A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures

LAPACK Working Note # 191

Alfredo Buttari[1], Julien Langou[4], Jakub Kurzak[1], Jack Dongarra[123]

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

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 Cholesky, LU and 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. This 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 algorithms where parallelism can only be exploited at the level of the BLAS operations and vendor implementations.

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. Multicore processors clearly represent the future of computing. It is easy to imagine that multicore technologies will have a deep impact on the High Performance Computing (HPC) world where high processor counts are involved and, thus, limiting power consumption and heat dissipation is a major requirement. The Top500 [1] list released in June 2007 shows that the number of systems based on the dual-core Intel Woodcrest processors grew in six months (i.e. from the previous list) from 31 to 205 and that 90 more systems are based on dual-core AMD Opteron processors.

Even if many attempts have been made in the past to develop parallelizing compilers, they proved themselves efficient only on a restricted class of problems. As a result, at this stage of the multicore era, programmers cannot rely on compilers to take advantage of the multiple execution units present on a processor. 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 [22] architecture or the Intel Polaris[3] 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. Moreover, asynchronous execution models allow the hiding of latency of access to memory.

The LAPACK [5] and ScaLAPACK [9] 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. In both cases exploitation of parallelism comes from the availability of parallel BLAS. In the LAPACK case, a number of BLAS libraries can be used to take advantage of multiple processing units on shared memory systems; for example, the freely distributed ATLAS [26] and GotoBLAS [15] or other vendor BLAS like Intel MKL [2] are popular choices. These parallel BLAS libraries use common techniques for shared memory parallelization like pThreads [21] or OpenMP [11]. This is represented in Figure 1 (*left*).

In the ScaLAPACK case, parallelism is exploited by PBLAS [10] which is a parallel BLAS implementation that uses the Message Passing Interface [13]

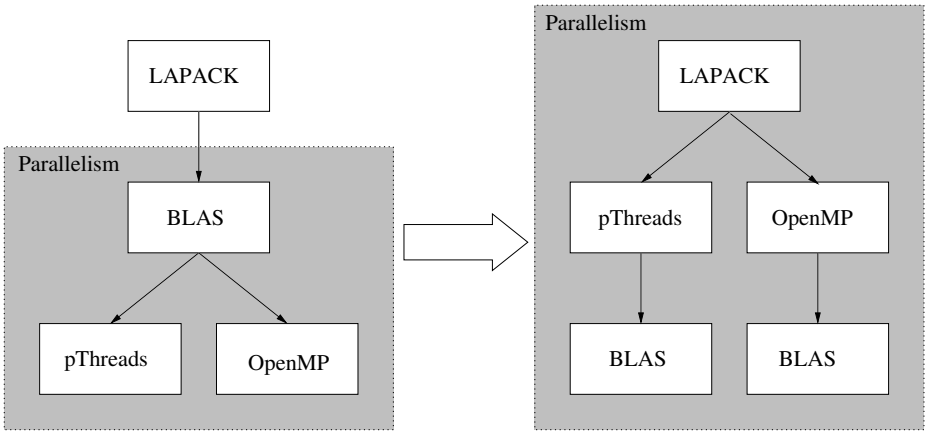

Fig. 1. Transition from sequential algorithms that rely on parallel BLAS to parallel algorithms.

(MPI) for communications on a distributed memory system. Substantially, both LAPACK and ScaLAPACK implement sequential algorithms that rely on parallel building blocks (i.e., the 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. This transition is shown in Figure 1. An approach along these lines has already been proposed in [7, 8, 20] where operations in the standard LAPACK algorithms for some common factorizations were 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 [19].

The rest of this document shows how this can be achieved for the Cholesky, LU and QR factorizations. Section 2 describes the block algorithms used in the LAPACK library and presents their limitations on parallel, shared memory system; Section 3 describes fine granularity, tiled algorithms for the Cholesky, LU and QR factorizations and presents a framework for their asynchronous and dynamic execution; performance results for this algorithm are shown in Section 5.

