1 Scheme of the empirical auto-tuning methodology

[18,22]. For these routines, which combine different parallel programming paradigms and several parallelism levels, it is more difficult to obtain satisfactory theoretical models of the execution time, and the auto-tuning approach based on empirical estimations through exhaustive testing is preferable. In this work an adaptation of this auto-tuning method to NUMA platforms is proposed in order to select automatically the number of threads to use in the different parallelism levels and the block sizes in algorithms working by blocks.

3 The Auto-Tuning Methodology

Implementations of shared-memory linear algebra routines are normally not very scalable. This produces a degradation of the performance in large cc-NUMA systems [10]. To improve the scalability of the routines, the auto-tuning methodology explained in [6] for the `gemm` routine can be extended to higher-level routines. The goal is to select the most appropriate number of threads at each level of parallelism, together with the values of other algorithmic parameters, like the block size (or sizes) in algorithms by blocks. The methodology of [9] is adapted to the empirical installation in NUMA systems. It is divided in three phases, which are represented in Fig. 1:

- **Design phase**. Initially, when the routine (LAR) is designed, a model of the execution time is developed, and this model is used in the subsequent phases. When an implemented routine or library is used and the model of the execution time is not available, it can be empirically estimated. Auto-tuning of linear algebra routines in large cc-NUMA based on theoretical models is a difficult task, which has been analyzed in [10], but with only partially satisfactory results. The approach used here for linear algebra routines optimization in NUMA systems is based on the empirical study of the behavior of the routine, which could in turn be combined with modelling techniques. In any case, the algorithmic parameters (AP) influencing the execution time of the routine are determined, and they are included in the theoretical model (if available) or they are used to design and implement an engine (the Self-Optimized Linear Algebra Routine Manager, SOLAR_Manager) with which satisfactory values of the algorithmic parameters are determined in the subsequent phases. Typical algorithmic parameters in linear algebra routines in cc-NUMA are:

- The number of threads at each parallelism level, which can be, for example, the number of OpenMP threads and of MKL threads in a two-level routine which combines OpenMP and MKL parallelism.
- The block size in algorithms by blocks, like those in LAPACK and some routines of PLASMA, or inner and outer block sizes in algorithms with two levels of blocks, like some PLASMA routines.

Furthermore, there may be influences between the optimum values of the algorithmic parameters. For example, the optimum block size changes with the number of threads, and so the values of the algorithmic parameters must be estimated *en bloc*, and the SOLAR_Manager must be prepared for a combined search for optimum values. This combined search process is not normally considered in linear algebra libraries, but the influence is especially important in large systems, with many cores and where many threads, at different levels, should be started to fully exploit the computational system.

- **Installation phase**. In the installation of the routine, when using the model of the execution time, some system parameters in the model are experimentally estimated for the specific platform in question. The model must be adapted to the particular characteristics of large NUMA platforms, where there is a shared memory space with non uniform data access time, which makes it difficult to develop an accurate model [10]. As an alternative, we consider an installation without the use of a theoretical model (because the code of the routine is not available, or due to the difficulty in obtaining an accurate model for a complex system). The SOLAR_Manager is ready to conduct some experiments with which the behavior of the routine in the system is analyzed. In this learning phase, experiments are carried out for some significant problem sizes (Installation_Set), and to search for the values of the algorithmic parameters (Tuned AP) with which the lowest experimental execution time is obtained for each one of the Installation_Set sizes. For example, experiments are conducted for different problem sizes, number of threads at the different parallelism levels, and block size or sizes in routines working by blocks. In large NUMA systems, due to the memory hierarchy and the affinity between memory blocks and processors/cores, the empirical representation of the behavior of the routine is not an easy task, and it may be necessary to have extensive experimentation together with some method to reduce the number of experiments during the installation, but obtaining results that represent the shape of the execution time curve well when the problem and system sizes vary. The information generated in the installation is stored for use when the routine is executed. This information (both in the theoretical and empirical approaches) is included in the routine together with a decision engine to obtain an auto-tuning version of the routine, which is compiled and ready to be called from a non-expert user to solve a problem efficiently.

- **Execution phase**. When a problem is being solved, the size of the problem and the system size established by the user (maximum number of cores) are used to decide the values of the algorithmic parameters which will be used. The SOLAR_Manager is prepared to select the number of threads at each parallelism level and the block sizes. The selection of these parameters is done in the auto-tuning routine prior to the call to the basic routine with the values selected for the parameters. When a theoretical model with experimental estimation of the system parameters is used, the different possible values (from a set of combinations) for the algorithm parameters are substituted in the model, and these values which provide the lowest theoretical execution time are used to solve the problem. In the empirical auto-tuning approach, the information generated in the learning phase is stored in a table with entries for the problem and the system sizes experimented with, and the number of threads and block sizes that are selected are those which give the lowest execution time for a stored size close to the size of the problem we are working with.

Thus, the general characteristics of the methodology are:

- The key factor for the success of the methodology is how the installation of the routine is done. The experiments in the installation should give a precise shape of the evolution of the execution time for different problem sizes and values of the algorithmic parameters. Exhaustive experimentation is necessary, but some pruning method should be used to reduce the search space [19,27,28].
- The results depend on how the experiments carried out in the installation reflect the behaviour of the routine. The SOLAR_Manager is prepared for an automatic installation and execution, but the system manager or the user of the routine could decide to change the parameters of the installation (the Installation_Set) to better reflect the behavior of the routine when a change in the system occurs, for a tuned installation for a particular system or for a better study of the behaviour of the routine for desired problem sizes.
- The method is simple enough to be adapted for routines of different complexity and for complete scientific problems. We analyse its application to linear algebra routines of different computational levels and in different libraries (LAPACK and PLASMA). The experiments show that the method, though simple, produces satisfactory results.

To illustrate how it works, the empirical auto-tuning methodology is applied in the next section to some basic routines (matrix multiplication and matrix factorizations), and extensions to higher-level routines and to routines in PLASMA are considered in subsequent sections.

4 Auto-Tuning of Basic Routines

Two-level routines are used to exploit the different memory levels in large NUMA systems. OpenMP and MKL parallelism are combined, with exploitation of the implicit parallelism provided by the multithread implementations of LAPACK routines in MKL. For example, a scheme of a two-level matrix multiplication may be that shown in Algorithm 1. There is OpenMP parallelism in the `pragma` constructor, with a number of threads `nthomp`, and MKL parallelism when the `dgemm` routine is called, with `nthmkl` threads working in each matrix multiplication. Nested parallelism is enabled and the MKL dynamic selection of threads is disabled because the combination of OpenMP and MKL parallelism does not work with the dynamic selection enabled [10]. The goal of the auto-tuning is to determine, for each problem size, the values of `nthomp` and `nthmkl` with which the lowest execution time is obtained.

Algorithm 1 Scheme of a two-level matrix multiplication.

```
omp_set_nested(1);
mkl_set_dynamic(0);
omp_set_num_threads(nthomp);
mkl_set_num_threads(nthmkl);
# pragma omp parallel {
  obtain size and initial position of the submatrix of A to be multiplied
  call dgemm to multiply this submatrix by matrix B
}
```

In matrix factorization routines the block size is an additional parameter to be decided. For example, in the Cholesky factorization scheme shown in Algorithm 2, the number of OpenMP and MKL threads in the two-level `dgemm` or `dtrsm` routines must be established, as well as the block size `NB`. The optimum values for the different parameters are interdependent, i.e. the value of the block size determines the size of the matrices in the matrix multiplication, and consequently the optimum number of threads in the two-level `dgemm`.

Algorithm 2 Scheme of the LAPACK Cholesky (`potrf`) routine.

```
   DO 20 J = 1, N, NB
* Update and factorize the current
* diagonal block and test
* for non-positive-definiteness.
      JB = MIN(NB, N-J+1)
      CALL dsyrk(...)
      CALL dpotf2(...)
      IF( J+JB .LE. N ) THEN
* Compute the current block column.
         CALL dgemm(...)
         CALL dtrsm(...)
      END IF
  20  CONTINUE
```

The installation of the routine in the system is made by executing the routine for each matrix size specified in the Installation_Set by varying the number of OpenMP and MKL threads at each level of parallelism from one to the number of available cores in the system, and using a combination of threads not exceeding the maximum number of cores (`nthomp×nthmkl≤cores`). Once the routine has been installed, the number of threads with which the lowest time is obtained for each problem size is stored, and, at execution time, for a particular problem size, the number of threads to be used to solve the problem is selected by using the information stored during the installation phase.

