Parallel Computing 40 (2014) 309–327

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Auto-tuned nested parallelism: A way to reduce the execution time of scientific software in NUMA systems

Jesús Cámara [a], Javier Cuenca [b,*], Luis-Pedro García [c], Domingo Giménez [a]

[a] Departamento de Informática y Sistemas, Facultad de Informática, Universidad de Murcia, 30100 Murcia, Spain
[b] Departamento de Ingeniería y Tecnología de Computadores, Facultad de Informática, Universidad de Murcia, 30100 Murcia, Spain
[c] Servicio de Apoyo a la Investigación Tecnológica, Universidad Politécnica de Cartagena, 30203 Cartagena, Spain

ARTICLE INFO

Article history:
Available online 2 April 2014

Keywords:
Auto-tuning
Linear algebra
Performance modeling
NUMA

ABSTRACT

The most computationally demanding scientific problems are solved with large parallel systems. In some cases these systems are Non-Uniform Memory Access (NUMA) multiprocessors made up of a large number of cores which share a hierarchically organized memory. The main basic component of these scientific codes is often matrix multiplication, and the efficient development of other linear algebra packages is directly based on the matrix multiplication routine implemented in the BLAS library. BLAS library is used in the form of packages implemented by the vendors or free implementations. The latest versions of this library are multithreaded and can be used efficiently in multicore systems, but when they are used inside parallel codes, the two parallelism levels can interfere and produce a degradation of the performance. In this work, an auto-tuning method is proposed to select automatically the optimum number of threads to use at each parallel level when multithreaded linear algebra routines are called from OpenMP parallel codes. The method is based on a simple but effective theoretical model of the execution time of the two-level routines. The methodology is applied to a two-level matrix–matrix multiplication and to different matrix factorizations (LU, QR and Cholesky) by blocks. Traditional schemes which directly use the multithreaded routine of BLAS, `dgemm`, are compared with schemes combining the multithreaded `dgemm` with OpenMP.

1. Introduction

Nowadays, it is possible to find a parallel system in a personal computer, in a laptop, or even in a smartphone. Furthermore, the most important scientific computing platforms are built by connecting multicore nodes. On the other hand, the complexity and the quantity of calculations required for the different scientific and engineering branches grows simultaneously with the improvement in computer technology. So, the design of scientific applications for these different platforms must consider their parallel nature in order to obtain maximum performance [1].

The work necessary for optimizing a real code could take several weeks or months and it requires deep knowledge in several disciplines, like computer architecture, programming and debugging tools, numerical analysis and mathematical software. Furthermore, the optimization task performed for a specific platform need not be suitable, in principle, for other platforms. Such a diverse working environment has led to important changes in the traditional way of optimizing the

* Corresponding author. Tel.: +34 868 88 4821; fax: +34 868 88 4151.
E-mail addresses: jcm23547@um.es (J. Cámara), jcuenca@um.es (J. Cuenca), luis.garcia@sait.upct.es (L.-P. García), domingo@um.es (D. Giménez).

http://dx.doi.org/10.1016/j.parco.2014.03.011
0167-8191/

software for scientific calculations, with the goal of following the pace of both the user needs and the new hardware developments. In this context, different automatic optimization techniques have emerged as valuable tools that provide scientific software with environment adaptation capacity. In this way, optimized code for each specific platform can be generated automatically by means of software models or certain experimenting. Some projects where these techniques have been applied are: ATLAS [2], with optimized computational kernels for dense linear algebra; SPARSITY [3], with kernels for sparse linear algebra; FFTW [4], with an optimized implementation of the Fourier discrete transform; ABCLib-DRSSED [5], with routines for obtaining eigenvalues.

In most scientific and engineering problems, computations are carried out by using basic BLAS-type matrix routines [6]. Therefore, the improvement in the performance of scientific codes is achieved in many cases by the efficient use of these routines. The basic kernel of BLAS and of many scientific codes is the matrix multiplication, and the efficient development of other higher-level linear algebra packages (LAPACK [7], ScaLAPACK [8], PLAPACK [9], HeteroScaLAPACK [10], PLASMA [11], etc.) is directly based on that of the matrix multiplication routine. The BLAS library is used in the form of packages implemented by the vendors (Intel MKL [12], IBM ESSL [13], etc.) or free implementations (ATLAS [2], Goto BLAS [14], etc.). The latest versions are multithreaded and can be used efficiently in multicore systems, where it is easy to obtain parallel versions from sequential codes using the shared-memory parallel programming environment OpenMP [15], and so scientific codes are currently being developed in OpenMP versions. Usually one thread in an OpenMP program calls to a multithreaded BLAS routine, so obtaining an implementation with nested parallelism. This two-level parallelism introduces more complexity in the program and makes the efficient use of the matrix multiplication more difficult due to possible interactions between OpenMP and the threaded BLAS. In [16] this interaction is empirically analyzed in order to design a model of the execution time in NUMA platforms of this kernel with two-level parallelism (`2L-dgemm`). Now, our goal is to measure the influence of an auto-tuned two-level parallel kernel, like `2L-dgemm`, in the execution conditions of scientific software in NUMA platforms. Therefore, this paper empirically analyzes the behaviour in NUMA systems of matrix factorizations by blocks, comparing implementations which directly use the multithreaded `dgemm` routine of BLAS with nested parallelism schemes using `2L-dgemm`.

We developed an Automatic Tuning System (ATS) for linear algebra routines in distributed memory environments [17]. The process performed by the ATS for each routine includes its design and installation. In the design phase, a theoretical model of the execution time of the routine is built. During the installation phase, thanks to a small set of experiments, this model becomes theoretical–experimental for the platform in use. Finally, in the execution phase, when the user decides to solve a particular problem, the model is used to take the appropriate decisions on how to execute the routine. 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 these two levels of parallelism. This methodology is explained in detail through matrix multiplication and LU factorization and, after that, it is shown how it can be extended to other linear algebra routines.

The rest of the paper is organized as follows. Section 2 briefly describes the characteristics of the systems and software used in the experiments. The empirical analysis of the behaviour of the kernel matrix multiplication in different NUMA platforms is presented in Section 3. An automatic tuning method based on modeling the execution time of linear algebra routines on NUMA platforms is described in Section 4, and Section 5 illustrates the application of the method to the matrix multiplication and an LU factorization. Section 6 shows some experimental results when the methodology is applied in different computational systems with new matrix factorizations (QR and Cholesky). Finally, in Section 7 some related works are compared with our approach, and in Section 8 the conclusions are summarized and some possible extensions of the work are considered.

2. Execution environment

In this section the characteristics of the systems and software used in the experiments are briefly described.

2.1. Computational systems

Different NUMA platforms have been used throughout this work, and experimental results are shown for three of them with different processors and number of cores ranging from 24 to 128. So, the conclusions will not be platform-dependent, and a more general picture of the behaviour of the routines in that type of systems is obtained, so facilitating the prediction for future, bigger systems. The platforms considered are shared-memory systems where the basic components are multicore computers:

- **Ben** is part of the system Ben-Arabí of the Supercomputing Center of the Fundación Parque Científico of Murcia. It comprises a shared-memory system with 128 cores and 1.5 TB memory. It is an HP Integrity Superdome with NUMA architecture, based on a hierarchical composition with crossbar interconnection. Its architecture has two basic components: the computers and two backplane crossbars. Each computer is an SMP with four CPUs dual core Itanium-2 and an ASIC controller to connect the CPUs with the local memory and the crossbar commuters. The maximum memory bandwidth in a computer is 17.1 GB/s and with the crossbar commuters 34.5 GB/s. So, access to the memory is non uniform and, furthermore, when programs are run in this system, the user does not control where the threads are assigned, so there are significantly different costs in the access to the shared-memory.

- **Saturno** belongs to our research group at the University of Murcia. It is a server SYS-8026B-TRF 2U supermicro, with 4 nodes Intel six-core NEHALEM-EX 6C E7530, so a total of 24 computing cores, and 32 GB of memory. Our aim by including this machine in the study has been to test our proposal in a small platform where the differences between the access times to the different memory locations are smaller than in the other platforms, making it more difficult to obtain important improvements with any automatic method.
- **Joule** belongs to the University Jaume I of Castellon. It is a NUMA system with 64 cores, 4 AMD Opteron 6276 (16 cores) processors, 2.3 GHz, and 64 GB of shared-memory. The theoretical and experimental basis of our proposal was set on the two above platforms. Therefore, our aim by including this machine in the experimental study is to show the viability of the proposal in another platform with a different architecture.

Hyperthreading is disabled in **Ben** and **Joule**, so it has been disabled in **Saturno** to ensure comparable results.