2 Block Algorithms

2.1 Description of the Block Algorithms in LAPACK and their Scalability Limits

The LAPACK library provides a broad set of Linear Algebra operations aimed at achieving high performance on systems equipped with memory hierarchies.

The algorithms implemented in LAPACK leverage the idea of blocking to limit the amount of bus traffic in favor of a high reuse of the data that is present in the higher level memories which are also the fastest ones. The idea of blocking revolves around an important property of Level-3 BLAS operations, the so called *surface-to-volume* property, that states that $\mathcal{O}(n^3)$ floating point operations are performed on $\mathcal{O}(n^2)$ data. Because of this property, Level-3 BLAS operations can be implemented in such a way that data movement is limited and reuse of data in the cache is maximized. Blocking algorithms consists in recasting Linear Algebra algorithms (like those implemented in LINPACK) in a way that only a negligible part of computations is done in Level-2 BLAS operations (where no data reuse possible) while the most part is done in Level-3 BLAS.

Most of the LAPACK algorithms can be described as the repetition of two fundamental steps:

panel factorization : depending of the Linear Algebra operation that has to be performed, a number of transformations are computed for a small portion of the matrix (the so called *panel*). These transformations, computed by means of Level-2 BLAS operations, can be accumulated (the way they are accumulated changes depending on the particular operation performed).

trailing submatrix update : in this step, all the transformations that have been accumulated during the panel factorization, can be applied at once to the rest of the matrix (i.e. the trailing submatrix) by means of Level-3 BLAS operations.

Because the panel size is very small compared to the trailing submatrix size, block algorithms are very rich in Level-3 BLAS operations which provides high performance on memory hierarchy systems.

The LAPACK library can use any flavor of parallel BLAS to exploit parallelism on a multicore, shared-memory architecture. This approach, however, has a number of limitations due to the nature of the transformation in the panel factorization. The panel factorization, in fact, is rich in Level-2 BLAS operations that cannot be efficiently parallelized on currently available shared memory machines. To understand this, it is important to note that Level-2 BLAS operations can be, generally speaking, defined as all those operations where $\mathcal{O}(n^2)$ floating-point operations are performed on $\mathcal{O}(n^2)$ floating-point data; thus, the speed of Level-2 BLAS computations is limited by the speed at which the memory bus can feed the cores. On current multicores architectures, there is a vast disproportion between the bus bandwidth and the speed of the cores. For example the Intel Clovertown processor is equipped with four cores each capable of a double precision peak performance of 10.64 GFlop/s (that is to say a peak of 42.56 GFlop/s for four cores) while the bus bandwidth peak is 10.64 GB/s which provides 1.33 GWords/s (a word being a 64 bit double precision number). As a result, since one core is largely enough to saturate the bus, using two or more cores does not provide any significant benefit. The LAPACK algorithms are, thus, characterized by the presence of a sequential operation (i.e., the panel factorizations) which represents a small fraction of the total number of FLOPS performed ($\mathcal{O}(n^2)$ FLOPS for a total of $\mathcal{O}(n^3)$ FLOPS) but limits the scalability

of block factorizations on a multicore system when parallelism is only exploited at the level of the BLAS routines. This approach will be referred to as the *fork-join* approach since the execution flow of a block factorization would show a sequence of sequential operations (i.e. the panel factorizations) interleaved to parallel ones (i.e., the trailing submatrix updates).

	Cholesky		LU		QR	
# cores	`panel` Gflop/s	`total` Gflop/s	`panel` Gflop/s	`total` Gflop/s	`panel` Gflop/s	`total` Gflop/s
1	2.6	7.5	2.1	7.8	1.7	7.7
2	2.5	12.7	2.1	13.6	1.9	14.0
4	2.0	21.2	2.2	21.3	1.8	21.5
8	2.1	15.1	2.0	26.4	1.9	18.8

Table 1. Scalability of the fork-join parallelization on a 2-way Quad Clovertown system (eight cores total).

Table 1 shows the scalability limits of the panel factorization and how this affects the scalability of the whole operation for the Cholesky, LU and QR factorizations respectively on an 2-way quad-core Clovertown system (eight cores total) using the MKL-9.1 parallel BLAS library.

