Parallel Tiled QR Factorization for Multicore Architectures

Alfredo Buttari[1], Julien Langou[2], Jakub Kurzak[1], and Jack Dongarra[1,3,4]

[1] Computer Science Dept. University of Tennessee Knoxville, USA
[2] Department of Mathematical Sciences, University of Colorado at Denver and Health Sciences Center, Colorado USA
[3] Oak Ridge National Laboratory, Oak Ridge, Tennessee USA
[4] University of Manchester, UK

Abstract. As multicore systems continue to gain ground in the High Performance Computing world, linear algebra algorithms have to be reformulated or new algorithms have to be developed in order to take advantage of the architectural features on these new processors. Fine grain parallelism becomes a major requirement and introduces the necessity of loose synchronization in the parallel execution of an operation. This paper presents an algorithm for the QR factorization where the operations can be represented as a sequence of small tasks that operate on square blocks of data. These tasks can be dynamically scheduled for execution based on the dependencies among them and on the availability of computational resources. Compared to the standard approach, say with LAPACK, may result in an out of order execution of the tasks which will completely hide the presence of intrinsically sequential tasks in the factorization. Performance comparisons are presented with the LAPACK algorithm for QR factorization where parallelism can only be exploited at the level of the BLAS operations.

1 Introduction

In the last twenty years, microprocessor manufacturers have been driven towards higher performance rates only by the exploitation of higher degrees of *Instruction Level Parallelism* (ILP). Based on this approach, several generations of processors have been built where clock frequencies were higher and higher and pipelines were deeper and deeper. As a result, applications could benefit from these innovations and achieve higher performance simply by relying on compilers that could efficiently exploit ILP. Due to a number of physical limitations (mostly power consumption and heat dissipation) this approach cannot be pushed any further. For this reason, chip designers have moved their focus from ILP to *Thread Level Parallelism* (TLP) where higher performance can be achieved by replicating execution units (or *cores*) on the die while keeping the clock rates in a range where power consumption and heat dissipation do not represent a problem. It is easy to imagine that multicore technologies will have a

R. Wyrzykowski et al. (Eds.): PPAM 2007, LNCS 4967, pp. 639–648, 2008.

deep impact on the High Performance Computing (HPC) world since supercomputers have very high number of processors and, thus, multicore technologies can help reducing the power consumption.

As a consequence, all the applications that were not explicitly coded to be run on parallel architectures must be rewritten with parallelism in mind. Also, those applications that could exploit parallelism may need considerable rework in order to take advantage of the fine-grain parallelism features provided by multicores.

The current set of multicore chips from Intel and AMD are for the most part multiple processors glued together on the same chip. There are many scalability issues to this approach and it is unlikely that type of architecture will scale up beyond 8 or 16 cores. Even though it is not yet clear how chip designers are going to address these issues, it is possible to identify some properties that algorithms must have in order to match high degrees of TLP:

fine granularity: cores are (and probably will be) associated with relatively small local memories (either caches or explicitly managed memories like in the case of the STI Cell [1] architecture or the Intel Polaris[2] prototype). This requires splitting an operation into tasks that operate on small portions of data in order to reduce bus traffic and improve data locality.

asynchronicity: as the degree of TLP grows and granularity of the operations becomes smaller, the presence of synchronization points in a parallel execution seriously affects the efficiency of an algorithm.

The LAPACK [3] and ScaLAPACK [4] software libraries represent a *de facto* standard for high performance dense Linear Algebra computations and have been developed, respectively, for shared-memory and distributed-memory architectures.

Substantially, both LAPACK and ScaLAPACK implement sequential algorithms that rely on parallel building blocks, i.e. parallel BLAS operations. As multicore systems require finer granularity and higher asynchronicity, considerable advantages may be obtained by reformulating old algorithms or developing new algorithms in a way that their implementation can be easily mapped on these new architectures.

A number of approaches along these lines have been proposed in [5,6,7,8,9]; block partitioning and hybrid data structures have been studied and significant gains can be obtained on more conventional processors either in shared or distributed memory environments. The more recent work has focused on looking at operations of the standard LAPACK algorithms for some common factorizations are broken into sequences of smaller tasks in order to achieve finer granularity and higher flexibility in the scheduling of tasks to cores. The importance of fine granularity algorithms is also shown in [10].

The rest of this document shows how this can be achieved for the QR factorization. Section 2 describes the tiled QR factorization that provides both fine granularity and high level of asynchronicity; performance results for this algorithm are shown in Section 3.