2.2. The software

Two different implementation schemes of matrix factorizations by blocks are compared: traditional schemes directly using the multithreaded `dgemm` routine of BLAS, and improved schemes, using `2L-dgemm`, with nested parallelism, where the matrices A and $B \in R^{n\times n}$ can be multiplied with two-level parallelism by generating q OpenMP threads, where each multiplies a block of adjacent rows of matrix A by matrix B, and by establishing a number of threads (p) to be used in the matrix multiplication in each OpenMP thread. Nested parallelism must be allowed with the environment variable `OMP_NESTED = true` or the function `omp_set_nested (1)`. In both cases, the kernel used in the experiments is the double precision routine `dgemm` of BLAS. Similar results are also obtained with basic routines in different libraries (MKL, ATLAS and Goto BLAS). In Section 6, the results shown are those obtained with the BLAS implementation of the Intel MKL toolkit version 10.2 in Ben and Saturno and 10.3 in Joule. Additional tests are performed with other libraries, and a summary appears in Section 7. This library is multithreaded, and parallelism can be achieved merely by calling the routine with the desired number of threads, which can be established with the environment variable `MKL_NUM_THREADS` or in the program with the function `mkl_set_num_threads`. When dynamic parallelism is enabled (with the environment variable `MKL_DYNAMIC = true` or in the program with the function `mkl_set_dynamic(1)`), the number of threads to use in the execution of the `dgemm` routine is decided by the system, and this is less than or equal to that established. To enforce the use of the number of threads indicated, the dynamic parallelism must be turned off (`MKL_DYNAMIC = false` or `mkl_set_dynamic (0)`).

The C compiler used was Intel icc version 11.1 in Ben, 12.0 in Saturno and 12.1 in Joule. Two-level parallelism is achieved by generating a number of OpenMP threads and by calls to the multithread BLAS library. The compiler optimization used was -O3.

3. Motivation: previous computational results

There are some software packages like MKL that are perfectly adapted to multithreading execution on generic multicore platforms, but when they run on large NUMA systems the performance achieved is far from the maximum achievable for not very large problems with many cores. Normally the routines of these libraries are called inside parallel programs, so they have two parallelism levels, one corresponding to the parallelism of program calling the routines and the other the intrinsic parallelism in the routines the program calls. Different numbers of threads can be used at the two parallelism levels, and the sizes of the matrices the basic routines work with can be not very large given the distribution of data between the threads at the program level. So, the low performance of the basic routines when working with not very large volume of data propagates to the whole program. To alleviate this problem, an autotuning methodology for selection of the number of threads to use at each parallelism level can be used. The methodology should take the decisions on the basis of the sizes of the subproblems generated for each thread in the program and the efficiency of the basic routines when working with this volume of data.

So, we begin by analyzing the behaviour of the `dgemm` MKL routine when it is executed in NUMA systems. Fig. 1 shows the speed-up achieved when varying the matrix size and the number of threads. The sizes used in the experiments are small in comparison with the memory of the computational systems, but the routine will be used inside parallel programs, where the data are divided to be assigned to different threads, which will work with smaller amounts of data. The optimum numbers of threads, which the lowest execution time is achieved with, changes from one platform to another due to running restrictions and to the characteristics of the systems, and the maximum speed-up is far from the number of cores in the system and is achieved with a number of threads lower than the number of cores, which indicates that in a parallel code calling the matrix multiplication routine it could be preferable to use two level parallelism and to select the number of threads to use in the basic routine and the number of OpenMP threads.

Figs. 2 and 3 show the speed-up achieved with different combinations of OpenMP and MKL threads and with the dynamic selection of MKL threads enabled and disabled. It seems that the number of MKL threads dynamically selected (Fig. 2) is just one when more than one OpenMP threads are running, so the dynamic selection of threads by MKL does not work when there is an upper parallel level, like OpenMP. On the other hand, when the number of MKL threads is established in the program by disabling the dynamic selection option (Fig. 3), bigger speed-ups are obtained. So, when a large number of cores is

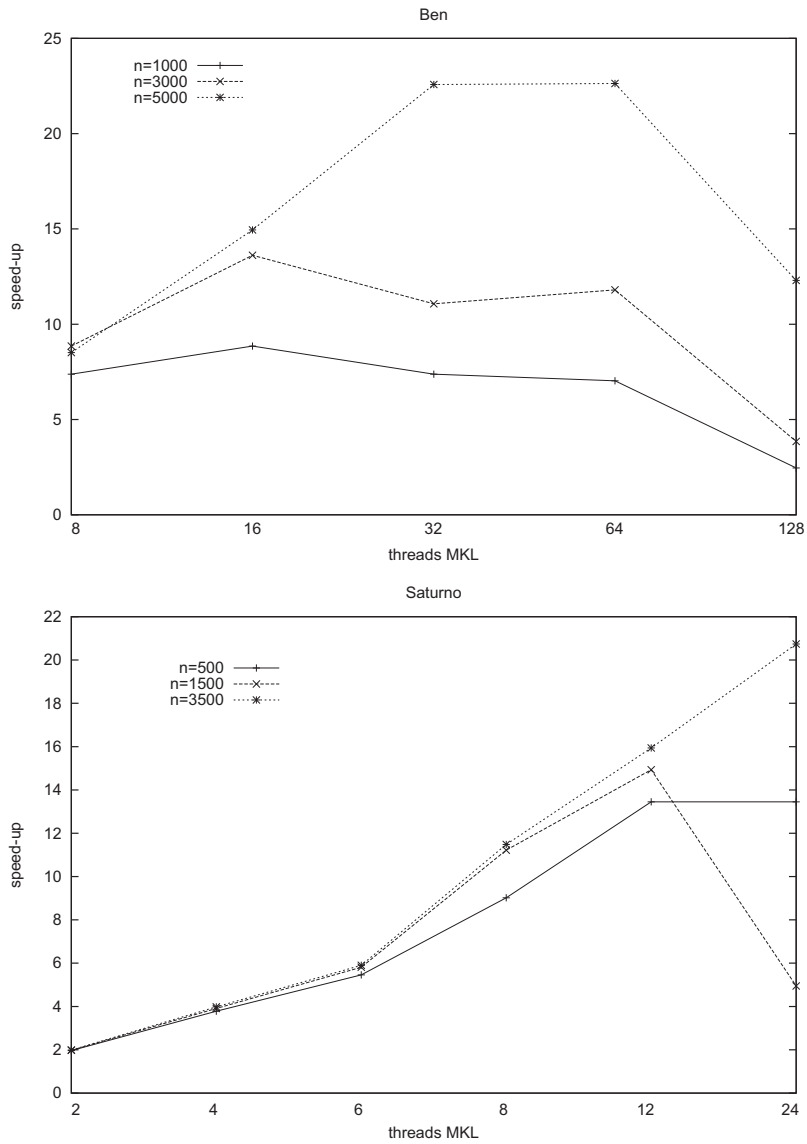

Fig. 1. Speed-up obtained with multithreaded MKL in NUMA systems (Ben and Saturno), varying the matrix size and the number of threads.

available, it seems a good option not to use dynamic selection of threads in MKL and to select a tuned combination of threads at the two parallelism levels.

Similar speed-up behaviors have been observed with this basic routine in other libraries like ATLAS and Goto BLAS (a summary of them is shown in Section 7). For this reason, the main experimental support for the general explanation of our proposal (Section 4) was performed with MKL toolkit (Sections 5 and 6).

4. Automatic tuning of a routine

For each routine, the core of our Automatic Tuning System (ATS) contains an execution time model of the routine. The model includes the characteristics of the platform (hardware + basic installed libraries) like system parameters (SP), and a set of algorithmic parameters (AP), whose values should be appropriately chosen by the ATS in order to reduce the execution time of the routine [17]:

$$T_{exe} = f(SP, n, AP) \tag{1}$$

The general process of automatic optimisation of routines following this methodology entails three phases:

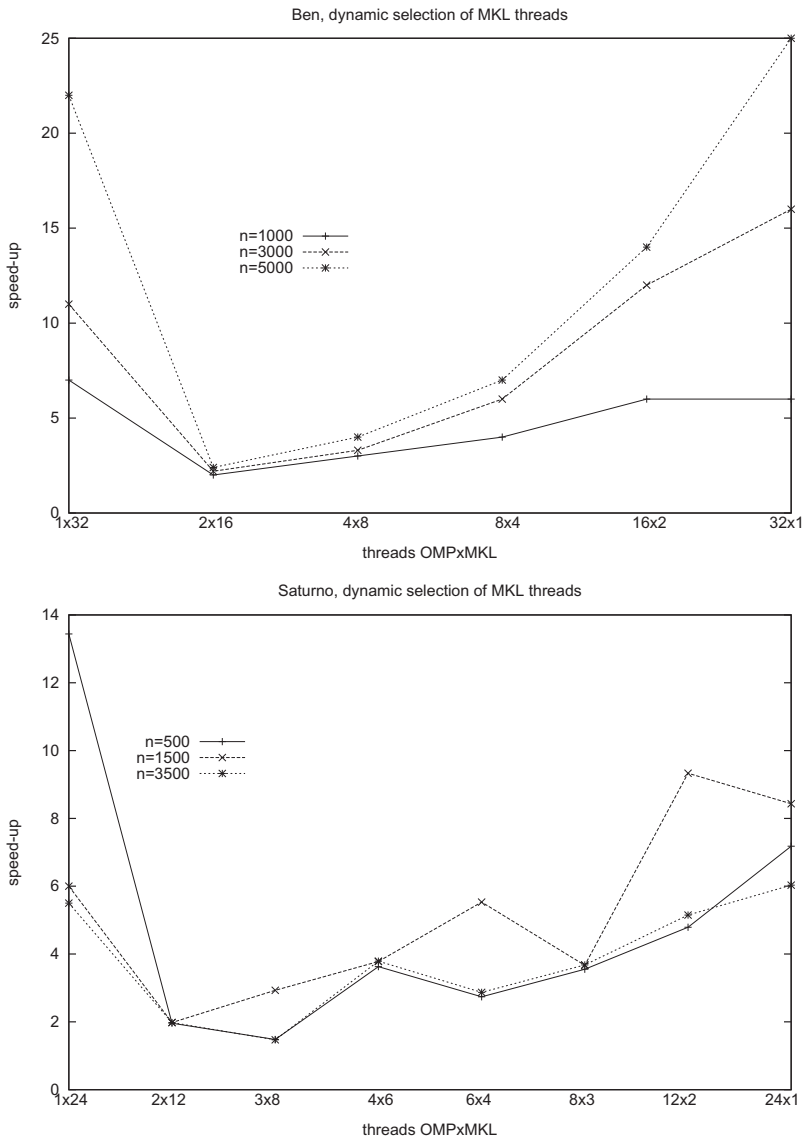

Fig. 2. Speed-up obtained in NUMA systems (Ben and Saturno), varying the matrix size, the number of OpenMP and MKL threads with dynamic selection of MKL threads.