Experiments have been carried out in different shared-memory systems, from a medium NUMA system to a large cc-NUMA system:

- **Hipatia**: A cluster with 14 nodes with 2 Intel Xeon Quad-Core, 2.80 GHz, and 2 nodes with 4 Intel Xeon Quad-Core, 2.93 GHz. The nodes with 16 cores are used in the experiments.
- **Saturno**: A NUMA system with 24 cores, Intel Xeon E7530 (hexa-core) processors, 1.87 GHz, 32 GB of shared-memory.
- **Ben**: Composed of 128 cores, Intel-Itanium-2 Dual-Core processors, 1.6 GHz, 1.5 TB of shared-memory.
- **Pirineus**: Comprising 1,344 cores, Intel Xeon X7542 (hexa-core) processors, 2.67 GHz, 6.14 TB of shared-memory. The maximum number of cores to be used together is 256.

4.1 Experiments with the Matrix Multiplication

We begin by analyzing the behavior of the matrix multiplication. Initially only MKL parallelism (1 OpenMP thread—multiple MKL threads) is considered and then the two levels of parallelism (OpenMP+MKL). Table 1 compares, for each system and with

Table 1 Comparison of the execution times of `dgemm`

N	Seq.	Max.Cores	Low.MKL	Low.OMP+MKL	Speed-up MKL/ (OpenMP+MKL)
Saturno					
1,000	0.2498	0.0335	0.0318 (20)	0.0291 (8-3)	1.09
2,000	1.9849	0.2257	0.2257 (24)	0.2151 (6-4)	1.05
3,000	6.6704	0.7255	0.5205 (17)	0.5205 (1-17)	1.00
Ben					
1,000	0.3144	0.0759	0.0269 (22)	0.0181 (5-8)	1.49
2,000	2.4626	0.3306	0.1294 (36)	0.0749 (10-6)	1.73
3,000	8.2604	0.6640	0.3076 (40)	0.2131 (24-4)	1.44
Pirineus					
1,000	0.2199	0.4547	0.0322 (12)	0.0235 (2-16)	1.37
2,000	1.6815	1.1569	0.4796 (16)	0.0797 (5-12)	6.01
3,000	5.4696	1.2903	0.3955 (60)	0.2752 (4-15)	1.44

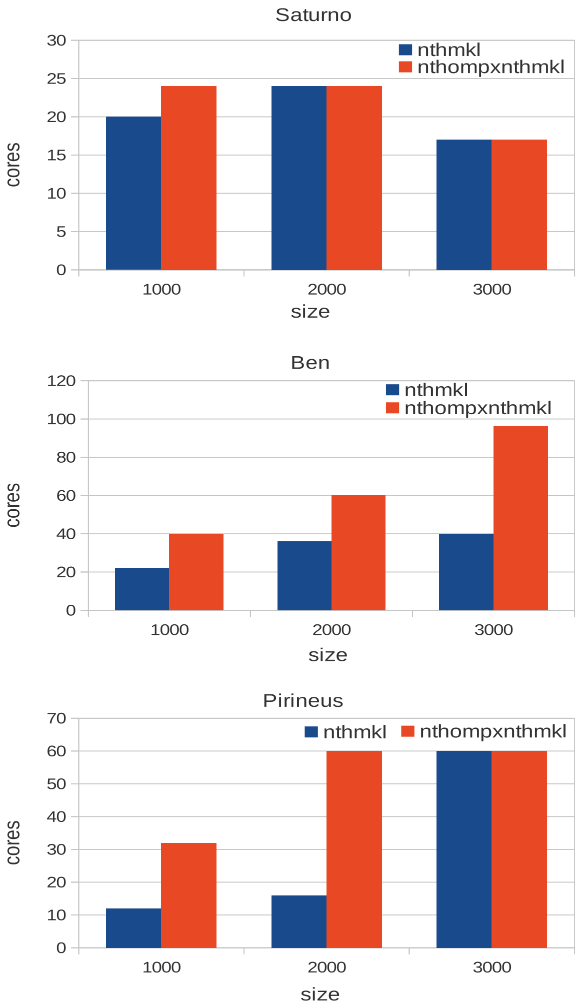
Sequential time (Seq.), execution time with the maximum number of cores (Max.Cores), lowest execution time obtained with MKL when varying the number of threads (Low.MKL) and lowest execution time obtained using two levels of parallelism (Low.OMP+MKL), and speed-up of the OpenMP+MKL implementation with respect to MKL. In brackets, the number of threads with which the lowest time is obtained. Times in seconds hyperthreading capability disabled, the execution time (in seconds) with a sequential (1 OpenMP thread + 1 MKL thread) execution (Seq.), the execution time with MKL parallelism and the maximum number of cores (Max.Cores), the lowest execution time when using only MKL parallelism (Low.MKL) and the lowest execution time obtained using two levels of parallelism (Low.OMP+MKL). The last column shows the speed-up achieved using two levels of parallelism with respect to only MKL parallelism (the quotient Low.MKL/Low.OMP+MKL). The numbers in brackets represent the threads with which the lowest time is obtained. The use of two levels of parallelism provides better speed-up with respect to the use of a single level of parallelism (Speed-up column). Furthermore, the largest reduction is not obtained with a number of threads equal to the maximum number of cores, but with an intermediate combination of OpenMP and MKL threads. The advantage of using two-level parallelism is more apparent in the largest systems, where more cores can be used efficiently with OpenMP+MKL parallelism. This can be observed in Fig. 2, which compares the number of cores with which the lowest execution time is obtained for MKL and OpenMP+MKL.

The results of these initial experiments led us to study auto-tuning techniques capable of determining (during the installation phase) the most appropriate number of OpenMP and MKL threads to establish at each level of parallelism in the execution phase, so ensuring execution times close to the optimum with a minimum installation time. One initial possibility is to fix the number of OpenMP and MKL threads according solely to the number of cores, but not to the characteristics of the system. For example, if the number of OpenMP and MKL threads is established as the square root of the number of cores in the system, satisfactory speed-ups are obtained in the systems where experiments have been carried out, but the number of threads in the two levels does not correspond to the optimal combination of OpenMP and MKL threads. For a more exact combination, exhaustive testing can be conducted in the installation phase, and the large installation time should be reduced with guided and pruned search. Two installation techniques are compared:

- The first technique performs an exhaustive search in all the possible combinations of OpenMP and MKL threads, with the total number of threads up to the number of cores. As a result, it obtains the optimal number of threads corresponding to the minimum execution time, but it uses a high installation time, because all possible executions are carried out for each matrix size in the Installation_Set, varying the number of threads up to the number of available cores.
- In order to reduce the installation time, another technique is applied, in which a guided search based on a percentage is performed. The number of threads at each level of parallelism is increased until we get an execution time that exceeds the current minimum by an amount equal to the percentage established. The precision required to get the optimum number of threads depends on the percentage chosen. Therefore, the use of high percentage values will give a result closer to the optimum, because more executions are carried out by ignoring values corresponding to local optima.

Experiments with exhaustive and guided search have been carried out. The same Installation_ Set {500, 1,000, 3,000} and Validation_Set {700, 2,000} are used during the installation phase and for validation. Table 2 shows the execution time (in seconds)

Fig. 2 Comparison of the number of cores with which the lowest execution time is obtained in MKL and OpenMP+MKL matrix multiplications



obtained when applying guided search with different percentages. The combination of OpenMP-MKL threads with which the optimum time is achieved is shown in brackets. Figure 3 shows the quotient of the execution time for the different percentages with respect to the lowest time. The differences with respect to the lowest experimental time is very small for large problems in large systems, and only in Saturno is a high percentage necessary in order not to fall in local minima, with which the execution time is much higher than the optimum experimental. In the cc-NUMA systems the execution times are very close to the optimum with medium percentages. Furthermore, the time employed in the installation phase is substantially reduced with respect to that with exhaustive search (Fig. 4), giving affordable installation times and satisfactory execution times with the guided search technique with 10 and 20 % thresholds.