In [7, 20], a solution to this scalability problem is presented. The approach described in [7, 20] consists of breaking the trailing submatrix update into smaller tasks that operate on a block-column (i.e., a set of b contiguous columns where b is the block size). The algorithm can then be represented as a Directed Acyclic Graph (DAG) where nodes represent tasks, either panel factorization or update of a block-column, and edges represent dependencies among them. The execution of the algorithm is performed by asynchronously scheduling the tasks in a way that dependencies are not violated. This asynchronous scheduling results in an out-of-order execution where slow, sequential tasks are hidden behind parallel ones. This approach can be described as a dynamic lookahead technique. Even if this approach provides significant speedup, as shown in [7, 20], it is exposed to scalability problems. Due to the relatively high granularity of the tasks, the scheduling of tasks may have a limited flexibility and the parallel execution of the algorithm may be affected by an unbalanced load.

The following sections describe the application of this idea of dynamic scheduling and out of order execution to a class of algorithms for Cholesky, LU and QR factorizations where finer granularity of the operations and higher flexibility for the scheduling can be achieved.

3 Fine Granularity Algorithms for the Cholesky, LU and QR Factorizations

As described in Section 1, fine granularity is one on the main requirements that is demanded to an algorithm in order to achieve high efficiency on a parallel multicore system. This section shows how it is possible to achieve this fine granularity for the Cholesky, LU and QR factorizations by using "tiled" algorithms. Besides providing fine granularity, the use of tiled algorithms also makes it possible to use more efficient storage format for the data such as Block Data Layout (BDL). The benefits of BDL have been extensively studied in the past, for example in [18], and recent studies [4, 8] demonstrate how fine-granularity parallel algorithms can benefit from BDL.

Section 4 shows how the idea of dynamic scheduling and out of order execution presented in [7, 20] can be applied to these algorithms in order to achieve the other important property described in Section 1, i.e. asynchronicity.

3.1 A Tiled Algorithm for the Cholesky Factorization

Developing a tiled algorithm for the Cholesky factorization is a relatively easy task since each of the elementary operations in the standard LAPACK block algorithm can be broken into a sequence of tasks that operate on small portions of data. The benefits of such approach on parallel multicore systems have been already discussed in the past [8, 19].

The tiled algorithm for Cholesky factorization will be based on the following set of elementary operations:

`DPOTF2` . This LAPACK subroutine is used to perform the unblocked Cholesky factorization of a symmetric positive definite tile A_{kk} of size $b \times b$ producing a unit, lower triangular tile L_{kk}. Thus, using the notation $input \longrightarrow output$, the call `DPOTF2`($A_{kk}$, L_{kk}) will perform

$$A_{kk} \longrightarrow L_{kk} = Cholesky(A_{kk})$$

`DTRSM` . This BLAS subroutine is used to apply the transformation computed by `DPOTF2` to a A_{ik} tile by means of a triangular system solve. The `DTRSM`(L_{kk}, A_{ik}, L_{ik}) performs

$$L_{kk}, A_{ik} \longrightarrow L_{ik} = A_{ik} L_{kk}^{-T}$$

`DGSMM` . This subroutine is used to update the tiles A_{ij} in the trailing submatrix by mean of a matrix-matrix multiply. In the case of diagonal tiles, i.e. A_{ij} tiled where $i = j$, this subroutine will take advantage of their triangular structure. The call `DGSMM`(L_{ik}, L_{jk}, A_{ij})

$$L_{ik}, L_{jk}, A_{ij} \longrightarrow A_{ij} = A_{ij} - L_{ik} L_{jk}^{T}$$

Assume a symmetric, positive definite matrix A of size $n \times n$ where $n = p * b$ for some value b that defines the size of the tiles

$$A = \begin{pmatrix} A_{11} & 0 & \cdots & 0 \\ A_{21} & A_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pp} \end{pmatrix}$$

where all the A_{ij} are of size $b \times b$; then the tiled Cholesky algorithm can be described as in Algorithm 1.