2 Tiled QR Factorization

The kernels (e.g. BLAS operations) on which common Linear Algebra are based can be broken into smaller tasks to achieve finer granularity. These tasks can thus be scheduled according to a dynamic, graph driven approach that leads to an out-of-order, asynchronous execution. These ideas are not new and have been proposed a number of times in the past [11,12,13,14]. More recent work in this direction show how this idea can be applied to common Linear Algebra operations as SYRK (symmetric rank-K update), Cholesky, block LU, and block QR factorizations [5,7,6] for multicore. In the case of SYRK and CHOL, no algorithmic change is needed, since both these operations can be naturally "tiled" to achieve a very fine granularity. A good summary of recursive and blocked approaches can be found in [8,15] (Sections 5 and 6). Kurzak et al. [5] recently showed that the application of this approach to the standard LAPACK algorithms for LU and QR is limited by the granularity that can be obtained by simply "tiling" the elementary operations that these two factorizations are based on. In order to achieve finer granularity in the LU and QR factorizations, a major algorithmic change is needed.

The algorithmic change we propose is actually well-known and takes its roots in updating factorizations [16,17]. Using updating techniques to tile the algorithms have first[1] been proposed by Yip [18] for LU to improve the efficiency of out-of-core solvers, and were recently reintroduced in [19,20] for LU and QR, once more in the out-of-core context. A similar idea has also been proposed in [21] for Hessenberg reduction in the parallel distributed context.

All of these approaches use the idea of manipulating and operating on the coefficient matrix by referencing small blocks of the matrix [22]. The block organization is a convenient way of expressing and moving parts of the matrix through the memory hierarchy.

The originality of this paper is to study the effectiveness of these algorithms in the context of multicore architectures where they can be used to achieve a fine granularity, high degree of parallelism and asynchronous execution.

2.1 A Fine-Grain Algorithm for QR Factorization

The tiled QR factorization will be constructed based on the following four elementary operations:

`DGEQT2.` This subroutine was developed to perform the unblocked factorization of a diagonal block A_{kk} of size $b \times b$. This operation produces an upper triangular matrix R_{kk}, a unit lower triangular matrix V_{kk} that contains b Householder reflectors and an upper triangular matrix T_{kk} as defined by the "WY" technique for accumulating the transformations (see [23], [24] and [25] for details). Note that both R_{kk} and V_{kk} can be written on the memory area that was used for A_{kk} and, thus, no extra storage is needed for them. A temporary work space is needed to store T_{kk}.

[1] To our knowledge.

$$H_1 H_2 \ldots H_b = I - VTV^T$$

where V is an n-by-b matrix whose columns are the individual vectors $v_1, v_2, \ldots, v_b$ associated with the Householder matrices $H_1, H_2, \ldots, H_b$, and T is an upper triangular matrix of order b.
Thus, `DGEQT2`(A_{kk}, T_{kk}) performs

$$A_{kk} \longleftarrow V_{kk}, R_{kk} \qquad T_{kk} \longleftarrow T_{kk}$$

`DLARFB`. This LAPACK subroutine will be used to apply the transformation (V_{kk}, T_{kk}) computed by subroutine `DGEQT2` to a block A_{kj}.
Thus, `DLARFB`(A_{kj}, V_{kk}, T_{kk}) performs

$$A_{kj} \longleftarrow (I - V_{kk} T_{kk} V_{kk}^T) A_{kj}$$

`DTSQT2`. This subroutine was developed to perform the unblocked QR factorization of a matrix that is formed by coupling an upper triangular block R_{kk} with a square block A_{ik}. This subroutine will return an upper triangular matrix $\tilde{R}_{kk}$ which will overwrite R_{kk} and b Householder reflectors where b is the block size. Note that, since R_{kk} is upper triangular, the resulting Householder reflectors can be represented as an identity block I on top of a square block V_{ik}. For this reason no extra storage is needed for the Householder vectors since the identity block need not be stored and V_{ik} can overwrite A_{ik}. Also a matrix T_{ik} is produced for which storage space has to be allocated. See Figure 1 for a graphical representation.
Then, `DTSQT2`(R_{kk}, A_{ik}, T_{ik}) performs

$$\begin{pmatrix} R_{kk} \\ A_{ik} \end{pmatrix} \longleftarrow \begin{pmatrix} I \\ V_{ik} \end{pmatrix}, \tilde{R}_{kk} \qquad T_{ik} \longleftarrow T_{ik}$$