Table 2 Execution times (in seconds) of two-level dgemm, obtained after applying exhaustive and guided search using different percentages

N	Exhaustive	1.00 %	10.00 %	20.00 %	50.00 %
Saturno					
500	0.004 (6-4)	0.009 (2-2)	0.009 (2-2)	0.009 (2-2)	0.004 (2-12)
700	0.010 (6-4)	0.023 (2-2)	0.023 (2-2)	0.023 (2-2)	0.011 (2-12)
1,000	0.029 (8-3)	0.033 (3-3)	0.033 (3-3)	0.033 (3-3)	0.030 (2-12)
2,000	0.215 (6-4)	0.260 (3-3)	0.260 (3-3)	0.260 (3-3)	0.258 (2-10)
3,000	0.521 (1-17)	0.834 (3-3)	0.834 (3-3)	0.834 (3-3)	0.709 (3-8)
Ben					
500	0.005 (23-1)	0.013 (1-4)	0.005 (4-6)	0.005 (4-6)	0.005 (1-20)
700	0.010 (7-4)	0.015 (1-10)	0.012 (5-6)	0.012 (5-6)	0.011 (1-19)
1,000	0.018 (10-4)	0.030 (1-16)	0.018 (6-7)	0.018 (6-7)	0.025 (1-19)
2,000	0.080 (10-5)	0.098 (3-15)	0.083 (6-8)	0.083 (6-8)	0.096 (3-16)
3,000	0.219 (25-3)	0.230 (5-14)	0.238 (6-10)	0.238 (6-10)	0.230 (5-14)
Pirineus					
500	0.006 (1-24)	0.008 (4-4)	0.008 (4-4)	0.008 (4-4)	0.008 (4-4)
750	0.013 (2-16)	0.016 (4-4)	0.025 (4-6)	0.025 (4-6)	0.025 (4-6)
1,000	0.026 (2-16)	0.033 (4-3)	0.026 (4-8)	0.026 (4-8)	0.026 (4-8)
2,000	0.080 (5-12)	0.232 (4-8)	0.081 (4-15)	0.081 (4-15)	0.081 (4-15)
3,000	0.275 (4-15)	0.275 (4-15)	0.275 (4-15)	0.275 (4-15)	0.275 (4-15)

In brackets, the combination of OpenMP-MKL threads with which the optimum time is achieved

4.2 Experiments with Matrix Factorizations

In matrix factorizations, the auto-tuning methodology can be applied to its internal dgemm routine, which is used to perform all the matrix multiplications involved in the computation by blocks [12]. As an example, we consider the Cholesky factorization, where the auto-tuning is applied to the dgemm routine used for the matrix multiplications to update the lower triangular parts of matrix A (Algorithm 2). When using the MKL implementation of LAPACK, the block size is automatically determined by the ILAENV function. Therefore, it is necessary to work directly with the reference potrf routine. The dgemm is replaced by a parallel implementation that uses two levels of parallelism, and the auto-tuning process is performed in order to select the most appropriate number of threads at each level. The other routines used internally in the Cholesky factorization are called using their corresponding multithreaded MKL implementations.

Experiments have been carried out in Saturno, and the Installation_Set used is: {256, 768, 1,280, 1,792, 2,304, 2,816, 3,328, 3,840, 4,352}. At running time, the decisions for the problem sizes in the Validation_Set are taken by applying an interpolation process to the information stored during the installation. Table 3 shows the execution times (in seconds) obtained with the auto-tuning methodology and the lowest execution times obtained experimentally by a perfect oracle. The number of

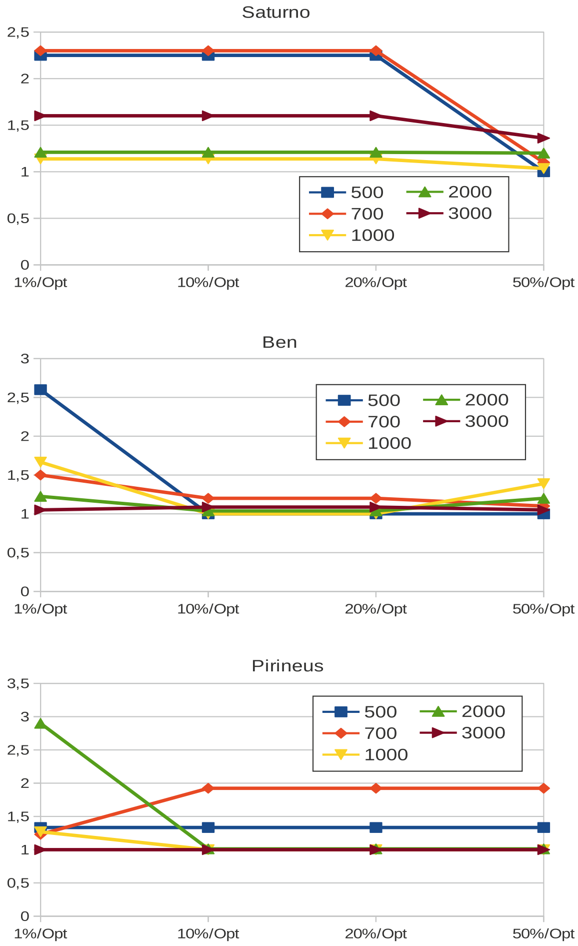


Fig. 3 Quotient of the execution time with the number of OpenMP and MKL threads selected with guided search with different percentages with respect to that with exhaustive search, for the OpenMP+MKL matrix multiplications

OpenMP and MKL threads at each level of parallelism is shown in brackets. The number of threads most frequently used is 24 (the total number of cores in the system), but with different combinations (3-8, 4-6, 6-4). The times obtained with the auto-tuning methodology are normally close to the optimum, and the total number of threads used is also similar. In larger systems, differences in the execution times are higher.

In order to analyse the improvement achieved with this methodology when applied to linear algebra routines which call lower level routines, a comparative study of the execution time obtained by different implementations of the Cholesky `potrf` routine has been carried out. Table 4 shows the results obtained for the reference LAPACK routine (LAPACK), a `potrf` LAPACK routine which internally calls the multithreaded MKL routines `dsyrk`, `dpotf2` and `dtrsm` (LAPACK+MKL), and the modified LAPACK routine where `dgemm` is replaced by the auto-tuned `dgemm2L` routine (LAPACK+AT). The results of applying the auto-tuning methodology are

Fig. 4 Installation time (in seconds) of the MKL and OpenMP+MKL matrix multiplication, obtained in different systems after applying exhaustive search and guided search with different percentages

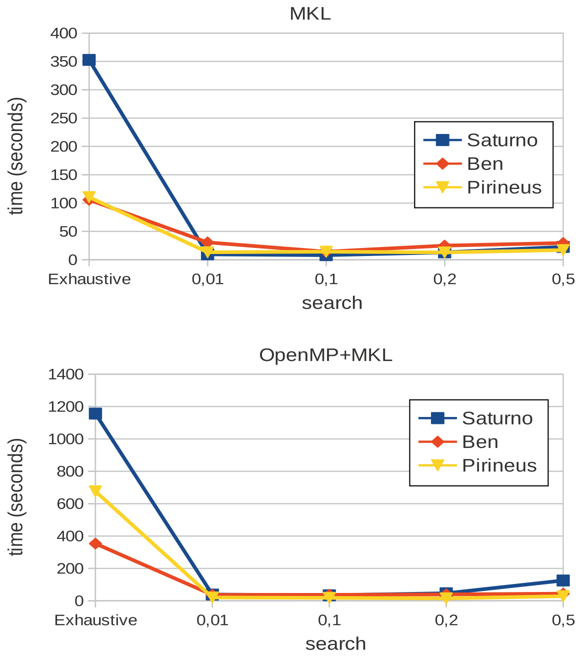


Table 3 Execution times (in seconds) obtained with the application of the auto-tuning methodology (Auto-tuning) to the `dgemm` routine of `potrf` and lowest experimental execution time (Optimum), and number of OpenMP and MKL threads used in these executions, in Saturno

N	Optimum	Auto-tuning
512	0.0012 (1-16)	0.0014 (1-14)
1,024	0.0040 (4-6)	0.0042 (3-8)
1,536	0.0076 (4-6)	0.0080 (6-4)
2,048	0.0134 (2-12)	0.0141 (4-6)
2,560	0.0326 (7-3)	0.0842 (6-4)
3,072	0.0505 (7-3)	0.0835 (6-4)
3,584	0.0780 (3-8)	0.0786 (4-6)
4,096	0.1247 (3-8)	0.1275 (4-6)

satisfactory, but for some problem sizes a loss of performance occurs due to the interpolation applied to select the number of OpenMP and MKL threads.