- A design phase, where the theoretical model of the execution time is synthesized by the routine designer.
- An installation phase, where the SP values are estimated experimentally for the platform where the installation is performed.
- An execution phase, where the AP values are selected automatically in order to reduce the execution time.

In general, to perform a basic arithmetic operation inside a routine, the probability of finding the operands in the CPU closest memory (first cache levels) is proportional to the data locality of the algorithm implemented in this routine. For this reason, in [17,20], the averaged time to perform a basic arithmetic operation (including memory access time) is kept within a specific parameter $k_{routine}$, in a simple but effective way of taking into account the data locality of the routine.

Since our methodology estimates the parameters values obtained at installation time, it is likely that the system state (CPU load and network traffic) at the moment the routines are to be used will be quite different than at installation time. This may lead to the use of inaccurate parameters and then to execution times far from the optimum. Therefore, we have not considered the interference with other programs running in the system. This would difficult even more to obtain a simple-satisfactory model, and so the executions have been carried out in exclusion. The adaptation of the methodology to changing environments can be studied in a near future, as we did in homogeneous and heterogeneous clusters [18].

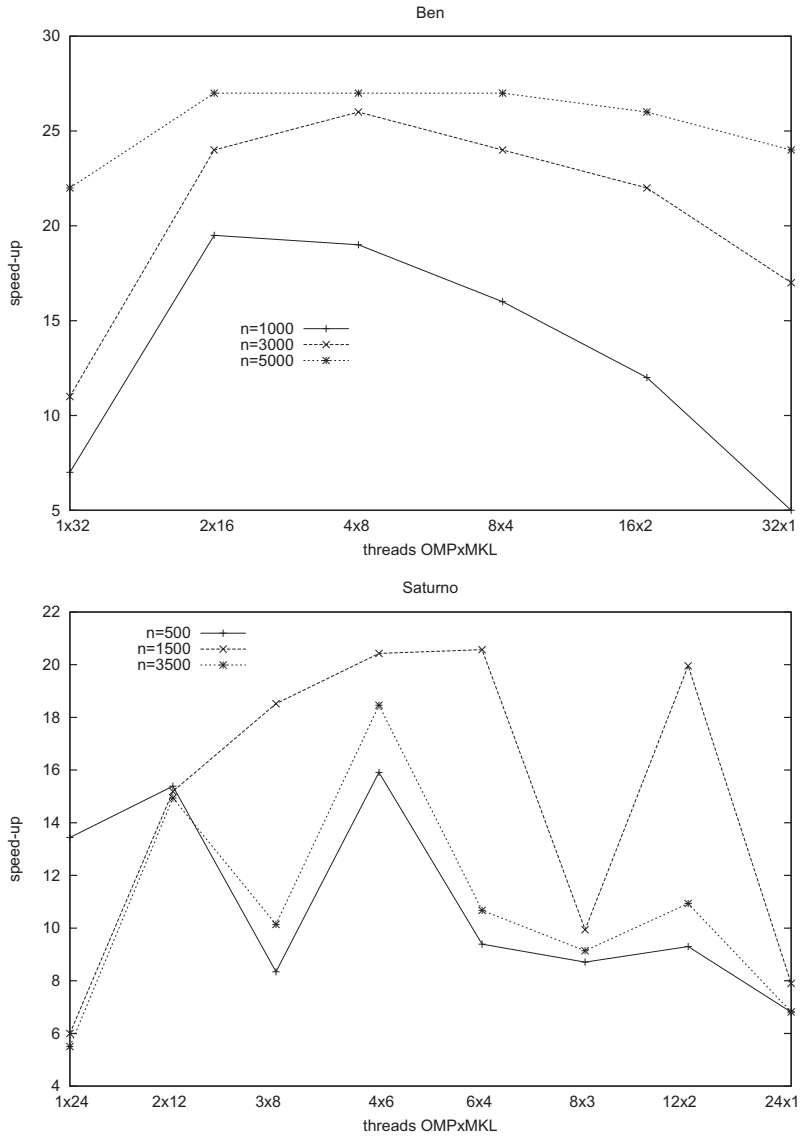

Fig. 3. Speed-up obtained in NUMA systems (Ben and Saturno), varying the matrix size, the number of OpenMP and MKL threads without dynamic selection of MKL threads.

In this work, from this general scheme, an adaptation is necessary of the ATS to the particular characteristics of large NUMA platforms, where there is a shared RAM memory space but with non uniform data access time. In this way, the execution time model of each routine (Eq. (1)) must cover this circumstance, where in each arithmetic operation the access time depends not only on the routine data locality, but also on the relative position in the shared memory space of the data and on the processor where the operation is performed. These data may or may not be in the local memory (RAM + cache) closest to the processor that operates with them, and it must be taken into account that the interconnection network could be non homogeneous, so these data could be at different distances from the processor that needs them. Therefore, the access time must be modeled with a distributed vision of the memory, from the point of view of the processor that performs the operation, with a first location level formed by the local memory of this processor and a series of location levels formed by the local memory of the processors that are at different distances in the interconnection network.[1] To do so, the system parameter representing the arithmetic cost in Eq. 1 will be substituted by a set of parameters corresponding to different memory levels-allocations, whose number and influence in the model will be experimentally calculated.

[1] In this proposal we use the term "memory location level" to refer to the distance in the NUMA intraconection network from the computing core to the memory node where these data are allocated; that is, for a given computing core, the first level corresponds to the data allocated in local memory of this core node, the second level to the data allocated in another node, but directly connected with this core node,... This terminology should not be confused with the typical architectural memory levels (cache L1, L2, L3, main memory, secondary memory).

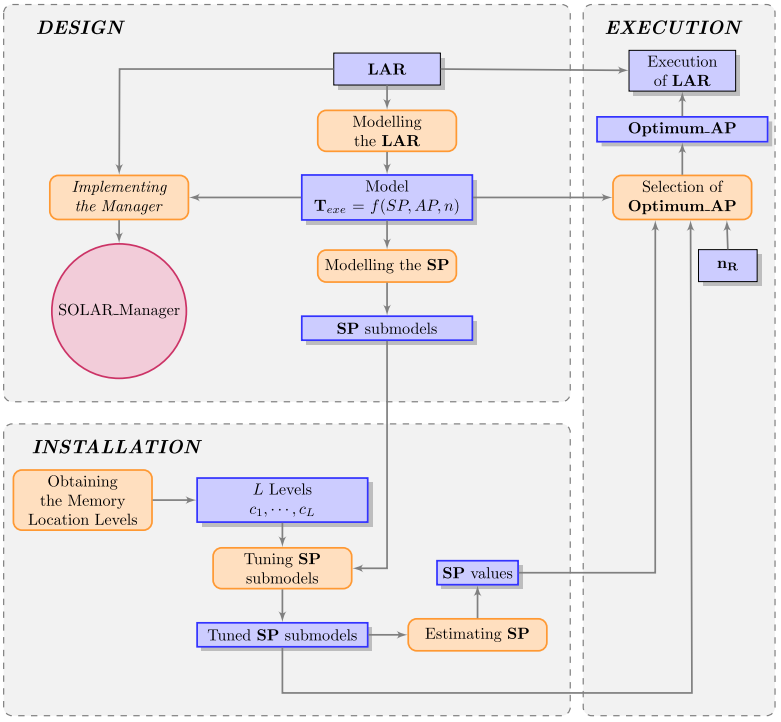

Fig. 4. Life-cycle of a Self Optimized Linear Algebra Routine (SOLAR) for NUMA platforms.

Our proposal focuses on auto-tuned routines that will be used by non expert users. These users do not know either the characteristics of the NUMA platforms or how to drive the mapping of the threads in the cores, or how to indicate the allocation of data for the threads. For these reasons, to design the execution time model, we suppose an automatic placement of the threads as packed as possible and an optimal allocation of the data as close as possible to the cores used for the threads, that is, an exclusive use of the platform by our routines.