`DSSRFB`. This subroutine was developed to apply the transformation computed by `DTSQT2` to a matrix formed coupling two square blocks A_{kj} and A_{ij}.
Thus, `DSSRFB`(A_{kj}, A_{ij}, V_{ik}, T_{ik}) performs

$$\begin{pmatrix} A_{kj} \\ A_{ij} \end{pmatrix} \longleftarrow \left(I - \begin{pmatrix} I \\ V_{ik} \end{pmatrix} \cdot (T_{ik}) \cdot \begin{pmatrix} I & V_{ik}^T \end{pmatrix} \right) \begin{pmatrix} A_{kj} \\ A_{ij} \end{pmatrix}$$

All of this elementary operations rely on BLAS subroutines to perform internal computations.

Assuming a matrix A of size $pb \times qb$ where b is the block size and each A_{ij} is of size $b \times b$, the QR factorization can be performed as in Algorithm 1.

The operations count for Algorithm 1 is 25% higher than the one of the LAPACK algorithm for QR factorization; specifically the tiled algorithm requires $5/2n^2(m-n/3)$ floating point operations compared to the $4/2n^2(m-n/3)$ for the LAPACK algorithm. Details of the operation count of the parallel tiled algorithm are reported in [26]. Performance results in Section 3 will demonstrate that it is worth paying this cost for the sake of scaling.

Algorithm 1. The block algorithm for QR factorization.

```
1: for k = 1, 2..., min(p, q) do
2:   DGEQT2(A_kk, T_kk);
3:   for j = k + 1, k + 2, ..., q do
4:     DLARFB(A_kj, V_kk, T_kk);
5:   end for
6:   for i = k + 1, k + 1, ..., p do
7:     DTSQT2(R_kk, A_ik, T_ik);
8:     for j = k + 1, k + 2, ..., q do
9:       DSSRFB(A_kj, A_ij, V_ik, T_ik);
10:    end for
11:  end for
12: end for
```

Figure 1 gives a graphical representation of one repetition (with $k = 1$) of the outer loop in Algorithm 1 with $p = q = 3$. The red, thick borders show what blocks in the matrix are being read and the light blue fill shows what blocks are being written in a step. The T_{kk} matrices are not shown in this figure for clarity purposes.

2.2 Graph Driven Asynchronous Execution

Following the approach presented in [5,6], Algorithm 1 can be represented as a Directed Acyclic Graph (DAG) where nodes are elementary tasks that operate on $b \times b$ blocks and where edges represent the dependencies among them. Figure 2 show the DAG when Algorithm 1 is executed on a matrix with $p = q = 3$. It can be noted that the DAG has a recursive structure and, thus, if $p_1 \geq p_2$ and $q_1 \geq q_2$ then the DAG for a matrix of size $p_2 \times q_2$ is a subgraph of the DAG for a matrix of size $p_1 \times q_1$. This property also holds for most of the algorithms in LAPACK.

Once the DAG is known, the tasks can be scheduled asynchronously and independently as long as the dependencies are not violated. A critical path can be identified in the DAG as the path that connects all the nodes that have the higher number of outgoing edges. Based on this observation, a scheduling policy can be used, where higher priority is assigned to those nodes that lie on the critical path. Clearly, in the case of our block algorithm for QR factorization, the nodes associated to the `DGEQT2` subroutine have the highest priority and then three other priority levels can be defined for `DTSQT2`, `DLARFB` and `DSSRFB` in descending order.

This dynamic scheduling results in an out of order execution where idle time is almost completely eliminated since only very loose synchronization is required between the threads. The graph driven execution also provides some degree of adaptivity since tasks are scheduled to threads depending on the availability of execution units.

2.3 Block Data Layout

The major limitation of performing very fine grain computations, is that the BLAS library generally have very poor performance on small blocks. This