The Cholesky factorization of LAPACK (Algorithm 2) is computed by blocks. The size and shape of these blocks vary depending on the value selected internally by the LAPACK `ILAENV` function. In this function, that value is selected according to the problem size, but not with respect to the number of threads used. Therefore, we can reduce the execution time even more by selecting the optimum block size for each value of the Installation_Set. To apply this idea to multithreaded routines, two parameters must be selected: the number of threads and the block size. The number of threads is selected for the `dgemm` routine by applying the auto-tuning methodology, and for the selection of the block size it is necessary to work directly with the LAPACK `potrf` routine, so that the block size selected by the `ILAENV` function can be modified

Table 4 Execution times (in seconds) obtained with different versions of the `potrf` routine: the reference LAPACK routine (LAPACK), the LAPACK routine with multithreaded MKL kernels (LAPACK+MKL) and the LAPACK routine with the auto-tuning methodology (LAPACK+AT), in Saturno

N	LAPACK	LAPACK+MKL	LAPACK+AT
512	0.043	0.0039 (9)	0.0038 (1-14)
1,024	0.333	0.0129 (12)	0.0116 (3-8)
1,536	1.105	0.0246 (24)	0.0244 (6-4)
2,048	2.614	0.0755 (24)	0.0766 (4-6)
2,560	5.075	0.1091 (24)	0.1656 (6-4)
3,072	8.787	0.2030 (21)	0.2376 (6-4)
3,584	13.935	0.2792 (21)	0.3230 (4-6)
4,096	20.895	0.3907 (21)	0.3839 (4-6)

In brackets, the number of threads with which the execution times are obtained

Table 5 Execution times (in seconds) obtained for the `potrf` LAPACK routine with a block size selected by the `ILAENV` function (LAPACK+MKL with ILAENV) and the LAPACK routine with auto-selection of the block size and number of OpenMP and MKL threads (LAPACK+AT), in Saturno

N	LAPACK+MKL ILAENV		LAPACK+AT threads+block	
	Block	Time	Block	Time
512	32	0.0039 (9)	128	0.0034 (1-14)
1,024	96	0.0129 (12)	128	0.0115 (3-8)
1,536	192	0.0246 (24)	64	0.0265 (6-4)
2,048	384	0.0755 (24)	128	0.0621 (4-6)
2,560	384	0.1091 (24)	64	0.0878 (6-4)
3,072	512	0.2030 (21)	64	0.1457 (6-4)
3,584	512	0.2792 (21)	256	0.2524 (4-6)
4,096	512	0.3907 (21)	256	0.3645 (4-6)

In brackets, the number of OpenMP and MKL threads with which the lowest times are obtained

in order to select the best block size. Table 5 compares, for different matrix sizes, the execution time obtained for the `potrf` routine when the block size is internally selected by `ILAENV` and the execution time when the block size is selected with the auto-tuning technique. All the experiments have been done in Saturno using the same Installation_Set $\{256, 768, 1{,}280, 1{,}792, 2{,}304, 2{,}816, 3{,}328, 3{,}840, 4{,}352\}$, with block sizes power of 2 from 32 to 512 and a number of cores from 1 to 24. When the routine `potrf` uses the `ILAENV` function to select the block size, the same value is used for several matrix sizes regardless of the number of threads. When the number of threads and the block size are selected with the auto-tuning methodology, lower execution times are obtained. For small matrix sizes, the use of larger blocks is preferable, but for larger sizes a lower value than that selected by the `ILAENV` function (which does not consider the number of threads, only the matrix size) is more appropriate. Figure 5 compares the speed-up with respect to the LAPACK+MKL version with the block size selected with the `ILAENV` function of the two versions of auto-tuning: when only the

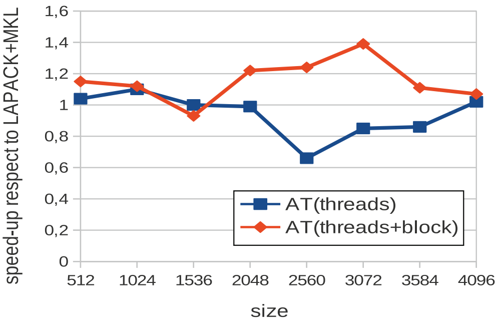


Fig. 5 Quotient of the execution time of auto-tuning with threads selection and with threads and block size selection, with respect to LAPACK+MKL with block size selected with ILANEV, for the potrf routine in Saturno

number of threads is selected and when the block size is also selected as a function of the number of threads. The improvement achieved by selecting the appropriate block size is between 6 and 30 % for most matrix sizes. Therefore, if we also use the block size joint with the selection of the number of threads in the auto-tuning methodology, better results are obtained for the potrf routine. Similar results are obtained for other routines, and the advantage of the auto-tuning is more apparent in larger systems.

5 Extension of the Methodology to Higher Level Routines

The same methodology applied to low level routines can be extended to other routines which call low level routines. With the Cholesky factorization we have shown how the methodology can be directly applied to the routine or to a lower level routine (dgemm) used internally. For higher level routines we have the same possibilities. For example, the dgesv routine to solve general multiple linear systems uses the LU factorization, which in turn uses the matrix multiplication. The auto-tuning technique can be applied to the matrix multiplication and then the LU factorization uses the two level matrix multiplication with a different number of threads at each level and for the different multiplications carried out in the factorization process. Otherwise, the auto-tuning can be made at a higher level in the LU factorization by selecting the optimum number of threads and the block size. Once we have an auto-tuning LU factorization it can be used in the linear system routine.

LAPACK routine sysv can be used to solve a linear system $AX = B$ with A a symmetric matrix. But the implementation of this routine in LAPACK (or in MKL) is not very scalable, and when the number of cores is large it may be preferable to use the general routine gesv. The auto-tuning methodology can be used in this case to decide both the routine to use and the values of the parameters. The joint installation of these routines should provide information of the preferred routine and of the number of threads (and consequently of cores) to use when solving a particular problem, depending on the problem size and the size of the computational system (the number of cores reserved to solve the problem).