Algorithm 1 Tiled Cholesky factorization

```
1: for k=1,2,...,p do
2:    DPOTF2(A_kk, L_kk)
3:    for i = k + 1, ..., p do
4:       DTRSM(L_kk, A_ik, L_ik)
5:    end for
6:    for i = k + 1, ..., p do
7:       for j = k + 1, ..., r do
8:          DGSMM(L_ik, L_jk, A_ij)
9:       end for
10:   end for
11: end for
```

Note that no extra memory area is needed to store the L_{ij} tiles since they can overwrite the corresponding A_{ij} tiles from the original matrix.

3.2 A Tiled Algorithm for the LU and QR Factorizations

In the case of the LU and QR factorizations, it is not possible to adopt the same approach used for the Cholesky operation. In fact, due to their nature, the elementary operations used in the LAPACK block algorithm cannot be simply broken into sequences of tasks that operate on smaller portions of data. Different algorithms must be used in order to achieve a fine granularity parallelism.

The algorithmic change we propose is actually well-known and takes its roots in updating factorizations [14, 25]. Using updating techniques to tile the algorithms have first[5] been proposed by Yip [27] for LU to improve the efficiency of out-of-core solvers, and were recently reintroduced in [17, 23] for LU and QR, once more in the out-of-core context. A similar idea has also been proposed in [6] for Hessenberg reduction in the parallel distributed context. The efficiency of these algorithms in a parallel multicore system has been discussed, for the QR factorization, in [4]; specifically the algorithm used in [4] is a simplified variant of that discussed in [17] that aims at overcoming the limitations of BLAS

[5] to our knowledge

libraries on small size tiles. The cost of this simplification is an increase in the operation count for the whole QR factorization. In this document the same algorithm as in [17] is used to achieve high efficiency for both the LU and QR factorizations; performance results show that this choice, while limiting the operation count overhead to a negligible amount, still delivers high execution rates as. This approach has been presented for the QR factorization in [16].

A stability analysis for the tiled algorithm for LU factorization may be found in [23].

Tiled Algorithm for the QR Factorization

The description of the tiled algorithm for the QR factorization will be based on the following sets of elementary operations:

`DGEQRT.` This subroutine was developed to perform the blocked factorization of a diagonal block A_{kk} of size $b \times b$ with internal block size s. 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 Householder transformations [24].
Thus, using the notation $input \longrightarrow output$, the call `DGEQRT`($A_{kk}$, V_{kk}, R_{kk}, T_{kk}) performs

$$A_{kk} \longrightarrow (V_{kk}, R_{kk}, T_{kk}) = QR(A_{kk})$$

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

$$A_{kj}, V_{kk}, T_{kk} \longrightarrow R_{kj} = (I - V_{kk}T_{kk}V_{kk}^T)A_{kj}$$

`DTSQRT.` This subroutine was developed to perform the blocked QR factorization of a matrix that is formed by coupling an upper triangular block R_{kk} with a square block A_{ik} with internal block size s. This subroutine will return an upper triangular matrix R_{kk}, an upper triangular matrix T_{ik} as defined by the WY technique for accumulating householder transformations, and a tile V_{ik} containing b Householder reflectors where b is the tile size.
Then, `DTSQRT`(R_{kk}, A_{ik}, V_{ik}, T_{ik}) performs

$$\begin{pmatrix} R_{kk} \\ A_{ik} \end{pmatrix} \longrightarrow (V_{ik}, T_{ik}, R_{kk}) = QR \begin{pmatrix} R_{kk} \\ A_{ik} \end{pmatrix}$$

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

$$\begin{pmatrix} R_{kj} \\ A_{ij} \end{pmatrix}, V_{ik}, T_{ik} \longrightarrow \begin{pmatrix} R_{kj} \\ A_{ij} \end{pmatrix} = (I - V_{ik}T_{ik}V_{ik}^T) \begin{pmatrix} R_{kj} \\ A_{ij} \end{pmatrix}$$

Note that no extra storage is required for the V_{ij} and R_{ij} since those tiles can overwrite the A_{ij} tiles of the original matrix A; a temporary memory area has to be allocated to store the T_{ij} tiles. Assuming a matrix A of size $pb \times qb$

$$\begin{pmatrix} A_{11} & A_{12} & \dots & A_{1q} \\ A_{21} & A_{22} & \dots & A_{2q} \\ \vdots & & \ddots & \vdots \\ A_{p1} & A_{p2} & \dots & A_{pq} \end{pmatrix}$$

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 2.