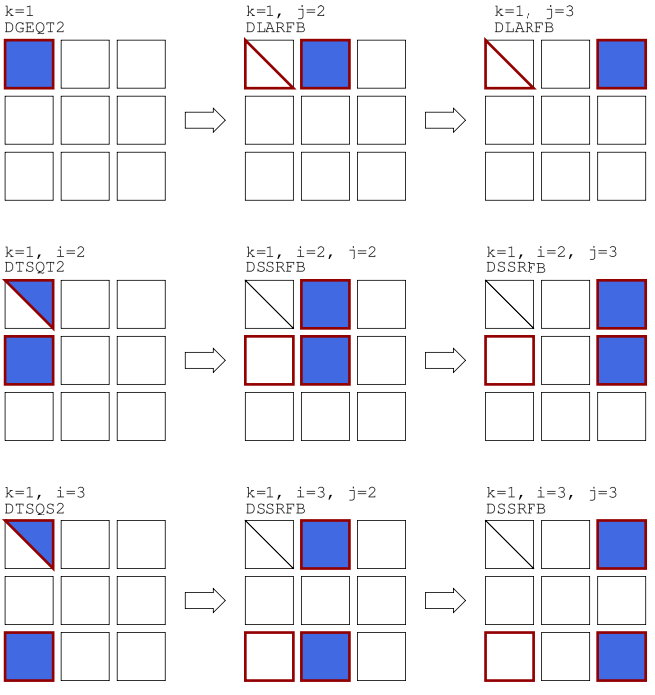

Fig. 1. Graphical representation of one repetition of the outer loop in Algorithm 1 on a matrix with $p = q = 3$. As expected the picture is very similar to the out-of-core algorithm presented in [20].

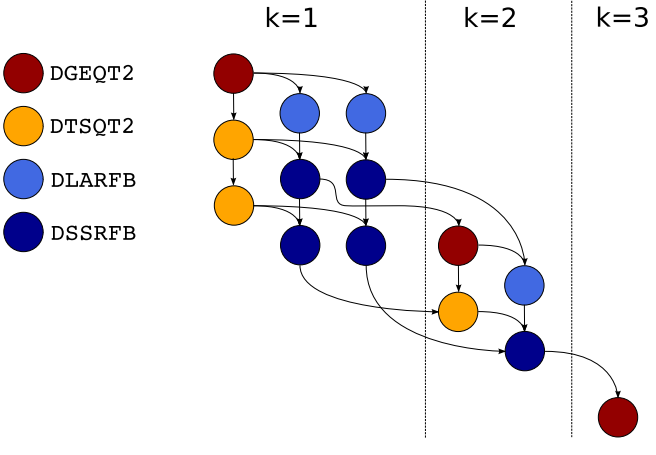

Fig. 2. The dependency graph of Algorithm 1 on a matrix with $p = q = 3$

situation can be considerably improved by storing matrices in Block Data Layout (BDL) instead of the Column Major Format that is the standard storage format for FORTRAN arrays.

In BDL a matrix is split into blocks and each block is stored into contiguous memory locations. Each block is stored in Column Major Format and blocks are stored in Column Major Format with respect to each other. As a result the access pattern to memory is more regular and BLAS performance is considerably improved. The benefits of BDL have been extensively studied in the past, for example in [22], and recent studies like [7] demonstrate how fine-granularity parallel algorithms can benefit from BDL. It is important to note that both [22] and [7] focus on algorithms that are based on the same approach presented here.

3 Performance Results

The performance of the tiled QR factorization with dynamic scheduling of tasks has been measured on the systems listed in Table 1 and compared to the performance of the fork-join approach, i.e., the standard algorithm for block QR factorization of LAPACK associated with multithreaded BLAS.

Table 1. Details of the systems used for the following performance results

	8-way dual Opteron	2-way quad Clovertown
Architecture	Dual-Core AMD Opteron^{TM}8214	Intel®Xeon®CPU X5355
Clock speed	2.2 GHz	2.66 GHz
# cores	$8 \times 2 = 16$	$2 \times 4 = 8$
Peak performance	70.4 Gflop/s	85.12 Gflop/s
Memory	62 GB	16 GB
Compiler suite	Intel 9.1	Intel 9.1
BLAS libraries	MKL-9.1	MKL-9.1

Figures 3, 4 report the performance of the QR factorization for both the block algorithm with dynamic scheduling and the LAPACK algorithm with multithreaded BLAS. A block size of 200 has been used for the block algorithm while the block size for the LAPACK algorithm[2] has been tuned in order to achieve the best performance for all the combinations of architecture and BLAS library.

In each graph, two curves are reported for the block algorithm with dynamic scheduling; the solid curve shows its relative performance when the operation count is assumed equal to the one of the LAPACK algorithm (i.e. $4/2n^2(m-n/3)$) while the dashed curve shows its "raw" performance, i.e. the actual flop rate computed with the exact operation count for this algorithm that is $5/2n^2(m-n/3)$. As already mentioned, the "raw performance" (dashed curve) is 25% higher than the relative performance (solid curve).

[2] The block size in the LAPACK algorithm sets the width of the so called panel factorization which determines the ratio between Level-2 and Level-3 BLAS operations.