It is important to emphasize that the models proposed in this work do not aim to detail the architectural behavior of the routines, but are just used to predict the execution time of these routines in different situations, with different values for some adjustable parameters. Therefore, the main goal is to design the models as simply as possible to be a versatile tool to select appropriate values for the parameters. In this work, optimized basic routines (MKL, ATLAS,...) are integrated in higher level OpenMP codes. Therefore, the only algorithmic parameter considered has been the number of threads because other possible optimizations, like tile or block sizes, are included in those basic routines.

4.1. Life-cycle of an auto-tuned routine

In this subsection, the design, installation and execution of a self-optimized linear algebra routine (SOLAR) for NUMA platforms is described step by step, following the scheme in Fig. 4. Details of the joint design-installation process for routines in NUMA platforms are discussed in the next section with the matrix multiplication and a LU factorization.

4.1.1. Design phase

This process is performed by the designer of the linear algebra routine (LAR designer) only once. The main tasks are:

- The LAR is created if a new library is being designed. Otherwise, it is not necessary.
- The complexity of the LAR is studied, obtaining an analytical model of its execution time similar to that of Eq. (1), but in the NUMA case the number of memory levels influencing the behaviour of the routine will be experimentally obtained in the installation phase.
- Each SP is modeled with an equation where the influence of the distributed structure of the memory on the data access time is reflected.
- The SOLAR_Manager is created. This is the engine of the auto-tuned routine. It is in charge of managing all the information and it is also the interface with the exterior of the SOLAR. At installation time, it will extract the information about the memory structure of the platform and it will give the order to estimate the SP values. At execution time, using the analytical model, it will decide the values for the AP and, finally, it will call the LAR.

4.1.2. Installation phase

When the system manager starts the installation process of a SOLAR in a specific platform he calls to the SOLAR_Manager that performs the following tasks:

- Information about the influence of memory allocation in the routine is obtained by executing the LAR for a fixed size, n, and different numbers of threads, p. From the experimental times and the basic execution time model of the routine (Eq. (1)), the value for its general basic system parameter k_{LAR} is calculated directly. This value should remain relatively constant along each range of p that corresponds to each location level of the memory, which allows estimation of the values for the various location levels of the memory, L, and the number of cores that have a similar access speed to each level, $c_1, \ldots, c_L$.
- The theoretical models of the SP are tuned according to the memory structure of the platform. In this way, the submodel for each system parameter now reflects the different data access time, depending on the location level where they are found.
- The value for each basic SP is obtained for each location level $(1,\ldots,L)$ with executions of the LAR for a fixed problem size, n, and a number of threads, p, that corresponds to each level $(p \leqslant c_1,\ c_1 < p \leqslant c_2, \ldots, c_{L-1} < p \leqslant c_L)$.

4.1.3. Execution phase

When a SOLAR is called to solve a problem of size n_R, the SOLAR_Manager carries out the following tasks:

- Using the Model of the LAR, with $n = n_R$, the SOLAR_Manager looks for the combination of AP values that, taking into account the SP values calculated at installation time, provide the lowest execution time.
- Finally, the LAR is called with the problem size n_R and the Optimum AP values selected previously.

5. Adaptation of the auto-tuning methodology for linear algebra routines in NUMA systems

This section describes the adaptation for NUMA systems of the proposed auto-tuning methodology. The adaptation is explained in detail for the matrix multiplication, and then for a higher-level routine (an LU factorization) which uses the two-level matrix multiplication as basic component.

5.1. The matrix multiplication

5.1.1. Design phase

In this phase, the complexity of the `dgemm` routine is studied, obtaining an analytical model of its execution time. This model has to be as simple as possible for it to be a versatile tool to select the most appropriate values for its adjustable parameters.

First, taking just the MKL routine `dgemm` without considering the possibility of generating OpenMP threads, the model of the execution time is:

$$T_{dgemm} = \frac{2n^3}{p} k_{dgemm} \tag{2}$$

where n is the problem size (we consider square matrices for simplicity), p is the number of threads that work together in the operation performed by the MKL routine `dgemm`, and k_{dgemm} is a system parameter that corresponds to the time to carry out a basic operation (addition or multiplication) between two double precision numbers inside `dgemm`. The value of k_{dgemm} includes the time to load the operands from the memory, perform the operation in the CPU and store the result. These operands may be in different RAM memory locations of the platform, with different access times from the thread that performs each basic operation. Therefore, the k_{dgemm} value can be modeled as a weighted average between two values k_{dgemm_NUMA} and $k_{dgemm\_M_1}$:

$$k_{dgemm} = \alpha k_{dgemm\_NUMA}(p) + (1-\alpha) k_{dgemm\_M_1} \tag{3}$$

where k_{dgemm_NUMA} is the time to perform the operation when the operands are in any memory location of the NUMA platform; and $k_{dgemm\_M_1}$ is the time to perform the operation when the operands are in the memory location closest to the core.

The α factor is a heuristic approximation obtained experimentally to represent the weight of the parameter k_{dgemm_NUMA} in the total k_{dgemm}. It can be considered directly proportional to the use each one of the p threads makes of data assigned initially to any of the other $(p-1)$ threads. This factor can be considered inversely proportional to the reusing data degree of the routine, because the data reusing increases the possibility of finding the operating data in the closest location memory of each core (`dgemm` accesses an order of n^2 data to perform an order of n^3 operations, so, the reusing degree will be $n^3/n^2 = n$, and obviously, other routines have different degrees):

$$\alpha = min\left\{1, \frac{p(p-1)}{n}\right\} \tag{4}$$

Therefore, the model collects how the data locality of the routine influences on its execution time with p threads. For this heuristic value, when the matrix size increases α tends to zero, and the data are considered distributed in the memory close to the threads (value $k_{dgemm\_M_1}$ on Eq. (3)), and when the number of threads increases and the matrix size is not very large α tends to one, so the memory allocation of the data has more influence.

On the other hand, the parameter $k_{dgemm\_NUMA}(p)$ can be divided in a set of submodels depending on the number of location levels of the memory. For a platform with L location levels (the location levels are detected experimentally in the installation phase), where c_l cores have a similar access speed to the level l, with $1 \leqslant l \leqslant L$, then $k_{dgemm\_NUMA}(p)$ can be modeled as:

if $0 < p \leqslant c_1$:

$$k_{dgemm\_M_1}$$

else if $c_1 < p \leqslant c_2$:

$$\frac{c_1 k_{dgemm\_M_1} + (p - c_1) k_{dgemm\_M_2}}{p}$$

else if $c_2 < p \leqslant c_3$:

$$\frac{c_1 k_{dgemm\_M_1} + (c_2 - c_1) k_{dgemm\_M_2} + (p - c_2) k_{dgemm\_M_3}}{p}$$

In general, if $c_{L-1} < p \leqslant c_L$:

$$\frac{\sum (c_l - c_{l-1}) k_{dgemm\_M_l} + (p - c_{L-1}) k_{dgemm\_M_L}}{p} \tag{5}$$

For example, if $c_2 < p \leqslant c_3$, for a specific thread t situated in the core x, from each p data accessed by t to operate with them, c_1 data will be in the closest local memory (first location level), $(c_2 - c_1)$ in the following level, that is, distributed along the $(c_2 - c_1)$ cores that share the second location level with x, but not the first level. Finally, the rest of the p data, $(p - c_2)$, will be at the third location level from the point of view of core x.

At this point, it is possible to model the complete routine that uses two-level parallelism by generating q threads OpenMP, with each one multiplying a block of adjacent rows of matrix A by the matrix B, and by establishing a number of threads (p) to be used in the matrix multiplication in each OpenMP thread. With $P = q \times p$, the model is:

$$T_{2L\_dgemm} = \frac{2n^3}{P} k_{2L\_dgemm} \tag{6}$$

The model for the parameter k_{2L_dgemm} is as described previously for k_{dgemm}, but taking into account that now a total of $P = q \times p$ threads are interacting globally instead of only p:

$$k_{2L\_dgemm} = \alpha k_{2L\_dgemm\_NUMA}(P, p) + (1 - \alpha) k_{dgemm\_M_1} \tag{7}$$

with:

$$\alpha = min\left\{1, \frac{P(P-1)}{n}\right\} \tag{8}$$

and where the model for the parameter $k_{2L\_dgemm\_NUMA}$ is:

$$k_{2L\_dgemm\_NUMA}(P, p) = \frac{\frac{n^2}{q} k_{dgemm\_NUMA}(p) + n^2 k_{dgemm\_NUMA}(P)}{\frac{n^2}{q} + n^2} \tag{9}$$

that is, an average between k_{dgemm_NUMA} when p threads are working with n^2/q data and k_{dgemm_NUMA} when P threads are working with n^2 data. This is because p threads are interacting with the data of matrix A (each one of the q sets of p threads shares one of the q blocks of adjacent rows of A inside each `dgemm` call. Each of these blocks has n^2/q data), whereas with the data of matrix B the total P threads are interacting (all the p threads inside each of the q `dgemm` calls access the complete matrix B, that is, n^2 data).