With the Installation_Set {256, 768, 1,280, 1,792, 2,304, 2,816, 3,328, 3,840} in Saturno the installation time with the exhaustive search is 79 s for dsysv, 62 for dgesv, 217 for zsysv and 231 for zgesv, giving a total of 589 s. This is not long, but it can be reduced with a guided search as explained in Sect. 4 for the matrix multiplication. For example, a guided search for `dsysv` with a stopping threshold of 1 % gives an installation time of 9 s, and when the threshold is 10 % the installation time rises to 36 s. This reduction in the installation time could produce a worse selection of the number of threads at execution time, but we can see in Fig. 6 (top) that the selection is satisfactory with the 10 % threshold, for which the execution time and the selected number of threads practically coincide with those obtained with exhaustive search. If we consider that the deviation in the execution time is obtained as the mean of the relative deviation for all the sizes $\left(\frac{|obtained\ time - optimum\ time|}{optimum\ time}\right)$, exhaustive search gives a deviation of approximately 4.8 %, and so this deviation is the minimum we can expect from any installation method. Lower deviations are obtained with guided search (4.3 and 2.5 % with guided search with 1 and 10 % threshold), which means guided search produces satisfactory results with a lower installation time. These results are due to the shape of the execution time of the routine in Saturno, where the execution time is practically constant from a number of threads on. For larger systems the differences are bigger. The figure also compares (bottom) the selection of the number of threads. The selection with exhaustive search and guided search with 10 % threshold practically coincide, and the number of threads is very close to the optimum. The mean deviation of the selected number of threads with respect to the optimum is of 2 threads for

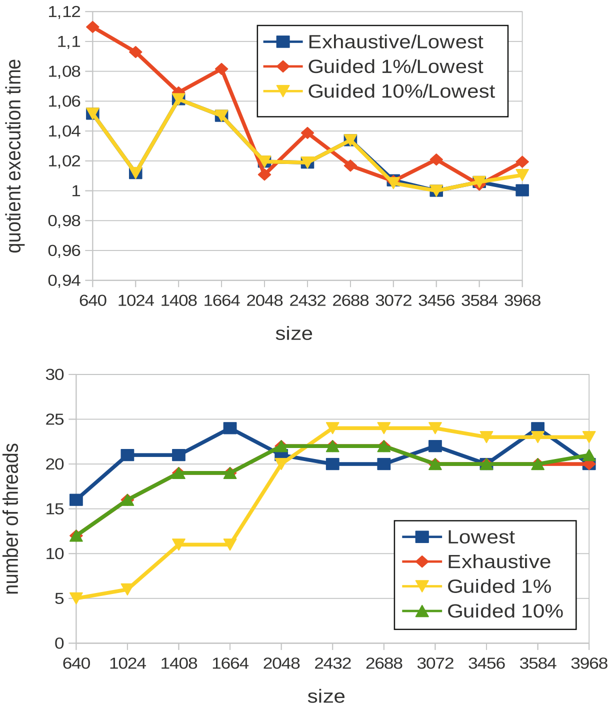


Fig. 6 Comparison of the optimum execution time (*top*) and number of threads (*bottom*) with those obtained with exhaustive search and guided search with 1 and 10 % threshold. Routine `dsysv`, in Saturno

exhaustive search and guided search with 10 % threshold, but with a 1 % threshold the deviation rises to approximately 5, which means a worse selection which has no proportional influence on the execution time, due to its flat shape.

The guided installation can be extended to be applied to the joint installation of several similar routines. For example, when installing `dsysv` and `dgesv` the information generated for one of them can be used to guide the search for the other routine. If the first routine being installed is `dsysv` and the number of threads selected for each problem size in the installation is used to start the search for this problem size for the routine `dgesv`, the installation time for `dgesv` with a guided search with 1 and 10 % threshold is 8 and 9 s. So, the installation time of the two routines is reduced from 141 to 17 and 45 s.

The routine to use in the solution of the problem can be selected using the information stored in the installation. To do so, it is necessary to store for each problem size in the Installation_Set the number of threads with which the execution time of the routine `gesv` is for the first time lower than that of the corresponding routine `sysv`. For a given problem size in a Validation_Set we consider that the change from `sysv` to `gesv` occurs for the same number of threads as for the closest problem size in the Installation_Set. The number of threads at which the change happens is shown in Table 6. The table shows results in Saturno and in a quadcore laptop. For the problem sizes in which the change does not happen at the predicted point, two values are shown: the number of threads at which `gesv` is better than `sysv` for the first time, and the number of threads at which the change happens according to the information stored in the installation. In Saturno, the number of threads at which the change happens is well predicted in 35 of 46 cases, and in the laptop in 30 of 46, which means the correct selection is made approximately 70 % of the times. The behavior in the two systems is slightly different, with the number of threads at which the change happens increasing with the problem size more quickly in Saturno than in the laptop, which may be due to the difference in the memory hierarchy in the two systems and the use of different versions of MKL. Furthermore, an unexpected behaviour is observed in the laptop, where the `gesv` routines with one thread are preferred to `sysv` for small problem sizes, which is detected at installation time.

If the routines `dsysv` and `dgesv` are installed together in Saturno with determination of the best routine for each number of cores, the installation time is reduced from 140 to 48 s, and in the joint installation of `zsysv` and `zgesv` the reduction is from 447 to 135 s.

As we did with the Cholesky factorization in the previous section, the multithread MKL or reference LAPACK routines can be used to introduce auto-tuning with selection of the number of threads and the block size. To compare the behavior of MKL and reference LAPACK routines, the routines `zsysv` and `zgesv` are compared in Saturno in Table 7 and Fig. 7. The table shows the execution time of the two routines in their implementation in MKL and reference LAPACK, with one thread and 24 threads, and the figure shows the quotient of the execution time obtained with reference LAPACK with respect to that with MKL. The block size used by MKL is not known, and LAPACK always selects a block size of 64, regardless of the number of threads. We can see that MKL slightly improves on LAPACK in the sequential version (approximately 4 %), which can be produced by a suitable selection of the

Table 6 Comparison of the number of threads at which `gesv` outperforms `sysv` and the number of threads at which this happens according to the information stored at installation time, with different problem sizes and in two different systems

Size	Saturno		Laptop	
	`dsysv-dgesv`	`zsysv-zgesv`	`dsysv-dgesv`	`zsysv-zgesv`
384	2	3/2	1	2/1
512	2	3/2	2/1	2/1
640	2	3	2/1	2
896	3/2	3	2/1	3/2
1,024	2	3	1	2
1,152	3	3	2	3
1,408	3	3	2	3
1,536	3	3	2	3
1,664	3	3	2	3
1,920	3	5/3	2	3
2,048	3	5/3	2	3
2,176	4	5	2	3
2,432	4	5	2	3
2,560	4	5	2	3
2,688	4	5	2	3
2,944	5/4	7	2	3
3,072	3/4	7	3/2	3
3,200	5/4	7	2/3	3
3,456	5/4	7	3	3
3,584	5/4	9/7	2/3	4/3
3,712	5	9	3/4	3/–
3,968	5	9	3/4	3/–
4,096	5	9	3/4	3/–
Deviation	0.26	0.35	0.30	0.43

block size in MKL, but also by additional internal optimizations. The improvement with 24 threads is higher, around 20 % for `zsysv` and 60 % for `zgesv`. No dynamic selection of the number of threads was allowed in MKL, so the improvement is not due to the number of threads. It is possible that MKL changes the block size with the number of threads, but additional optimizations must be included, especially in `zgesv`. In reference LAPACK the parallelism is achieved in the basic routines which `zsysv` and `zgesv` call (MKL routines), and so the parallelism could be at a lower level than that in MKL.

Our goal is not to reduce the execution time obtained with the MKL routines, but to analyze the parameters selection methodology. We consider the joint selection of the number of threads and the block size. Table 8 compares the execution time in Saturno of the routines `zsysv` and `zgesv` in the cases:

Table 7 Comparison of the execution time of MKL and reference LAPACK routines `zsysv` and `zgesv` for different problem sizes, when using 1 and 24 threads, in Saturno

Size	`zsysv`		`zgesv`	
	MKL	LAPACK	MKL	LAPACK
	1 thread			
512	0.036	0.040	0.060	0.060
1,024	0.257	0.274	0.458	0.455
1,536	0.831	0.879	1.499	1.516
2,048	1.926	2.013	3.487	3.560
2,560	3.697	3.865	6.707	6.981
3,072	6.304	6.554	11.49	11.99
Total	13.05	13.62	23.70	24.55
	24 threads			
512	0.024	0.032	0.009	0.020
1,024	0.112	0.139	0.038	0.090
1,536	0.294	0.332	0.113	0.229
2,048	0.525	0.624	0.250	0.472
2,560	0.873	1.042	0.540	0.842
3,072	1.269	1.525	0.903	1.296
Total	3.09	3.69	1.85	2.94

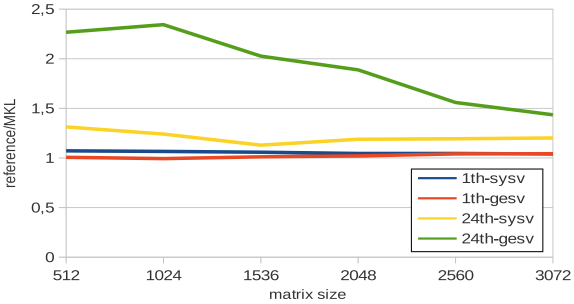


Fig. 7 Quotient of the execution time of routines `zsysv` and `zgesv` in reference LAPACK with respect to MKL for different problem sizes, when using 1 and 24 threads, in Saturno

