Proceedings of the Federated Conference on Computer Science and Information Systems pp. 489–492
DOI: 10.15439/2017F231
ISSN 2300-5963 ACSIS, Vol. 11

OpenMP Thread Affinity for Matrix Factorization on Multicore Systems

Beata Bylina and Jarosław Bylina
Marie Curie-Skłodowska University,
Institute of Mathematics,
Pl. M. Curie-Skłodowskiej 5,
20-031 Lublin, Poland
Email: {beata.bylina, jaroslaw.bylina}@umcs.pl

***Abstract*—The aim of this paper is to investigate the impact of thread affinity on computing performance for matrix factorization on shared memory multicore systems with hierarchical memory. We consider two parallel block matrix factorizations (LU and WZ) and employ thread affinity to improve their performance. We study decomposition without pivoting and we compare differences between various affinity strategies for diagonally dominant matrices. Our results show that the choice of thread affinity has the measurable impact on the performance of the matrice factorizations.**

I. Introduction

THE ADVANCE of the shared memory multicore and manycore architectures caused a rapid development of one type of the parallelism, namely the *thread level parallelism*. This kind of parallelism relies on splitting a program into subprograms which can be executed concurrently. Each of such subprograms is performed by one or more software threads.

The *thread affinity* is a set of policies that determine how software threads are pinned to processing units [1]. The goal of the thread affinity is to bind software threads to the hardware threads in such a way that memory accesses to data shared between software threads are optimized and all the cores are equally loaded.

Determining the efficiency of the thread mapping depends on the machine and the application. There is not a single thread mapping strategy that suits all the applications. In this work, we are going to try to state rules which guide us to determine efficient thread affinity to improve the performance of matrix factorization on shared memory multithreaded machines.

Efficient parallel matrix factorizations and their implementations on different contemporary parallel machines are crucial for engineering applications and computational science. In this work, we study the LU factorization, and another form of the factorization, namely the WZ [3], [4] factorization. We assume that the factorized diagonally dominant matrix is dense, nonsingular, square. For both factorizations (LU and WZ), we consider block versions which use a standard set of Basic Linear Algebra Subprograms (BLAS) [2].

The rest of this paper is organized as follows. Section II describes the methodology of the numerical experiments. Section III shows the results of numerical experiments carried out on shared memory multicore architectures and evaluates different thread affinities for the matrix decomposition. Section IV concludes our research.

II. Environment

We tested the execution time of two matrix decompositions, namely the LU factorization and the WZ factorization. We compared three implementations of these matrix decompositions, namely:

- a multithreaded implementation of the `dgetrfnpi` routine from the MKL library, which computes the complete LU factorization of a general matrix without pivoting. In our case, the matrices are square which size is $n \times n$. In the implementation of the `dgetrfnpi` routine the panel factorization (factorization of a block of columns) is used, as well as the level 3 BLAS routines (`DTRSM` and `DGEMM`). We denoted this LU factorization implementation by LU.
- a parallel block WZ factorization with the use of multithreaded level 3 BLAS routines (`DTRSM` and `DGEMM`), where the matrix is partitioned into $r \times r$ tiles (denoted by TWZ(r));
- a parallel block WZ factorization with the use of level 3 BLAS routines (`DTRSM` and `DGEMM`) and the OpenMP standard (denoted by TWZ(r)-OpenMP). OpenMP is used to parallelize for loops with the `dynamic` scheduler.

Experiments were carried out on Intel Xeon E5-2670 v.3 (Haswell) with two 12-cores (24 cores). All applications were compiled with `icc` using the following options: `-xHost`, `-mkl`, `-openmp`, `-O3`. Here, the `-xHost` option generates instructions for the highest instruction set and processor available on the compilation host machine. The `-mkl` and `-openmp` options link the program against two libraries (MKL and OpenMP). The last one, `-O3`, orders the compiler to optimize the code automatically with the use of vectorization and parallelization (among others).