The graphs on the left part of each figure show the performance measured using the maximum number of cores available on each system with respect to the problem size. The graphs on the right part of each figure show the weak scalability, i.e. the flop rates versus the number of cores when the local problem size is kept constant (nloc=5,000) as the number of cores increases. Figures 3, 4 show that, despite the higher operation count, the block algorithm with dynamic scheduling is capable of completing the QR factorization in less time than the LAPACK algorithm when the parallelism degree is high enough that the benefits of the asynchronous execution overcome the penalty of the extra flops. For lower numbers of cores, in fact, the fork-join approach has a good scalability and

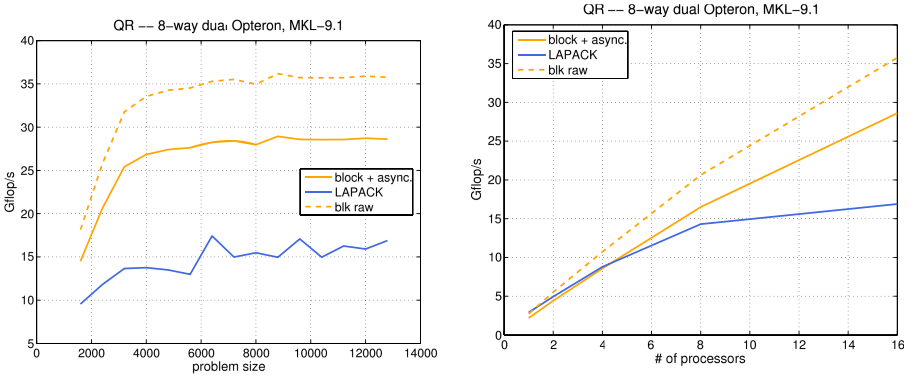

Fig. 3. Comparison between the performance of the block algorithm with dynamic scheduling using MKL-9.1 on an 8-way dual Opteron system. The dashed curve reports the raw performance of the block algorithm with dynamic scheduling, i.e., the performance as computed with the true operation count $5/2n^2(m - n/3)$.

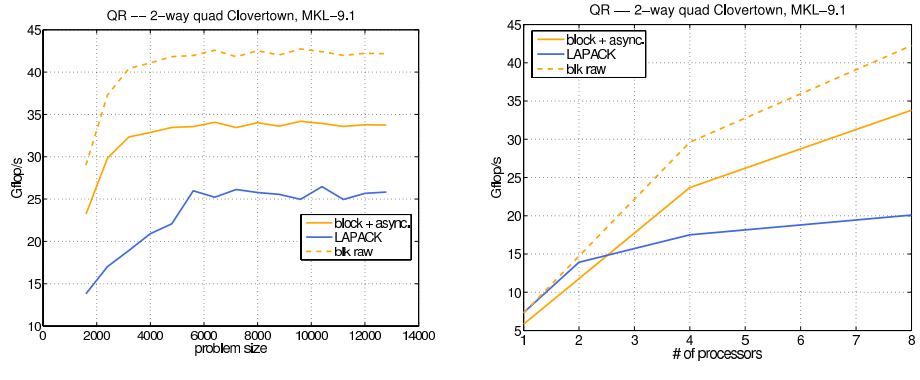

Fig. 4. Comparison between the performance of the block algorithm with dynamic scheduling using MKL-9.1 on an 2-way quad Clovertown system. The dashed curve reports the raw performance of the block algorithm with dynamic scheduling, i.e., the performance as computed with the true operation count $5/2n^2(m - n/3)$.

completes the QR factorization in less time than the block algorithm because of the lower flop count. Note that the actual execution rate of the block algorithm for QR factorization with dynamic scheduling (i.e., the dashed curves) is always higher than that of the LAPACK algorithm with multithreaded BLAS even for low numbers of cores. The actual performance of the block algorithm, even if considerably higher than that of the fork-join one, is still far from the peak performance of the systems used for the measures. This is mostly due to two factors: the nature of the BLAS operations involved and the performance of BLAS routines on small size blocks.

4 Conclusions

By adapting known algorithms for updating the QR factorization of a matrix, we have derived a fine-granularity implementation scheme of the QR factorization for multicore architectures based on dynamic scheduling and block data layout. Although the proposed algorithm is performing 25% more FLOPS than the regular algorithm, the gain in flexibility allows an efficient dynamic scheduling which enables the algorithm to scale almost perfectly when the number of cores increases.

