Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 110 – 119

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Empirical Modelling of Linear Algebra Shared-Memory Routines

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

[a]Departamento de Informática y Sistemas, University of Murcia, Murcia 30071, Spain
[b]Departamento de Ingeniería y Tecnología de Computadores, University of Murcia, Murcia 30071, Spain
[c]Servicio de Apoyo a la Investigación Tecnológica, Polytechnic University of Cartagena, Cartagena 30202, Spain

Abstract

In this work the behavior of the multithreaded implementation of some LAPACK routines on PLASMA and Intel MKL is analyzed. The main goal is to develop a methodology for the installation and modelling of shared-memory linear algebra routines so that some decisions to reduce the execution time can be taken at running time. Typical decisions are: the number of threads to use, the block or tile size in algorithms by blocks or tiles, and the routine to use when there are several algorithms or implementations to solve the problem available. Experiments carried out with PLASMA and Intel MKL show that decisions can be taken automatically and satisfactory execution times are obtained.

Keywords: Performance modelling; Linear algebra; Shared-memory; PLASMA; LAPACK

1. Introduction

Multicore processors and cc-NUMA systems can offer performance improvements, but often demanding new programming methods and algorithms to utilize efficiently the complex architecture involved. In dense linear algebra software, PLASMA [1] and FLAME [2] are examples of libraries that have been designed to achieve high performance on multicore architectures. Software optimization techniques are necessary to obtain low execution times and benefit fully from the potential of the hardware. Decisions are taken at running time as a result of the work performed at installation time, by modelling the execution time of the routines or by applying some empirical approach to study the behavior of the routines. There have been studies on automatically tuning libraries. PHiPAC [3] and ATLAS [4] tune matrix multiplication codes automatically on a large range of CPU platforms. FFTW [5] is a self-tuning library designed to generate high performance code for discrete Fourier transform. OSKI [6] combines install-time evaluations with run-time models to tune sparse-matrix vector multiplication. SPIRAL [7] is a high-performance code generation system for digital signal processing transforms. ABCLib_DRSSED [8] is a parallel eigensolver with an auto-tuning facility. Depending on the type of the computational system used, the decisions to take may differ. For instance: selecting the appropriate number of threads to use at each level of parallelism, how to assign processes to processors or select the correct combination of algorithmic parameters (block size in algorithms by blocks, tile size in algorithms by tiles, etc.).

In [9], auto-tuning is carried out by applying installation techniques to a multithread version of the BLAS-3 matrix multiplication routine (`dgemm`), which constitutes the basic subroutine for many other higher-level linear

*Corresponding author.
E-mail address: luis.garcia@sait.upct.es.

doi:10.1016/j.procs.2013.05.174

algebra packages (LAPACK, ScaLAPACK, PLAPACK, HeteroScaLAPACK, etc.). Since in PLASMA parallelism is not hidden inside Basic Linear Algebra Subprograms (BLAS) [10], in our paper previous ideas for installing multithreaded basic linear algebra routines are extended to higher-level routines. We propose a new empirical modelling method, and the results obtained when applying our auto-tuning methodology to PLASMA are compared with the highly tuned implementations supplied by vendors such as Intel MKL [11].

The rest of the paper is organized as follows. Section 2 presents the auto-tuning methodology for linear algebra routines in shared-memory systems and describes the empirical modelling method proposed. This method obtains a theoretical model of the execution time with experimental estimation of coefficients. The application of empirical modelling to PLASMA routines is described in section 3. In section 4 we evaluate our auto-tuning methodology experimentally in different kinds of shared-memory systems. Finally, in section 5 the conclusions are summarized and some possible extensions of the work are considered.

2. The auto-tuning methodology

In this section we describe our auto-tuning methodology for linear algebra routines. To improve the scalability of shared-memory linear algebra routines, the auto-tuning methodology explained in [9] for the `gemm` routine can be extended to higher-level routines. The goal of this methodology is to find the most appropriate number of threads to use, together with the values of other algorithmic parameters. The methodology is divided in three phases:

- When a new routine is designed, or its code is known, a model of the execution time can be developed, which is used in the subsequent phases; in other cases, the model is empirically estimated. This approach is used here to auto-tune PLASMA routines. Auto-tuning of linear algebra routines in large cc-NUMA based on theoretical models is analyzed in [12] and can be combined with the empirical approach studied here.
- When a model is not available, some experiments are conducted in the installation of the routine, to analyze the behavior of the routine in the system for some significant values (*Installation_Set*). For example, experiments are conducted for different problem sizes, numbers of threads, and block sizes in routines working by blocks. In large NUMA systems, there is a shared RAM memory space but with non uniform data access time, making it difficult to develop an accurate model. In this case, the empirical representation of the behaviour of the routine is not easy, and extensive experimentation may be necessary. The installation process is performed once for a given routine on a given platform. The information generated in the installation is stored for use when the routine is executed. This information can be included in the routine together with a decision engine to obtain an auto-tuning version of the routine.
- When a problem is being solved, the problem size and the maximum number of cores indicated by the user are used to decide the number of threads for the solution of the problem. In routines working by blocks the block size should also be selected. The selection of those parameters is done in the auto-tuning routine prior to the call to the basic routine with the values selected for the parameters. The different possible values for the algorithm parameters are substituted in the empirically estimated model, and those values which provide the lowest theoretical execution time are used in the solution of the problem.

So, an empirically estimated model of the execution time can be used to determine the most appropriate values for the algorithm parameters (number of threads, block sizes, etc.) as well as the routine to use if several are available for the problem in question. The sequential execution time of all the routines considered has a cost of order $O(n^3)$, with n being the size of the matrix, and so in the theoretical model there will be terms in n^3, n^2 and n. For the parallel version where the number of threads (t) is the only algorithm parameter, the highest cost (n^3) should be divided by t, and other terms should be multiplied by t. Thus, we could consider all possible combinations $\{n^3, n^2, n, 1\} \times \{t, 1, \frac{1}{t}\}$, but for n^3 we consider only $\frac{n^3}{t}$, and the lowest order terms ($\frac{n}{t}$, t, 1 and $\frac{1}{t}$) are not included in the model. The execution time is modelled by:

$$T(n, t) = k_1 \frac{n^3}{t} + k_2 n^2 t + k_3 n^2 + k_4 \frac{n^2}{t} + k_5 nt + k_6 n \tag{1}$$