All floating point calculations were performed in the double precision. The input matrices were generated by the author. They were random matrices, with a dominant diagonal of an even size of $1024 \times \{1, \ldots, 9\}$. Various numbers of tiles were tested, namely, each matrix was divided into $r = 8, 16, 32, 64$ tiles for each side (for the rows and the columns). The

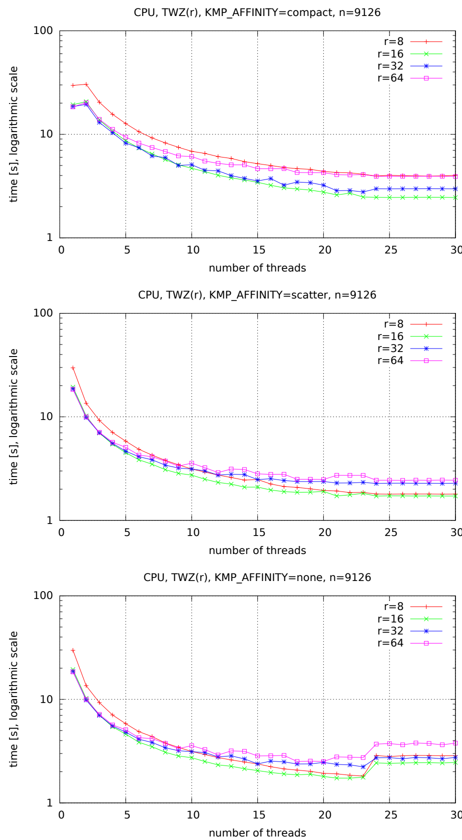

Fig. 1. Run-time in seconds of the parallel block WZ factorization algorithm for different $r = 8, 16, 32, 64$ for a matrix of size 9216 for various numbers of threads and three values of the KMP_AFFINITY environment variable — TWZ(r) implementation.

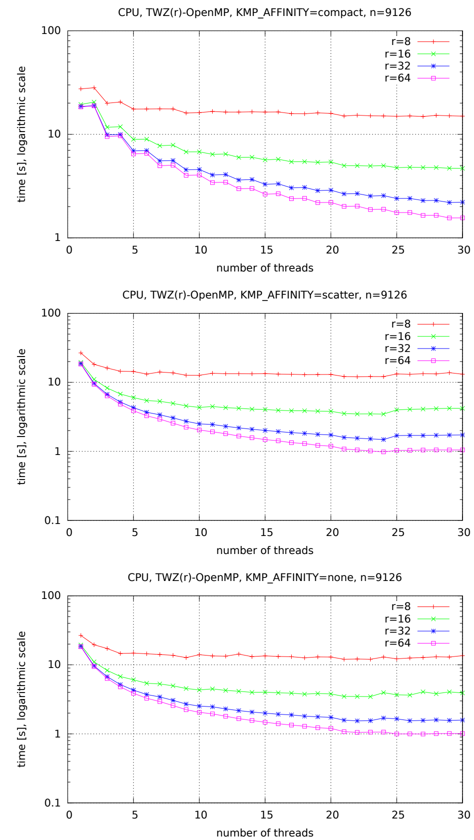

Fig. 2. Run-time in seconds of the parallel block WZ factorization algorithm for different $r = 8, 16, 32, 64$ for a matrix of size 9216 for various numbers of threads and three values of the KMP_AFFINITY environment variable — TWZ(r)-OpenMP implementation.

performance times were measured with the use of a standard function, namely `dsecnd()` from MKL library. Another environment variable used in the tests is KMP_AFFINITY, which is set to one of the three values: `compact`, `scatter`, `none`. To better control assigning software threads to hardware threads, we chose the granularity as `thread`.

We studied a connection between the KMP_AFFINITY environment variable and the following parameters: the number of the threads — from 1 to 30, the size of the matrix: $1024 \times \{1, \ldots, 9\}$, the number of the tiles for each side (for the rows and the columns): $r = 8, 16, 32, 64$.

III. Results

Figures 1 and 2 present the time (in seconds) of the block WZ factorization for 4 different numbers of tiles ($r = 8, 16, 32, 64$), for different number of threads (1 – 30 threads), for a matrix of the size 9216, for three values of the KMP_AFFINITY environment variable — for the TWZ(r) and TWZ(r)-OpenMP implementations (respectively).