Finally, as described previously for the one-level version (Eq. (5)), with x threads, if $c_{L-1} < x \leqslant c_L$, $k_{dgemm\_NUMA}(x)$ is:

$$\frac{\sum (c_l - c_{l-1}) k_{dgemm\_M_l} + (x - c_{L-1}) k_{dgemm\_M_L}}{x} \tag{10}$$

5.1.2. Installation phase

Information about the interaction between the data allocation of the algorithm and the memory structure of the platform is experimentally obtained when obtaining the values of L and $c_1, \ldots, c_L$. Hence, this phase complements the design phase and the theoretical model is not completed until the installation finished. The routine `dgemm` is executed for a fixed size, n, and different numbers of threads, p. With the experimental times obtained and the basic execution time model of the

Table 1
Installation phase in Ben: obtaining information about the memory location levels, $n = 1000$.

Number of threads p	Measured execution time T_{dgemm} (s)	$k_{dgemm} = pT_{dgemm}/2n^3$ (ms)	Memory location levels
1	0.390	0.20	1
4	0.096	0.19	
8	0.048	0.19	
12	0.045	0.27	1,2
24	0.024	0.29	
32	0.015	0.24	
36	0.078	1.41	1,2,3
50	0.056	1.39	
64	0.044	1.41	
68	0.056	1.89	1,2,3,4
98	0.035	1.72	
128	0.028	1.79	

routine (Eq. (2)), the k_{dgemm} value is calculated. This value will remain relatively constant along each range of p that corresponds to each location level of the memory, which allows estimation of the values for L and $c_1, \ldots, c_L$.

In Ben, the SOLAR_Manager experimentally obtains (Table 1) 4 allocation levels for the data managed by the routine in the memory hierarchy, $L = 4$, with $c_1 = 8,\ c_2 = 32,\ c_3 = 64$ and $c_4 = 128$. Therefore, the model for the SP k_{dgemm_NUMA} can be tuned for this platform, according to the number of threads, p, as:

If $0 < p \leqslant 8$:

$$k_{dgemm\_M_1}$$

else if $8 < p \leqslant 32$:

$$\frac{8k_{dgemm\_M_1} + (p-8)k_{dgemm\_M_2}}{p}$$

else if $32 < p \leqslant 64$:

$$\frac{8k_{dgemm\_M_1} + 24k_{dgemm\_M_2} + (p-32)k_{dgemm\_M_3}}{p}$$

else if $64 < p \leqslant 128$:

$$\frac{8k_{dgemm\_M_1} + 24k_{dgemm\_M_2} + 32k_{dgemm\_M_3} + (p-64)k_{dgemm\_M_4}}{p} \tag{11}$$

Similarly, 4 location levels are obtained for the routine in Saturno (Table 2), with $c_1 = 6,\ c_2 = 12,\ c_3 = 18$ and $c_4 = 24$. It is important to consider that in this platform the differences between these levels are smaller than in Ben. The model for k_{dgemm_NUMA} can be tuned for this platform, according to the number of threads, p, as:

If $0 < p \leqslant 6$:

$$k_{dgemm\_M_1}$$

else if $6 < p \leqslant 12$:

$$\frac{6k_{dgemm\_M_1} + (p-6)k_{dgemm\_M_2}}{p}$$

else if $12 < p \leqslant 18$:

$$\frac{6k_{dgemm\_M_1} + 6k_{dgemm\_M_2} + (p-12)k_{dgemm\_M_3}}{p}$$

else if $18 < p \leqslant 24$:

$$\frac{6k_{dgemm\_M_1} + 6k_{dgemm\_M_2} + 6k_{dgemm\_M_3} + (p-18)k_{dgemm\_M_4}}{p} \tag{12}$$

When a routine is installed in a specific platform, the SP values to calculate are $k_{dgemm\_M_1}, \ldots, k_{dgemm\_M_L}$. The SOLAR_Manager performs the estimation of the values of these system parameters by executing the routine `dgemm` for a fixed problem size, n (preferably with a small n because we are estimating the cost for routines which will be used inside parallel codes also to reduce the installation time) and different number of threads. With these execution times and using the basic model of

Table 2
Installation phase in Saturno: obtaining information about the memory location levels, $n = 1000$.

Number of threads p	Measured execution time T_{dgemm} (s)	$k_{dgemm} = pT_{dgemm}/2n^3$ (ms)	Memory location levels
1	0.286	0.14	1
6	0.047	0.14	
7	0.043	0.15	1,2
12	0.025	0.15	
13	0.026	0.17	1,2,3
18	0.018	0.16	
19	0.019	0.19	1,2,3,4
24	0.016	0.19	

the routine (Eq. (2)), the successive values of k_{dgemm} for the different location levels of the memory will be obtained (Tables 1 and 2). Finally, taking into account the values of $c_1, \ldots, c_L$ and using the corresponding equations of the model for k_{dgemm_NUMA} (Eqs. (11) and (12)), the values of $k_{dgemm\_M_1}, \ldots, k_{dgemm\_M_L}$ are calculated (Tables 3 and 4).

At this point, the initial theoretical model of the execution time of the routine `2L-dgemm` (Eqs. (11) and (12)) has become an empirical–theoretical model, thanks to this experimental estimation of the SP values (Tables 3 and 4).

5.1.3. Execution phase

Finally, in the execution phase, when the user wants to solve a specific problem with size n, the SOLAR_Manager takes the model of the routine, with the SP values calculated for this platform and the value n, and directly selects the most appropriate values for the AP, that is, the number of OpenMP threads, q, and MKL threads, p, between a set of possible combinations. Therefore, the overhead of this automatic selection of the AP values is negligible in terms of the total execution time of the routine.

5.2. Automatic tuning of a higher-level routine: LU factorization by blocks

In this subsection, the application of the methodology to high level routines that use an auto-tuned kernel is described. An LU factorization is used to illustrate the methodology, and the experimental section also shows results with the QR and Cholesky factorizations when the same methodology is applied to them. The LU factorization has the same scheme as the LAPACK routine `dgetrf` that uses the MKL kernels `dgemm`, `dtrsm` and `dgetf2`. Two versions are compared: the version with a more standard scheme, that calls directly to the kernel `dgemm` from the BLAS implementation of the MKL package, and a version with an improved scheme, that calls to `2L-dgemm`, adapted to NUMA platforms. The rest of the LU routine is unaltered.

5.2.1. Design phase

As with the matrix–matrix multiplication, the model for the LU factorization by blocks is described by the routine designer:

$$T_{blu} = \frac{2n^3}{3P} k_{xdgemm} + \frac{bn^2}{P} k_{dtrsm} + \frac{b^2 n}{3P} k_{dgetf2} \tag{13}$$

Table 3
Installation phase in Ben: obtaining specific system parameter values.

Memory location level, i	Number of threads, p	$k_{dgemm\_M_i}$
1	$0 < p \leqslant 8$	0.19
2	$8 < p \leqslant 32$	0.33
3	$32 < p \leqslant 64$	1.44
4	$64 < p \leqslant 128$	1.90

Table 4
Installation phase in Saturno: obtaining specific system parameter values.

Memory location level, i	Number of threads, p	$k_{dgemm\_M_i}$
1	$0 < p \leqslant 6$	0.14
2	$6 < p \leqslant 12$	0.16
3	$12 < p \leqslant 18$	0.18
4	$18 < p \leqslant 24$	0.19

where n is the problem size (dimension of the matrices), b is the computing block size, P is the number of threads, k_{xdgemm} is a system parameter that corresponds to the time to carry out a basic operation (addition or multiplication) between two double precision numbers inside the matrix–matrix multiplication, so it corresponds to k_{dgemm} (Eq. (3)) if the kernel used is that provided by MKL directly, or to k_{2L_dgemm} (Eq. (7)) if the proposed two-level kernel is used. Finally, k_{dtrsm} and k_{dgetf2} are the system parameters that correspond to the time to carry out a basic operation (addition or multiplication) between two double precision numbers inside the kernels `dtrsm` and `dgetf2`.

5.2.2. Installation phase

The experimental process to obtain the information about the memory structure of the platform has been previously performed in the installation process of the matrix–matrix multiplication, so all the information about the memory location levels is calculated just once and can then be used by the other routines. When tuning the submodel of the general SP k_{dgemm} (Eq. (3)) or k_{2L_dgemm} (Eq. (7)), the basic SP values to calculate would be $k_{dgemm\_M_1}, \ldots, k_{dgemm\_M_L}$. The SOLAR_Manager of this routine can reuse the information obtained by the SOLAR_Manager of `dgemm` when this kernel was installed. The same operation can be performed to obtain the information for the other two general SP k_{dtrsm} and k_{dgetf2}, with their respective kernels.

5.3. Execution phase

A user invokes the LU routine in order to solve a specific problem with size n in a platform. Then, the SOLAR_Manager takes the experimental-theoretical model of the routine, with the SP values calculated for this platform and the value n, and selects the most appropriate number of OpenMP threads, q, and MKL threads, p, from a set of possible combinations.