Algorithm 2 The tiled algorithm for QR factorization.

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

If an unblocked algorithm is used to perform the `DTSQRT` and `DSSRFB` subroutines (as suggested in [4]), this tiled algorithm has a total cost that is 25% higher than the cost of the LAPACK block algorithm (see [4] for details), i.e. $\frac{5}{2}n^2(m - \frac{n}{3})$ (where $m = p \cdot b$ and $n = q \cdot b$) versus $2n^2(m - \frac{n}{3})$. As suggested in [16, 17], a block approach can be used for the operations implemented in the `DTSQRT` and `DSSRFB` subroutines with a block size $s \ll b$. To understand how this cuts the operation count of the tiled algorithm, it is important to note that the `DGEQRT`, `DLARFB` and `DTSQRT` only account for lower order terms in the total operation count for the tiled algorithm. It is, thus, possible to ignore these terms and derive the operation count for the tiled algorithm for QR factorization as the sum of the cost of all the `DSSRFB` calls. The cost of a single `DSSRFB` call, ignoring the lower order terms, is $4b^3 + sb^2$ and, assuming $q < p$, the total cost of the tiled algorithm with internal blocking is

$$\sum_{k=1}^{q}(4b^3 + sb^2)(p - k)(q - k) \simeq 2n^2(m - \frac{n}{3})(1 + \frac{s}{4b}). \quad (1)$$

The operation count for the tiled QR algorithm with internal blocking is bigger than that of the LAPACK algorithm only by the factor $(1 + \frac{s}{4b})$ which is

negligible, provided that $s \ll b$. Note that, in the case where $s = b$, the tiled algorithm performs 25% more floating point operations than the LAPACK block algorithm, as stated before.

Tiled Algorithm for the LU Factorization

The description of the tiled algorithm for the LU factorization will be based on the following sets of elementary operations.

`DGETRF`. This subroutine performes a block LU factorization of a tile A_{kk} of size $b \times b$ with internal block size s. As a result, two matrices L_{kk} and U_{kk}, unit-lower and upper triangular respectively, and a permutation matrix P_{kk} are produced. Thus, using the notation $input \longrightarrow output$, the call `DGETRF`($A_{kk}$, L_{kk}, U_{kk}, P_{kk}) will perform

$$A_{kk} \longrightarrow L_{kk}, U_{kk}, P_{kk} = LU(A_{kk})$$

`DGESSM`. This routine was developed to apply the transformation (L_{kk}, P_{kk}) computed by the `DGETRF` subroutine to a tile A_{kj}. thus the call `DGESSM`(A_{kj}, L_{kk}, P_{kk}, U_{kj}) will perform

$$A_{kj}, L_{kk}, P_{kk} \longrightarrow U_{kj} = L_{kk}^{-1} P_{kk} A_{kj}$$

`DTSTRF`. This subroutine was developed to perform the block LU factorization of a matrix that is formed by coupling an upper triangular block U_{kk} with a square block A_{ik} with internal block size s. This subroutine will return an upper triangular matrix U_{kk}, a unit, lower triangular matrix L_{ik} and a permutation matrix P_{ik}. Thus, the call `DTSTRF`(U_{kk}, A_{ik}, P_{ik}) will perform

$$\begin{pmatrix} U_{kk} \\ A_{ik} \end{pmatrix} \longrightarrow U_{kk}, L_{ik}, P_{ik} = LU \begin{pmatrix} U_{kk} \\ A_{ik} \end{pmatrix}$$