Figures 3 and 4 present the time (in seconds) of the block WZ factorization for various number of tiles ($r = 8, 16, 32, 64$) for 24 threads and different sizes of matrices for all three considered values of the KMP_AFFINITY environment variable for the TWZ(r) and TWZ(r)-OpenMP implementations (respectively).

At least one important conclusion can be seen from these results. Namely, the number of tiles influences the time of the computation. For $r = 64$ the TWZ(r) implementation is the slowest but TWZ(r)-OpenMP is the fastest. The TWZ(r) implementation is the fastest for $r = 16$. This conclusion holds true independent of the value of the KMP_AFFINITY variable, the number of threads or the matrix size.

Secondly, we investigate the effect of the thread affinity on the execution time for all three implementations. Figure 5 compares all three values of the KMP_AFFINITY environment variable and it shows the time (in seconds) for different numbers of threads (1–30 threads) for a matrix of the size 9216 — for TWZ(16) (top left), TWZ(64)-OpenMP (top right) and the LU factorization (at the bottom).

Figure 6 compares all three values of the KMP_AFFINITY environment variable and it shows the time (in seconds) for different matrix sizes for 24 threads — for TWZ(16) (top left), TWZ(64)-OpenMP (top right) and the LU factorization (at the bottom). Thus, it implies that (for the considered applications) the best value of the KMP_AFFINITY environment variable is `scatter` which is further investigated.

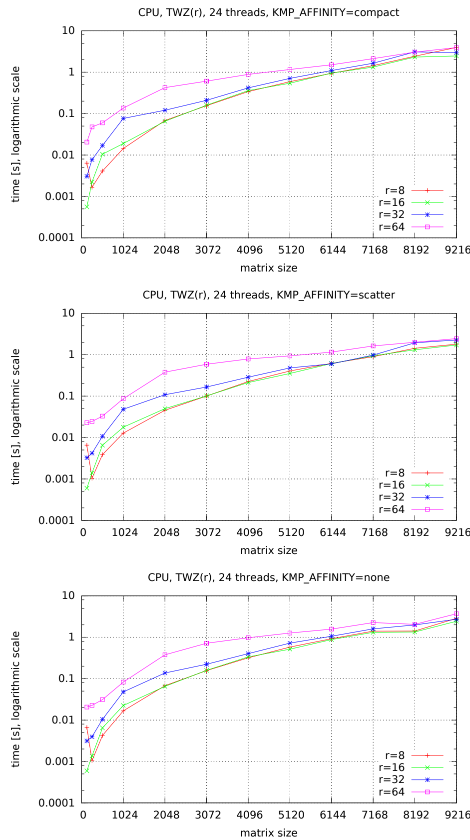

Fig. 3. Run-time in seconds of the parallel block WZ factorization algorithm for different $r = 8, 16, 32, 64$ for various sizes of matrices for 24 threads and three values of the KMP_AFFINITY environment variable — TWZ(r) implementation.

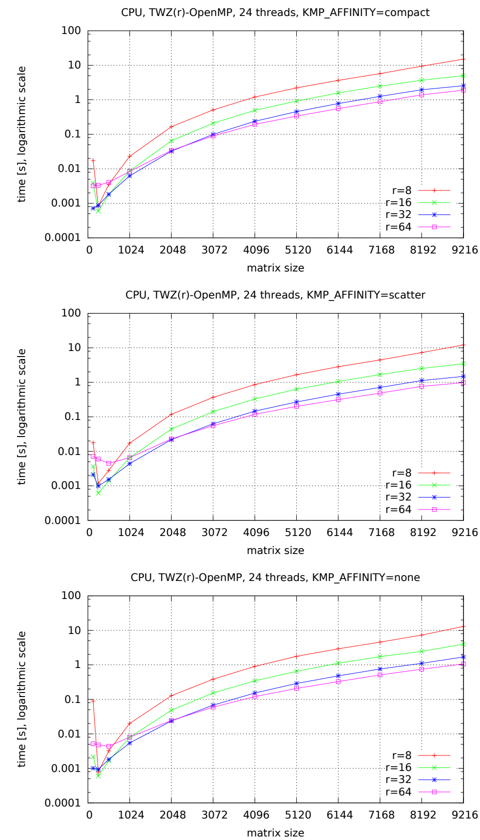

Fig. 4. Run-time in seconds of the parallel block WZ factorization algorithm for different $r = 8, 16, 32, 64$ for various sizes of matrices for 24 threads and three values of the KMP_AFFINITY environment variable — TWZ(r)-OpenMP implementation.