6. Experimental results

The methodology explained with the matrix multiplication and the LU factorization can be extended to different linear algebra routines, with different basic libraries and in different computational systems. First, experimental results obtained with these two routines and the two systems so far considered are shown, then, complementary experiments in another platform and with new matrix factorizations (QR and Cholesky) are described. In the next section (related work) a brief comparison with other basic libraries is included.

We have not considered the interference with other programs running on the system because this would make it even more difficult a simple-satisfactory model of the execution time [18]. Therefore, all the executions have been carried out alone.

For each experiment, five executions were carried out, and the minimum and maximum execution times discarded. Then the average of the remaining values is used.

6.1. Experiments with the matrix multiplication

In order to test the goodness of our proposal we look at the general vision of the behaviour of the matrix multiplication routine provided by this model. Figs. 5 and 6 show a comparison between the execution time of the routine `2L-dgemm` and the time predicted by the model with MKL as basic library. The model does not predict the behaviour of the routine exactly, and this was not its goal, rather it captures the tendencies of the execution time when the AP values change. The model collects the general growing and decreasing tendency of the time for different configurations of number of OpenMP threads, q, and MKL threads, p. This quality means the proposed methodology can take a general vision of the behaviour of the routine to decide appropriate values for the algorithmic parameters, q and p, that minimize the execution time.

Table 5 shows a comparison of different execution times obtained for different problem sizes in Ben and Saturno. The different columns correspond to the minimum time that could be obtained ideally with a perfect MKL oracle (MKL-ORA) that always generates the optimum two-level threads configuration (OpenMP $\times$ MKL) in any situation (including the possibility of using only MKL threads and just one OpenMP thread); the time obtained on generating as many MKL threads as cores available in the platform (MKL-AC); the time obtained with MKL with dynamic selection of threads activated (MKL-dyn); and the time obtained with the two-level thread configuration selected by our auto-tuning system in the execution phase (MKL-ATS). Our aim with the MKL-ATS is to improve automatically the efficiency that a normal user can obtain using MKL (MKL-AC or MKL-dyn), and we get execution times close to those with the MKL ideal execution (MKL-ORA).

In the bigger platform, Ben, it can be appreciated that the automatic selection of the AP values with the proposed auto-tuning method (MKL-ATS) always outperforms what a non-expert user could obtain (A non-expert user could decide to use the MKL routine with as many threads as available cores in the platform, MKL-AC, or to use the auto-tuning engine provided by MKL, MKL-dyn). In all the cases, MKL-ATS obtains execution times very close to those obtained with a perfect oracle for MKL (An MKL oracle would have the difficult task of guessing the optimum combination of number of OpenMP and MKL threads from 1 to 128, the number of available cores of the platform). However, in Saturno the situation is a bit different. Here, the model has an excellent behaviour, following the tendencies of the execution time (Fig. 5). However, due to the simplicity of this platform, where the differences between the memory locations are smaller than in the other platform, it is more difficult to obtain significant benefits in either case (MKL-ORA or MKL-ATS).

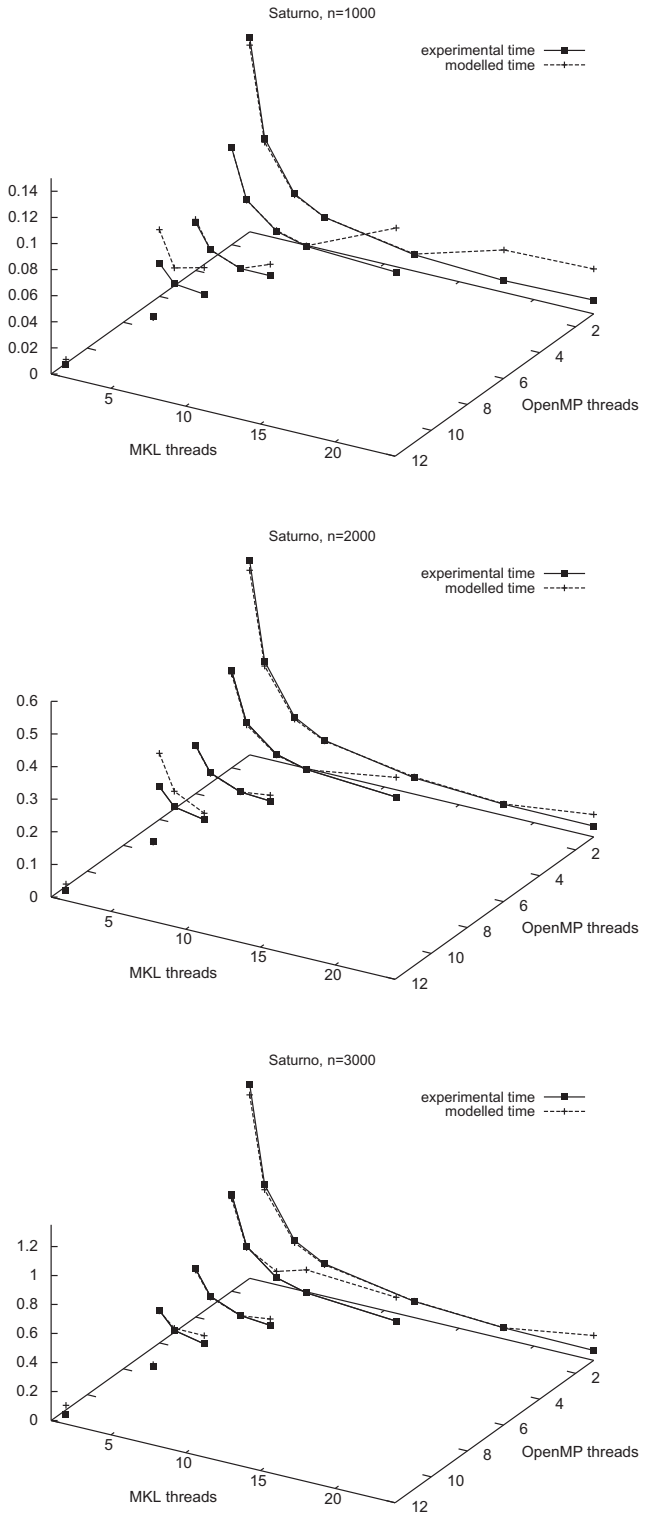

Fig. 5. Comparison of the execution time and the modeled time of the `2L-dgemm`, in the platform Saturno with MKL the basic library for different problem sizes.

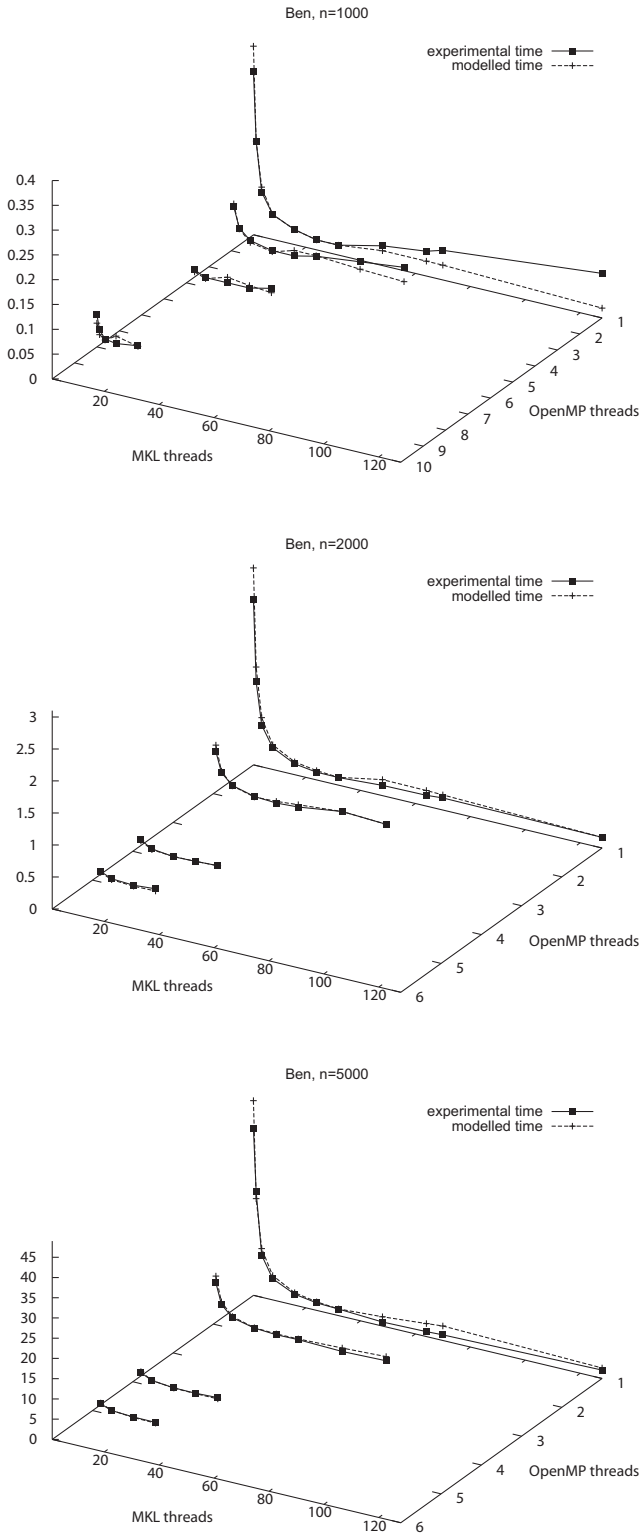

Fig. 6. Comparison of the execution time and the modeled time of the `2L-dgemm`, in the platform Ben with MKL the basic library for different problem sizes.