- With a block size of 64 and a number of threads equal to that of available cores (column 64-24). This seems to be a good choice because when more cores are used a larger reduction of the execution time is expected, and 64 is a reasonable block size, and is selected by the LAPACK routine `ILAENV`.
- With a block size of 64 and the number of threads with which the lowest execution time is obtained (column 64-thr).
- With the combination of block size and number of threads which gives the lowest experimental execution time (Lowest).
- The rows for sizes 1,024, 2,048 and 3,072 show, in the column AT-blo-thr, the execution time with the block size and number of threads selected. The rows for sizes 512, 1,536 and 2,560 correspond to the values obtained with executions for all the number of cores (from 1 to 24) and block sizes a multiple of 8 from 8 to 256. These three matrix sizes could be used for installation, and the decisions for other

Table 8 Comparison of the execution time of the routines zsysv and zgesv in Saturno, for block size and number of threads fixed at 64 and 24 (64-24), for the best number of threads with block size 64 (64-thr), for the best combination of block size and number of threads (optimum), and with the block size and number of threads empirically selected from information obtained at installation time (emp-blo-thr)

Size	64-24	64-thr	Lowest	AT-blo-thr
zsysv				
512	0.032	0.029 (17)	0.025 (18-40)	
1,024	0.139	0.131 (21)	0.116 (18-40)	0.116 (18-32)
1,536	0.332	0.322 (21)	0.292 (18-32)	
2,048	0.624	0.611 (21)	0.585 (21-32)	0.596 (20-32)
2,560	1.042	1.032 (17)	0.961 (22-40)	
3,072	1.525	1.525 (24)	1.446 (24-48)	1.517 (22-40)
zgesv				
512	0.020	0.020 (21)	0.018 (18-32)	
1,024	0.090	0.090 (24)	0.083 (24-32)	0.085 (21-32)
1,536	0.228	0.228 (24)	0.207 (22-24)	
2,048	0.472	0.472 (24)	0.449 (24-48)	0.460 (23-32)
2,560	0.819	0.819 (24)	0.804 (24-56)	
3,072	1.296	1.296 (24)	1.294 (22-56)	1.307 (23-56)

The number of threads and the optimum block size used in the different cases are also shown

problem sizes (1,024, 2,048 and 3,072 in the table) are taken using the information stored in the installation.

Some conclusions can be drawn:

- The selection of the number of threads slightly reduces the execution time for small matrices or in zsysv, with a worse parallelism, which means that the best number of threads does not coincide normally with the maximum. In larger systems the differences would be more apparent.
- The selection of the block size and the number of threads allows additional reduction of the execution time, which is greater for small matrices, and with a maximum percentage of reduction in the experiments of about 18 %.
- The use of auto-tuning normally produces a reduction in the execution time with respect to that with block size 64 and 24 cores, but this reduction is lower than the optimum obtained experimentally, which is normal because auto-tuning does not always select the best parameters combination.

6 Application of the Methodology to PLASMA Routines

The same methodology shown with the multithread MKL implementation of LAPACK can be applied to other linear algebra packages. The PLASMA library [24] is conceived to exploit the multicore capacity of present systems, and we show here some experiments which prove that empirical auto-tuning techniques are also applicable to PLASMA.

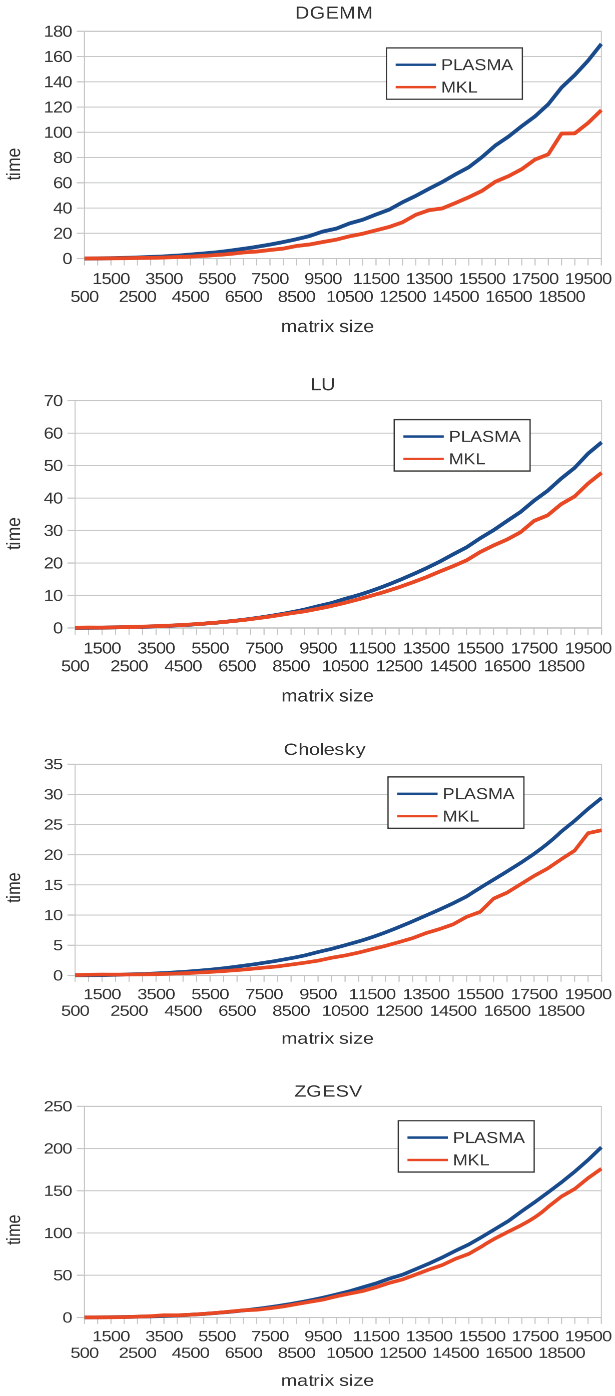


Fig. 8 Comparison of the execution times of the routines `dgemm`, `dpotrf`, `dgetrf` and `zgesv`, in Hipatia with 16 threads