Finally, we analyzed the execution times of the three implementations (namely LU, TWZ(16) and TWZ(64)-OpenMP) for the KMP_AFFINITY environment variable set to scatter. Figure 7 compares the performance times (in seconds) of the TWZ(16), TWZ(64)-OpenMP and LU implementations for scatter and for different numbers of threads and matrix sizes.

The shortest execution time is obtained for LU. Our TWZ(64)-OpenMP implementation is a little worse. The slowest implementation is TWZ(16). To better investigate the performance time for LU and our best implementation, they were tested for larger matrices and various values of the KMP_AFFINITY environment variable. Table I presents the times (in seconds) of the LU and TWZ(64)-OpenMP implementations.

For none our implementation is faster than LU; however, for scatter, LU wins. This implies that LU is more sensitive for the KMP_AFFINITY setting.

The tests show the following facts. **The number of the threads.** To achieve the shortest execution time, it is the best to use all the physical cores (here, 24 threads), although, without hyper-threading. **The matrix size.** For small matrices, our implementations are better. It is caused by the fact that MKL does not create threads for small problems. However, for the size of 9216, the shortest time is achieved by LU. All three implementations scale well in regard to the size of the matrix. **The number of the tiles**. The block size used by the MKL implementation of dgetrf is internally hidden in the library and unknown to us at the time of this writing. The r parameter impacts the execution time of the block WZ factorization for both implementations. **Thread Affinity.** The thread affinity had an important impact on the performance. All three implementations (both the LU factorization implementation provided by a vendor as well as the WZ factorization implemented by the author) work fastest

TABLE I
THE TIMES (IN SECONDS) OF THE LU AND TWZ(64)-OPENMP IMPLEMENTATIONS — FOR VARIOUS VALUES OF KMP_AFFINITY

matrix size	implementation	compact	scatter	none
12 288	TWZ(64)-OpenMP	4.06	**2.22**	2.39
	LU	3.66	**1.80**	3.50
13 312	TWZ(64)-OpenMP	5.16	2.70	**2.24**
	LU	4.53	**2.23**	4.70
14 336	TWZ(64)-OpenMP	6.59	**3.39**	3.50
	LU	5.69	**2.77**	4.30
15 360	TWZ(64)-OpenMP	8.18	**4.08**	4,24
	LU	6.96	**3.42**	6.40

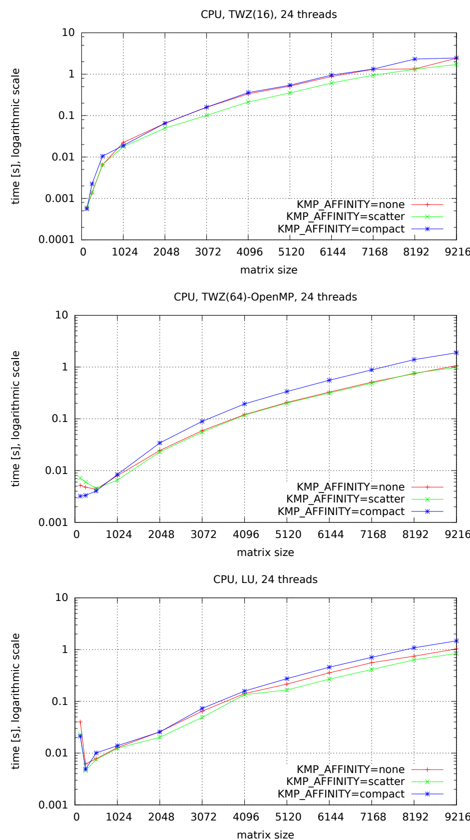

Fig. 6. Run-time in seconds of the parallel block WZ factorization algorithm (top left: TWZ(16); top right: TWZ(64)-OpenMP) and the LU factorization (at the bottom) for different matrix sizes for 24 threads for three values of the `KMP_AFFINITY` environment variable.