Table 5
Matrix–matrix multiplication kernel. Minimum time obtained generating the optimum number of threads (MKL-ORA), time obtained when generating as many MKL threads as available cores in the platform (MKL-AC), time obtained with MKL with dynamic selection of threads activated (MKL-dyn), and finally, time obtained using `2L-dgemm` with the thread configuration (between brackets) selected by our auto-tuning system (MKL-ATS). Times in seconds.

Size	MKL-ORA	MKL-AC	MKL-dyn	MKL-ATS
Saturno				
1000	0.014	0.044	0.049	0.016 (1×18)
2000	0.059	0.146	0.145	0.059 (2×12)
3000	0.119	0.141	0.146	0.127 (2×12)
4000	0.203	0.221	0.241	0.208 (2×12)
5000	0.251	0.335	0.333	0.306 (2×12)
Ben				
1000	0.024	0.091	0.098	0.012 (2×8)
2000	0.07	0.39	0.40	0.07 (4×16)
3000	0.23	0.82	0.81	0.23 (4×16)
4000	0.59	1.40	1.41	0.74 (4×32)
5000	1.12	2.11	2.10	1.44 (4×32)

6.2. Experiments with the LU factorization

Table 6 shows a comparison of different execution times obtained with the LU factorization for different problem sizes in the different platforms. The entries in the columns have the same meaning as in Table 5. It can be appreciated that, in general, the time obtained with the automatic selection of the AP values with the proposed auto-tuning method is always lower than that obtained using only the MKL routine with as many threads as available cores in the platform, or when MKL dynamically selects the number of threads (the two default options for a non-expert user). In Saturno the differences in the execution times for the different configurations are small, as befits the size and complexity of this platform. In Ben, the times with the auto-tuning method are very close to those obtained with a perfect oracle of MKL, which would have the hard task of selecting the optimum combination of OpenMP and MKL threads, between 1 and 128. In this platform, the differences with the default options (MKL-AC and MKL-dyn) are quite significant. So, in large systems, like Ben, the advantage of an auto-tuning methodology is more apparent.

6.3. Experiments with the QR and the Cholesky factorizations

In the previous subsections, experimental results obtained with the two routines and the two systems so far considered have been described. In this subsection, in order to show that this methodology can be extended to other linear algebra

Table 6
LU factorization by blocks. Minimum time obtained when generating the optimum number of threads (MKL-ORA), time obtained generating as many MKL threads as cores available (MKL-AC), time obtained with MKL with dynamic selection of threads (MKL-dyn), time obtained with the two-level thread configuration selected by our auto-tuning system (MKL-ATS). Times in seconds.

Size	MKL-ORA	MKL-AC	MKL-dyn	MKL-ATS
Saturno				
1000	0.06	0.06	0.06	0.12
2000	0.12	0.20	0.12	0.15
3000	0.26	0.26	0.27	0.33
4000	0.54	0.76	0.62	0.59
5000	0.88	1.00	0.98	0.88
6000	1.44	1.44	1.46	1.49
7000	1.98	2.19	2.22	1.98
8000	2.76	3.14	3.07	2.76
9000	4.00	4.42	4.50	4.00
10000	5.17	5.30	5.27	5.17
Ben				
1000	0.06	0.34	0.35	0.25
2000	0.17	0.79	0.70	0.42
3000	0.54	2.14	2.13	0.85
4000	0.95	3.39	3.35	1.32
5000	1.64	6.36	6.32	2.21
6000	2.63	8.32	8.34	2.83
7000	3.84	10.20	10.25	4.55
8000	5.19	14.49	14.45	5.39
9000	6.95	20.21	20.35	7.30
10000	8.12	22.28	22.21	8.70

routines and in different computational systems, complementary experiments in an additional medium size platform, Joule, and with new matrix factorizations, QR and Cholesky, are shown.

Tables 7 and 8 show a comparison of different execution times obtained with these factorizations for different problem sizes. The entries in the columns have the same meaning as in Table 5. We can observe that in a small size platform like Saturno, the tuning possibilities are limited and so MKL-ORA does not obtain significant improvements regarding MKL-AC. However, it can be appreciated that, in general, the time obtained with the automatic selection of the AP values with the proposed auto-tuning method (MKL-ATS) is always lower than that obtained by a normal user generating as many threads as cores available in the platform (MKL-AC).

Table 7
QR factorization by blocks. Minimum time obtained when generating the optimum number of threads (MKL-ORA), time obtained generating as many MKL threads as cores available (MKL-AC), time obtained with the two-level thread configuration selected by our auto-tuning system (MKL-ATS). Times in seconds.

Size	MKL-ORA	MKL-AC	MKL-ATS
Saturno			
1000	0.10	0.18	0.14
2000	0.46	0.52	0.46
3000	1.14	1.21	1.21
4000	2.15	2.15	2.19
5000	3.61	3.61	4.00
6000	5.40	5.40	5.40
7000	7.78	7.78	7.78
8000	10.64	10.85	10.64
9000	14.57	14.88	14.57
10000	19.47	19.80	19.47
Joule			
1000	0.21	0.47	0.18
2000	0.42	0.84	0.42
3000	1.18	1.43	1.18
4000	2.33	2.68	2.51
5000	4.01	3.92	4.23
6000	5.77	5.97	6.37
7000	6.69	8.60	8.37
8000	9.83	13.10	11.70
9000	14.27	15.90	14.27
10000	19.86	21.35	19.86

Table 8
Cholesky factorization by blocks. Minimum time obtained when generating the optimum number of threads (MKL-ORA), time obtained generating as many MKL threads as cores available (MKL-AC), time obtained with the two-level thread configuration selected by our auto-tuning system (MKL-ATS). Times in seconds.

Size	MKL-ORA	MKL-AC	MKL-ATS
Saturno			
1000	0.03	0.36	0.15
2000	0.09	0.24	0.15
3000	0.23	0.31	0.28
4000	0.45	0.54	0.49
5000	0.88	0.93	0.89
6000	1.45	1.56	1.46
7000	2.20	2.27	2.27
8000	3.10	3.23	3.20
9000	4.36	4.50	4.36
10000	5.96	6.12	6.00
Joule			
1000	0.06	0.33	0.09
2000	0.16	0.59	0.30
3000	0.40	0.93	0.67
4000	0.79	1.44	1.28
5000	1.23	2.08	2.08
6000	1.83	3.27	2.88
7000	2.82	4.35	3.73
8000	4.06	5.79	5.25
9000	5.24	7.70	6.64
10000	6.84	9.93	8.42

In a medium size platform like Joule, the tuning potential grows, so a MKL perfect oracle could obtain a clearer improvement in the execution time. Therefore, in this platform, the proposed auto-tuning method can perform a most useful task, reducing the default user execution time by about 10% in QR factorization and about 20% in Cholesky.

7. Related work

In recent years several groups have been working on the design of highly efficient parallel linear algebra libraries. These groups work in different ways: optimizing the library in the installation process for shared memory machines [2] or for message-passing systems [20]; analyzing the adaptation of the routines to the conditions of the system at a particular moment [21]; using poly-algorithms [22] and poly-libraries [23]; and redesigning LAPACK for platforms based on multi-core processors [11] and for heterogeneous/hybrid architectures [19]. A number of auto-tuning approaches are focused on modeling the execution time of the routine to optimize. The approach chosen by FAST [24] is an extensive benchmark followed by a polynomial regression to find optimal parameters for different routines in homogeneous and heterogeneous environments. Vuduc et al. [25] apply the polynomial regression in their methodology to decide the most appropriate version from variants of a routine. They also introduce a black-box pruning method to reduce the enormous implementation spaces. In the FIBER approach [26] 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 grades 1 to 5 are 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 is varied. Tanaka et al. [27] introduce a new method, Incremental Performance Parameter Estimation, in which the estimation of the theoretical model by polynomial regression is started from the least sampling points and incremented dynamically to improve accuracy. Initially, they apply it on sequential platforms and seek just one algorithmic parameter. Lastovetsky et al. [28] reduce the number of sampling points starting from a previous shape of the curve that represents the execution time. They also introduce the concept "speed band" as the natural way to represent the inherent fluctuations in the speed due to changes in load.

In most scientific and engineering problems, computations are carried out by using basic matrix routines of BLAS type. Therefore, the improvement in the performance of scientific codes is achieved in many cases by the efficient use of those routines. In this context, different automatic optimization techniques emerge as valuable tools that provide scientific software with environment adaptation capacity. Some projects where these techniques have been applied are: Intel MKL [12], IBM ESSL [13], etc. or free implementations (ATLAS [2], Goto BLAS [14], etc.). By way of illustration, a comparison of

Table 9
LU factorization by blocks. Time obtained using the `dgemm` kernel from different basic libraries (MKL, ATLAS, GotoBLAS). Times in seconds.