References

1. Pham, D., Asano, S., Bolliger, M., Day, M.N., Hofstee, H.P., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T., Yazawa, K.: The design and implementation of a first-generation CELL processor. In: IEEE International Solid-State Circuits Conference, pp. 184–185 (2005)
2. Teraflops research chip, `http://www.intel.com/research/platform/terascale/teraflops.htm`
3. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK User's Guide, 3rd edn. SIAM, Philadelphia (1999)
4. Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. Computer Physics Communications 97, 1–15 (1996), (also as LAPACK Working Note #95)
5. Kurzak, J., Dongarra, J.: Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. LAPACK Working Note 178 (September 2006), Also available as UT-CS-06-581
6. Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Luszczek, P., Tomov, S.: The impact of multicore on math software. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 1–10. Springer, Heidelberg (2007)
7. Chan, E., Quintana-Orti, E.S., Quintana-Orti, G., van de Geijn, R.: Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In: SPAA 2007: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, pp. 116–125. ACM Press, New York (2007)

8. Elmroth, E., Gustavson, F., Jonsson, I., Kågström, B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Review 46(1), 3–45 (2004)
9. Gustavson, F., Karlsson, L., Kågström, B.: Three algorithms for cholesky factorization on distributed memory using packed storage. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 550–559. Springer, Heidelberg (2007)
10. Kurzak, J., Buttari, A., Dongarra, J.: Solving systems of linear equations on the CELL processor using Cholesky factorization. Technical Report UT-CS-07-596, Innovative Computing Laboratory, University of Tennessee Knoxville (April 2007)
11. Lord, R.E., Kowalik, J.S., Kumar, S.P.: Solving linear algebraic equations on an mimd computer. J. ACM 30(1), 103–117 (1983)
12. Dongarra, J.J., Hiromoto, R.E.: A collection of parallel linear equations routines for the Denelcor HEP 1(2), 133–142 (December 1984)
13. Agarwal, R.C., Gustavson, F.G.: Vector and parallel algorithms for cholesky factorization on ibm 3090. In: Supercomputing 1989: Proceedings of the 1989 ACM/IEEE conference on Supercomputing, pp. 225–233. ACM Press, New York (1989)
14. Agarwal, R.C., Gustavson, F.G.: A parallel implementation of matrix multiplication and LU factorization on the IBM 3090. In: Proceedings of the IFIP WG 2.5 Working Group on Aspects of Computation on Asychronous Parallel Processors, Stanford CA, Augest 22-26,1988, North Holland, Amsterdam (1988)
15. Elmroth, E., Gustavson, F.G.: Applying recursion to serial and parallel QR factorization leads to better performance. IBM Journal of Research and Development 44(4), 605 (2000)
16. Golub, G., Van Loan, C.: Matrix Computations, 3rd edn. Johns Hopkins University Press, Baltimore (1996)
17. Stewart, G.W.: Matrix Algorithms, 1st edn., vol. 1. SIAM, Philadelphia (1998)
18. Yip, E.L.: FORTRAN Subroutines for Out-of-Core Solutions of Large Complex Linear Systems. Technical Report CR-159142, NASA (November 1979)
19. Quintana-Orti, E., van de Geijn, R.: Updating an LU factorization with pivoting, Technical Report TR-2006-42, The University of Texas at Austin, Department of Computer Sciences (2006), FLAME Working Note 21
20. Gunter, B.C., van de Geijn, R.A.: Parallel out-of-core computation and updating of the QR factorization. ACM Trans. Math. Softw. 31(1), 60–78 (2005)
21. Berry, M.W., Dongarra, J.J., Kim, Y.: A parallel algorithm for the reduction of a nonsymmetric matrix to block upper-hessenberg form. Parallel Comput. 21(8), 1189–1211 (1995)
22. Gustavson, F.G.: New generalized data structures for matrices lead to a variety of high performance algorithms. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 418–436. Springer, Heidelberg (2002)
23. Bischof, C., van Loan, C.: The WY representation for products of householder matrices. SIAM J. Sci. Stat. Comput. 8(1), 2–13 (1987)
24. Schreiber, R., van Loan, C.: A storage-efficient WY representation for products of Householder transformations. SIAM J. Sci. Stat. Comput. 10(1), 53–57 (1989)
25. Bischof, C., van Loan, C.: The WY representation for products of householder matrices. SIAM J. Sci. Stat. Comput. 8(1), 2–13 (1987)
26. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: Parallel Tiled QR Factorization for Multicore Architectures. Technical Report UT-CS-07-598, University of Tennessee (2007), LAPACK Working Note 190