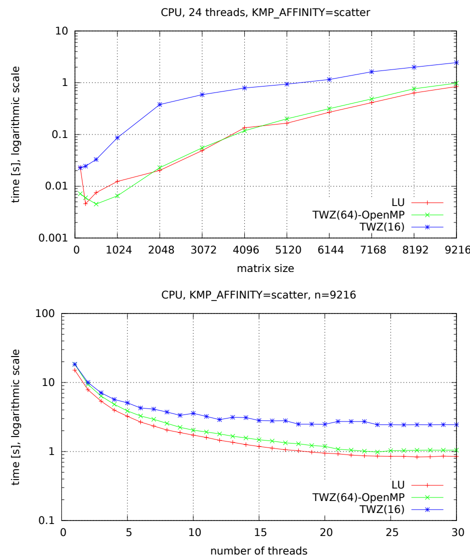

Fig. 7. Run-time in seconds of TWZ(16), TWZ(64)-OpenMP and LU for `KMP_AFFINITY=scatter` — for various numbers of threads and matrix sizes

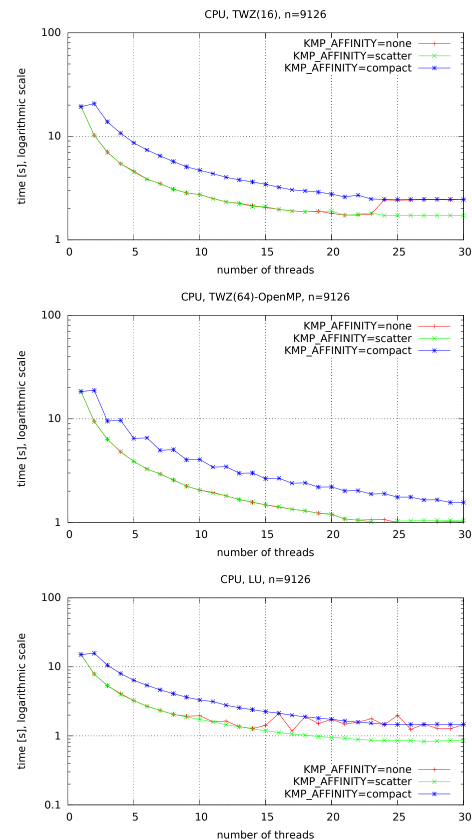

Fig. 5. Run-time in seconds of the parallel block WZ factorization algorithm (top left: TWZ(16); top right: TWZ(64)-OpenMP) and the LU factorization (at the bottom) for a matrix of the size 9216 for different numbers of threads for three values of the `KMP_AFFINITY` environment variable.

for `scatter`, and slowest for `compact` — and it does not depend on the number of threads, the matrix size or the value of r.

IV. Conclusion

The paper highlights the significant impact of thread affinity on the performance of the matrix factorizations which use BLAS operations in their implementations. For the matrix factorization, the `KMP_AFFINITY` environment variable should be set to `scatter` because this way we efficiently exploit the potential of modern shared memory multicore machines. With this setting, threads are put far from each other (as on different packages) what improves the total memory throughput and the usage of the caches.

References

[1] Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux, and Israel Koren. Affinity-based thread and data mapping in shared memory systems. *ACM Comput. Surv.*, 49(4):64:1–64:38, December 2016.

[2] J. Dongarra, J. DuCroz, I. S. Duff, and S. Hammarling. A set of level-3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–28, 1990.

[3] D.J. Evans and M. Hatzopoulos. A parallel linear system solver. *International Journal of Computer Mathematics*, 7(3):227–238, 1979.

[4] P. Yalamov and D.J. Evans. The WZ matrix factorisation method. *Parallel Computing*, 21(7):1111–1120, 1995.