Size	MKL	ATLAS	GotoBLAS
Saturno			
1000	0.22	0.13	0.02
2000	0.25	0.04	0.08
3000	0.35	0.93	0.20
4000	0.73	1.57	0.38
5000	1.09	2.38	0.66
6000	1.65	3.67	1.07
7000	2.44	4.97	1.63
8000	3.46	6.60	2.76
9000	4.83	8.85	3.26
10000	5.90	10.88	4.39

Table 10
QR factorization by blocks. Time obtained using the `dgemm` kernel from different basic libraries (MKL, ATLAS, GotoBLAS). Times in seconds.

size	MKL	ATLAS	GotoBLAS
Saturno			
1000	0.21	0.30	0.08
2000	0.48	1.24	0.34
3000	1.13	2.87	0.87
4000	2.22	4.86	1.79
5000	3.94	7.66	3.24
6000	6.63	11.38	5.37
7000	10.31	16.25	8.23
8000	15.00	22.42	11.87
9000	21.25	29.95	16.46
10000	28.84	38.86	22.21

Table 11
Cholesky factorization by blocks. Time obtained using the `dgemm` kernel from different basic libraries (MKL, ATLAS, GotoBLAS). Times in seconds.

Size	MKL	ATLAS	GotoBLAS
Saturno			
1000	0.25	0.15	0.01
2000	0.21	0.74	0.06
3000	0.35	1.41	0.19
4000	0.50	2.46	0.48
5000	0.95	3.34	0.95
6000	1.55	4.36	1.65
7000	2.39	5.67	2.59
8000	3.26	7.18	3.84
9000	4.50	9.14	5.36
10000	6.18	11.42	7.32

the execution time in the platform Saturno of LU, QR and Cholesky, using the basic `dgemm` kernel from MKL, ATLAS and Goto BLAS is shown in Tables 9–11. The performance obtained using `dgemm` from ATLAS is clearly inferior to that when MKL or Goto BLAS are used to solve these factorizations in this platform.

However, it was not our aim to compare the performance obtained with these basic libraries, but to study optimization techniques when efficient multithread matrix multiplications are used in higher level routines in NUMA systems. We have used MKL toolkit in this work, but it is equally possible to use other libraries installed in the platform.

8. Conclusions

It is necessary to adjust the running conditions of scientific software to the specific characteristics of NUMA platforms if we want to obtain performance close to the optimum. In these platforms, the memory is physically distributed among the computing nodes. These conditions entail different values for the access times to the data involved in the calculations, depending on the computing core that performs the operation and the localization of the data in the memory. An auto-tuning methodology for linear algebra routines in NUMA systems has been shown, and a LU factorization by blocks with similar scheme that `dgetrf` routine of LAPACK library has been used to illustrate the methodology. This routine has as main basic computation kernel the matrix–matrix multiplication. A comparison of two versions has been carried out: the version with a more standard scheme, that call directly to the kernel `dgemm` from the BLAS implementation of the MKL package, and a version with an improved scheme, that calls to `2L-dgemm`, adapted to NUMA platforms. The kernel `2L-dgemm` is built on the basis of two levels of parallelism, with a first level constituting OpenMP threads and a second level that corresponds to the parallelism intrinsic on the MKL routine `dgemm`. The objetive of this two-level parallelism is to obtain a better correspondence between the threads and the distributed structure of the memory of NUMA platforms.

In the experiments in platforms with a large number of cores, an improvement is obtained with respect to the use of MKL parallelism. So, taking this into consideration, a reduction in the execution time of scientific codes, which use matrix multiplications or linear algebra routines based on them intensively, can be achieved by adequately selecting the threads to be used in the solution of the problem. This selection is performed automatically by the auto-tuning system proposed in this work. The experiments show, for NUMA platforms, the usefulness of two levels of parallelism where the number of threads for each level is selected automatically by the proposed methodology of automatic tuning.

This methodology, which is explained in detail with the matrix multiplication and the LU factorization, can also be extended to different linear algebra routines. Complementary experiments with QR and Cholesky factorizations are shown.

We are currently working on applying an experimental version of our auto-tuning methodology to PLASMA [19], where it is more complicated to design an execution time model of the routines because this framework is based on asynchronous, out of order scheduling of operations.

Acknowledgments

This work was supported by the Spanish MINECO, as well as by European Commission FEDER funds, under grant TIN2012-38341-C04-03. The authors would like to thank Enrique S. Quintana-Ortí from the High Performance Computing & Architectures group at the University Jaume I for granting us access to their Joule computing platform, and acknowledge the computer resources and assistance provided by the Supercomputing Centre of the Scientific Park Foundation of Murcia.

References

[1] S. Akhter, J. Roberts, Multi-Core Programming, Intel Press, 2006.
[2] R.C. Whaley, A. Petitet, J. Dongarra, Automated empirical optimizations of software and the ATLAS project, Parallel Comput. 27 (1–2) (2001) 3–35.
[3] E.J. Im, K. Yelick, R. Vuduc, Sparsity: optimization framework for sparse matrix kernels, Int. J. High Perform. Comput. Appl. 18 (1) (2004) 135–158.

[4] M. Frigo, S.G. Johnson, The design and implementation of FFTW3, Proc. IEEE 93 (2) (2005) 216–231 (Special issue on Program Generation, Optimization, and Platform Adaptation).
[5] T. Katagiri, K. Kise, H. Honda, T. Yuba, ABCLib DRSSED: a parallel eigensolver with an auto-tuning facility, Parallel Comput. 32 (3) (2006) 231–250.
[6] J. Dongarra, J.D. Croz, S. Hammarling, R.J. Hanson, An extended set of fortran basic linear algebra subroutines, ACM Trans. Math. Software 14 (1988) 1–17.
[7] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J.D. Croz, A. Grenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen, LAPACK User's Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1995.
[8] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, W.C. Whaley, ScaLAPACK User's Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.
[9] R.A. van de Geijn, Using PLAPACK, The MIT Press, 1997.
[10] R. Reddy, A.L. Lastovetsky, HeteroMPI + ScaLAPACK: towards a ScaLAPACK (dense linear solvers) on heterogeneous networks of computers, in: HiPC, 2006, pp. 242–253.
[11] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Parallel Comput. 35 (1) (2009) 38–53.
[12] Intel MKL web page, <http://software.intel.com/en-us/intel-mkl/>.
[13] IBM ESSL web page, <http://www-03.ibm.com/systems/software/essl/index.html>.
[14] K. Goto, R.A. van de Geijn, Anatomy of high-performance matrix multiplication, ACM Trans. Math. Software 34 (3) (2008) 1–25.
[15] OpenMP web page, <http://openmp.org/wp/>.
[16] J. Cuenca, L.P. García, D. Giménez, Improving linear algebra computation on NUMA platforms through auto-tuned nested parallelism, in: 20th EUROMICRO Workshop on Parallel, Distributed and Networked Processing, IEEE, 2012, pp. 66–74.
[17] J. Cuenca, D. Giménez, J. González, Architecture of an automatic tuned linear algebra library, Parallel Comput. 30 (2) (2004) 187–220.
[18] J. Cuenca, L.P. García, D. Giménez, J. Dongarra, Processes distribution of homogeneous parallel linear algebra routines on heterogeneous clusters, in: Proceedings of the HeteroPar'05, 2005.
[19] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects, J. Phys.: Conf. Ser. 180 (2009).
[20] J. Cuenca, D. Giménez, J. González, Towards the design of an automatically tuned linear algebra library, in: 10th EUROMICRO Workshop on Parallel, Distributed and Networked Processing, IEEE, 2002, pp. 201–208.
[21] A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, S. Vadhiyar, Numerical libraries and the grid, Int. J. High Perform. Appl. Supercomput. 15 (2001) 359–374.
[22] A. Skjellum, P.V. Bangalore, Driving issues in scalable libraries: polyalgorithms, data distribution independence, redistribution, local storage schemes, in: Seventh SIAM Conf. on Parallel Proc. for Scientific, Computing, 1995, pp. 734–737.
[23] P. Alberti, P. Alonso, A. Vidal, J. Cuenca, L.P. García, D. Giménez, Designing polylibraries to speed up linear algebra computations, Int. J. High Perform. Comput. Network. 1 (1–3) (2004) 75–84.
[24] E. Caron, F. Desprez, F. Suter, Parallel extension of a dynamic performance forecasting tool, Scalable Comput.: Pract. Exp. 6 (1) (2005) 57–69.
[25] R. Vuduc, J.W. Demmel, J. Bilmes, Statistical models for automatic performance tuning, in: Proceedings of the International Conference on Computational Science, ICCS, 2001, pp. 117–126.
[26] T. Katagiri, K. Kise, H. Honda, T. Yuba, FIBER: A generalized framework for auto-tuning software, Springer LNCS 2858 (2003) 146–159.
[27] T. Tanaka, T. Katagiri, T. Yuba, d-Spline based incremental parameter estimation in automatic performance tuning, in: Proceedings of the PARA'06, 2006, pp. 3–13.
[28] A. Lastovetsky, R. Reddy, R. Higgins, Building the functional performance model of a processor, in: Proceedings of the SAC'06, 2006, pp. 23–27.