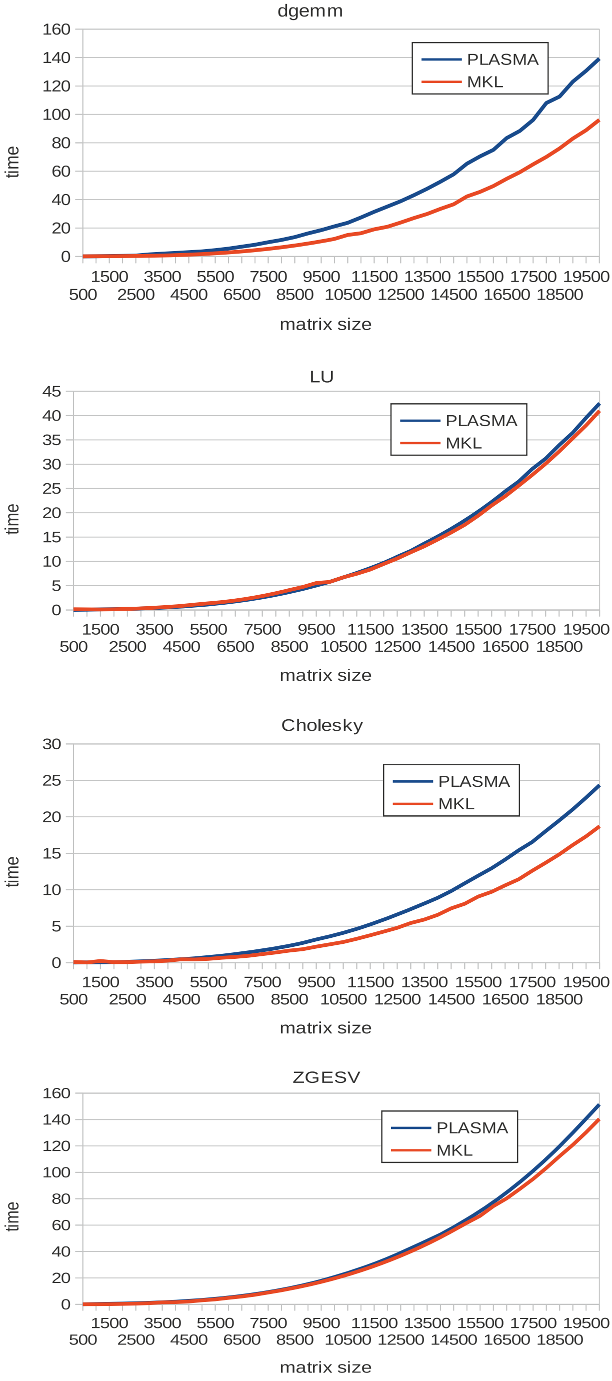


Fig. 9 Comparison of the execution times of the routines `dgemm`, `dpotrf`, `dgetrf` and `zgesv`, in Saturno with 24 threads

The behavior of some routines in LAPACK and PLASMA in medium multicore systems is shown in Figs. 8 (Hipatia) and 9 (Saturno). The results with Hipatia have been obtained in a node with 16 cores using 16 threads, and in Saturno 24 threads have been used. For all the routines the default values were taken for the block sizes: block and inner block equal to 120 for the matrix multiplication and the Cholesky factorization and block of size 200 and inner block of size 20 for the LU factorization and the complex linear system routine (which uses the LU factorization). The comparison of the two libraries is similar in both systems:

- Better results are obtained with MKL for large matrices. PLASMA outperforms MKL only for some small problem sizes.
- The percentage of the improvement of MKL with respect to PLASMA is different in the two systems and for the different routines, which makes it impossible to draw general conclusions about the advantage of using MKL.
- The differences in the execution time between the two libraries is not very large, and the selection of the optimum block sizes in PLASMA could make it competitive with MKL. One good option for optimization is to consider the selection of the library together with the number of threads and the block sizes.

So, the same techniques used with LAPACK and MKL routines to select the best implementation and configuration of parameters for a particular problem size and in a particular system can be applied to PLASMA.

To achieve high performance with multicore architectures, PLASMA relies on tile algorithms, which provide fine granularity parallelism. Its performance strongly depends on tunable execution parameters trading off utilization of different system resources. The outer block size (NB) trades off parallelization granularity and scheduling flexibility with single core utilization, while the inner block size (IB) trades off memory load with extra-flops. In this case the values to search for in the installation are the number of threads, NB and IB. The search begins with the smallest problem size in the Installation_Set, with which the experimentation time is small. From these initial optimum tuning parameters the search for larger problems can continue in different ways. Two possibilities are considered:

- A guided search exploring all the possible combinations (`PLASMA-GS-All`) of number of threads, outer block size, and inner block size (th, NB, IB) begins by using the best combination of parameters found for the previous matrix size in the Installation_Set. The search continues using neighbouring values for the parameters obtained by applying a pre-set of increments for each parameter ($th \pm th\_inc$, $NB \pm NB\_inc$, $IB \pm IB\_inc$), until we get an execution time that exceeds the minimum in an amount equal to the threshold established. As a result, we obtain the optimal number of threads, NB and IB for each problem size in the Installation_Set, but with a high installation time, because 27 executions are carried out for each matrix size in the Installation_Set, varying th, NB and IB.
- The second technique (`PLASMA-GS-Par`) performs a guided search in each parameter, varying the value of one parameter and maintaining the value of the other parameters fixed. As in the previous case, the search starts with the value for the parameters that provide the best execution time for the previous problem size in

Table 9 Comparison of the time obtained for the PLASMA LU routine with the thread, NB and IB configuration (in brackets) selected with the installation techniques `PLASMA-GS-All` and `PLASMA-GS-Par`, the lowest time obtained with MKL LU and the lowest time obtained with PLASMA LU

Size	`PLASMA-GS-All`	`PLASMA-GS-Par`	MKL	PLASMA
2,000	0.11 (17-103-10)	**0.10** (15-128-98)	0.11	0.15
3,000	1.17 (23-78-34)	**0.21** (23-158-106)	0.28	0.29
4,000	**0.39** (24-158-47)	0.40 (24-173-119)	0.55	0.50
5,000	**0.69** (24-188-52)	0.70 (24-183-148)	1.06	0.87
6,000	1.17 (23-228-65)	1.17 (23-198-184)	**0.87**	**0.87**
7,000	1.76 (23-248-59)	**1.74** (24-218-208)	2.36	2.13
8,000	**2.48** (24-248-78)	2.52 (24-258-238)	3.45	3.08
9,000	**3.50** (24-268-87)	**3.50** (24-288-270)	4.73	4.31
10,000	4.70 (24-320-110)	**4.68** (24-288-276)	5.78	5.79
Total	15.97	**15.02**	19.19	17.99

In Saturno, times in seconds

Bold values indicate the lowest time obtained for each problem size

the Installation_Set. The search for the best possible value for each parameter ends when the execution time exceeds the threshold.

Some PLASMA routines depend on one parameter and on the number of threads (for example, the Cholesky factorization) while others depend on the three parameters. As an example we analyse how the installation process works for the LU factorization, which depends on the three parameters. The Installation_Set used in the experiments is {500, 1500, 2500, 3500, 4500, 5500, 6500, 7500, 8500, 9500}; the increment values are $th\_inc = 1$, $NB\_inc = 10$ and $IB\_inc = 2$; and the threshold value is set to 10 %.

Tables 9 (Saturno) and 10 (Hipatia) show, for different matrix sizes, the execution time (in seconds) obtained for the PLASMA LU factorization when the parameters are selected with the two techniques considered, the lowest execution time obtained for the MKL LU factorization, and the lowest time obtained with PLASMA LU factorization without parameter tuning. The improvements obtained with the two techniques are close, with a difference in the total time of approximately 6 % in favour of `PLASMA-GS-Par` in Saturno and of approximately 10 % in favour of `PLASMA-GS-All` in Hipatia. The time required to tune the parameters with `PLASMA-GS-All` is much higher than that with `PLASMA-GS-Par`. For example, in Saturno, this time with `PLASMA-GS-All` is about 50 min, while with `PLASMA-GS-Par` the installation time is reduced to approximately 10 min. In both systems, the reduction of the execution time obtained with the parameters tuning methodology is important, of about 26 % with respect to MKL and 20 % with respect to PLASMA without tuning in Saturno, and of about 15 % with respect to MKL and 23 % with respect to PLASMA without tuning in Hipatia.

The same techniques used with the LU factorization can be applied to other routines, but different results are obtained depending on the particular routine and system. For example, when the auto-tuning methodology is applied to the Cholesky factorization in Saturno with the Installation_Set {500, 1000, 1500, 2000, 2500, 3000, 3500,

Table 10 Comparison of the time obtained for the PLASMA LU routine with the thread, NB and IB configuration (in brackets) selected with the installation techniques `PLASMA-GS-All` and `PLASMA-GS-Par`, the lowest time obtained with MKL LU and the lowest time obtained with PLASMA LU

Size	`PLASMA-GS-All`	`PLASMA-GS-Par`	MKL	PLASMA
2,000	**0.14** (15-154-30)	0.18 (16-157-41)	0.15	0.15
3,000	0.37 (16-204-55)	0.37 (16-204-167)	0.37	**0.28**
4,000	**0.59** (16-234-75)	0.66 (16-234-221)	0.70	0.66
5,000	**0.95** (16-284-100)	1.17 (16-274-239)	1.20	1.12
6,000	**1.58** (16-334-125)	1.72 (16-324-290)	1.85	1.82
7,000	**2.26** (16-364-140)	2.54 (16-364-343)	2.66	2.78
8,000	**3.26** (16-404-165)	3.61 (16-404-385)	3.86	4.09
9,000	**4.48** (16-424-185)	4.85 (16-424-416)	5.11	5.66
10,000	**5.86** (16-424-190)	6.40 (16-424-420)	6.71	7.67
Total	**19.49**	21.50	22.61	24.23

In Hipatia, time in seconds
Bold values indicate the lowest time obtained for each problem size

4000, 4500, 5000}, the installation time is reduced from approximately 57 min with exhaustive search to approximately 7 min with guided search with 10 % threshold. The auto-tuning selects satisfactory values of the parameters, but in this case only two parameters are tuned (number of threads and block size) and the advantage of MKL with respect to PLASMA is bigger than for the LU factorization (Fig. 9), and the auto-tuned PLASMA version is surpassed by the implementation in MKL.

7 Conclusions

An empirical auto-tuning methodology for linear algebra routines in NUMA systems has been analyzed. The methodology has been tested in different multicore systems and with routines of different computational levels and from different linear algebra packages. Some aspects of the methodology have been explained and exhaustive experimentation shows their applicability:

- The values of some parameters are selected: the number of OpenMP and MKL threads in two-level routines, the block size in algorithms by blocks, or the outer and inner block sizes in implementations with two block levels (for example, the PLASMA LU factorization).
- When working at a medium or high computational level (routines which call lower level routines), the methodology can be applied directly to the medium or high level routine or it can use an auto-tuned version of the lower level routines.
- Better results are obtained when the parameters are auto-tuned jointly rather than separately. This happens if we consider the block size as dependent on the number of threads and the problem size, and not only on the latter.
- Satisfactory parameters selection is obtained with low installation time with guided search techniques, which can be implemented to avoid falling into local minima.

- The joint installation of a group of similar routines can be conducted in such a way that an important reduction of the installation time is obtained.
- In some cases in which several routines, implementations or libraries are available to solve a particular problem, the joint installation of them for auto-tuning allows the best routine together with the values of the parameters to be selected.
- The results obtained are system-dependent, and no general conclusions about the best values for the parameters can be drawn, but the methodology has been shown to provide satisfactory results in systems of different characteristics, and the advantage of using the methodology is more apparent in larger systems, where the range of possible values for the parameters increases.

In conclusion, the methodology has been shown to be versatile and useful for the design of efficient routines, which can execute efficiently (obtaining execution times close to the lowest achievable) without the need for user expertise.

We are working on the application of similar methodologies to other types of parallelism (message-passing and GPUs) to routines of different types or to particular scientific applications.

Acknowledgments Partially supported by Fundación Séneca, Consejería de Educación de la Región de Murcia, 08763/PI/08, PROMETEO/2009/013 from Generalitat Valenciana, the Spanish Ministry of Education and Science through TIN2012-38341-C04-03, and the High-Performance Computing Network on Parallel Heterogeneus Architectures (CAPAP-H). The authors gratefully acknowledge the computer resources and assistance provided by the Supercomputing Centre of the Scientific Park Foundation of Murcia and by the Centre de Supercomputació de Catalunya.

References

1. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. J. Phys. Conf. Ser. **180**(1), 1–5 (2009)
2. Alberti, P., Alonso, P., Vidal, A.M., Cuenca, J., Giménez, D.: Designing polylibraries to speed up linear algebra computations. Int. J. High Perform. Comput. Netw. **1/2/3**(1), 75–84 (2004)
3. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J.J., Du Croz, J., Grenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., D. Sorensen, S.: LAPACK User's Guide. Society for Industrial and Applied Mathematics, Philadelphia (1995)
4. Bernabé, G., Cuenca, J., Giménez, D.: Optimization techniques for 3D-FWT on systems with manycore GPUs and multicore CPUs. In: ICCS (2013)
5. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. **35**(1), 38–53 (2009)
6. Cámara, J., Cuenca, J., Giménez, D., Vidal. A.M.: Empirical autotuning of two-level parallel linear algebra routines on large cc-NUMA systems. In: ISPA (2012)
7. Caron, E., Desprez, F., Suter, F.: Parallel extension of a dynamic performance forecasting tool. Scalable Comput. Pract. Exp. **6**(1), 57–69 (2005)
8. Chen, Z., Dongarra, J., Luszczek, P., Roche, K.: Self adapting software for numerical linear algebra and LAPACK for clusters. Parallel Comput. **29**, 1723–1743 (2003)
9. Cuenca, J., Giménez, D., González, J.: Achitecture of an automatic tuned linear algebra library. Parallel Comput. **30**(2), 187–220 (2004)
10. Cuenca, J., García, L.P., Giménez, D.: Improving linear algebra computation on NUMA platforms through auto-tuned nested parallelism. In: Proceedings of the 2012 EUROMICRO Conference on Parallel, Distributed and Network Processing (2012)
11. Frigo, M.: FFTW: An adaptive software architecture for the FFT. In: Proceedings of the ICASSP Conference, vol. 3, p. 1381 (1998)

12. Golub, G., Van Loan, C.F.: Matrix Computations, 3rd edn. The John Hopkins University Press, Baltimore (1996)
13. Im, E.-J., Yelick, K., Vuduc, R.: Sparsity: optimization framework for sparse matrix kernels. Int. J. High Perform. Comput. Appl. (IJHPCA) **18**(1), 135–158 (2004)
14. Intel MKL web page.: http://software.intel.com/en-us/intel-mkl/
15. Jerez, S., Montávez, J.-P., Giménez, D.: Optimizing the execution of a parallel meteorology simulation code. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium. IEEE (2009)
16. Katagiri, T., Kise, K., Honda, H., Yuba, T.: Fiber: a generalized framework for auto-tuning software. Springer LNCS **2858**, 146–159 (2003)
17. Katagiri, T., Kise, K., Honda, H., Yuba, T.: ABCLib-DRSSED: a parallel eigensolver with an auto-tuning facility. Parallel Comput. **32**(3), 231–250 (2006)
18. Kurzak, J., Tomov, S., Dongarra, J.: Autotuning gemm kernels for the FERMI GPU. IEEE Trans. Parallel Distrib. Syst. **23**(11), 2045–2057 (2012)
19. Lastovetsky, A.L., Reddy, R., Higgins, R.: Building the functional performance model of a processor. In: SAC, pp. 746–753 (2006)
20. Li, J., Skjellum, A., Falgout, R.D.: A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. Concurrency Pract. Exp. **9**(5), 345–389 (1997)
21. Naono, K., Teranishi, K., Cavazos, J., Suda, R., (eds.): Software Automatic Tuning. From Concepts to State-of-the-Art Results. Springer, Berlin (2010)
22. Nath, R., Tomov, S., Dongarra, J.: An improved MAGMA gemm for FERMI graphics processing units. IJHPCA **24**(4), 511–515 (2010)
23. Petitet, A., Blackford, L.S., Dongarra, J., Ellis, B., Fagg, G.E., Roche, K., Vadhiyar, S.S.: Numerical libraries and the grid. IJHPCA **15**(4), 359–374 (2001)
24. PLASMA.: http://icl.cs.utk.edu/plasma/
25. Püschel, M., Moura, J.M.F., Singer, B., Xiong, J., Johnson, J.R., Padua, D.A., Veloso, M.M., Johnson, R.W.: Spiral: a generator for platform-adapted libraries of signal processing algorithms. IJHPCA **18**(1), 21–45 (2004)
26. Seshagiri, L., Wu, M.-S., Sosonkina, M., Zhang, Z., Gordon, M.S., Schmidt, M.W.: Enhancing adaptive middleware for quantum chemistry applications with a database framework. In: IPDPS Workshops, pp. 1–8 (2010)
27. Tanaka, T., Katagiri, T., Yuba, T.: d-Spline based incremental parameter estimation in automatic performance tuning. In: PARA, pp. 986–995 (2006)
28. Vuduc, R., Demmel, J., Bilmes, J.: Statistical models for automatic performance tuning. In: International Conference on Computational Science (1), pp. 117–126 (2001)
29. Whaley, R.C., Petitet, A., Dongarra, J.: Automated empirical optimizations of software and the ATLAS project. Parallel Comput. **27**(1–2), 3–35 (2001)