`DSSSSM`. This subroutine was developed to update the matrix formed by coupling two square blocks U_{kj} and A_{ij} applying the transformation computed by `DTSTRF`. Thus the call `DSSSSM`(U_{kj}, A_{ij}, L_{ik}, P_{ik}) performs

$$\begin{pmatrix} U_{kj} \\ A_{ij} \end{pmatrix}, L_{ik}, P_{ik} \longrightarrow \begin{pmatrix} U_{kj} \\ A_{ij} \end{pmatrix} = L_{ik}^{-1} P_{ik} \begin{pmatrix} U_{kj} \\ A_{ij} \end{pmatrix}$$

Note that no extra storage is required for the U_{ij} since they can overwrite the correspondent A_{ij} tiles of the original matrix A. A memory area must be allocated to store the P_{ij} and part of the L_{ij}; the L_{ij} tiles, in fact, are $2b \times b$ matrices, i.e. two tiles arranged vertically and, thus, one tile can overwrite the corresponding A_{ij} tile and the other is stored in the extra storage area[6].

[6] the upper part of L_{ij} is, actually, a group of b/s unit, lower triangular matrices each of size $s \times s$ and, thus, only a small memory area is required to store it.

Assuming a matrix A of size $pb \times qb$

$$\begin{pmatrix} A_{11} & A_{12} & \dots & A_{1q} \\ A_{21} & A_{22} & \dots & A_{2q} \\ \vdots & & \ddots & \vdots \\ A_{p1} & A_{p2} & \dots & A_{pq} \end{pmatrix}$$

where b is the block size and each A_{ij} is of size $b \times b$, the LU factorization can be performed as in Algorithm 3.

Algorithm 3 The tiled algorithm for LU factorization.

```
1: for k = 1, 2..., min(p, q) do
2:    DGETRF(A_kk, L_kk, U_kk, P_kk)
3:    for j = k + 1, k + 2, ..., q do
4:       DGESSM(A_kj, L_kk, P_kk, U_kj)
5:    end for
6:    for i = k + 1, k + 1, ..., p do
7:       DTSTRF(U_kk, A_ik, P_ik)
8:       for j = k + 1, k + 2, ..., q do
9:          DSSSSM(U_kj, A_ij, L_ik, P_ik)
10:      end for
11:   end for
12: end for
```

If an unblocked algorithm is used to perform the `DTSTRF` and `DSSSM` subroutines, this tiled algorithm has a total cost that is 50% higher than the cost of the LAPACK block algorithm, i.e. $\frac{3}{2}n^2(m - \frac{n}{3})$ (where $m = p \cdot b$ and $n = q \cdot b$) versus $n^2(m - \frac{n}{3})$. In this tiled algorithm for LU factorization, as in the case of the tiled algorithm for QR factorization, a block approach can be used for the operations implemented in the `DTSTRF` and `DSSSSM` subroutines with a block size $s \ll b$. To understand how this cuts the operation count of the tiled algorithm, it is important to note that the `DGETRF`, `DGESSM` and `DTSTRF` only account for lower order terms in the total operation count for the tiled algorithm. It is, thus, possible to ignore these terms and derive the operation count for the tiled algorithm for LU factorization as the sum of the cost of all the `DSSSSM` calls. The cost of a single `DSSSSM` call, ignoring the lower order terms, is $2b^3 + sb^2$ and, assuming $q < p$, the total cost of the tiled algorithm with internal blocking is

$$\sum_{k=1}^{q} (2b^3 + sb^2)(p - k)(q - k) \simeq n^2(m - \frac{n}{3})(1 + \frac{s}{2b}). \tag{2}$$

The operation count for the tiled LU algorithm with internal blocking is bigger than that of the LAPACK algorithm only by the factor $(1 + \frac{s}{2b})$ which is negligible, provided that $s \ll b$. Note that, in the case where $s = b$, the tiled

algorithm performs 50% more floating point operations than the LAPACK block algorithm, as stated before.

Since the only difference between Algorithms 2 and 3 is the elementary operations, and noting, as explained before, that the R_{ij}, V_{ij}, U_{ij} and L_{ij} tiles are stored in the corresponding memory locations that contain the tiles A_{ij} of the original matrix A (the L_{ij} only partially), a graphical representation of Algorithms 2 and 3 is as in Figure 2.

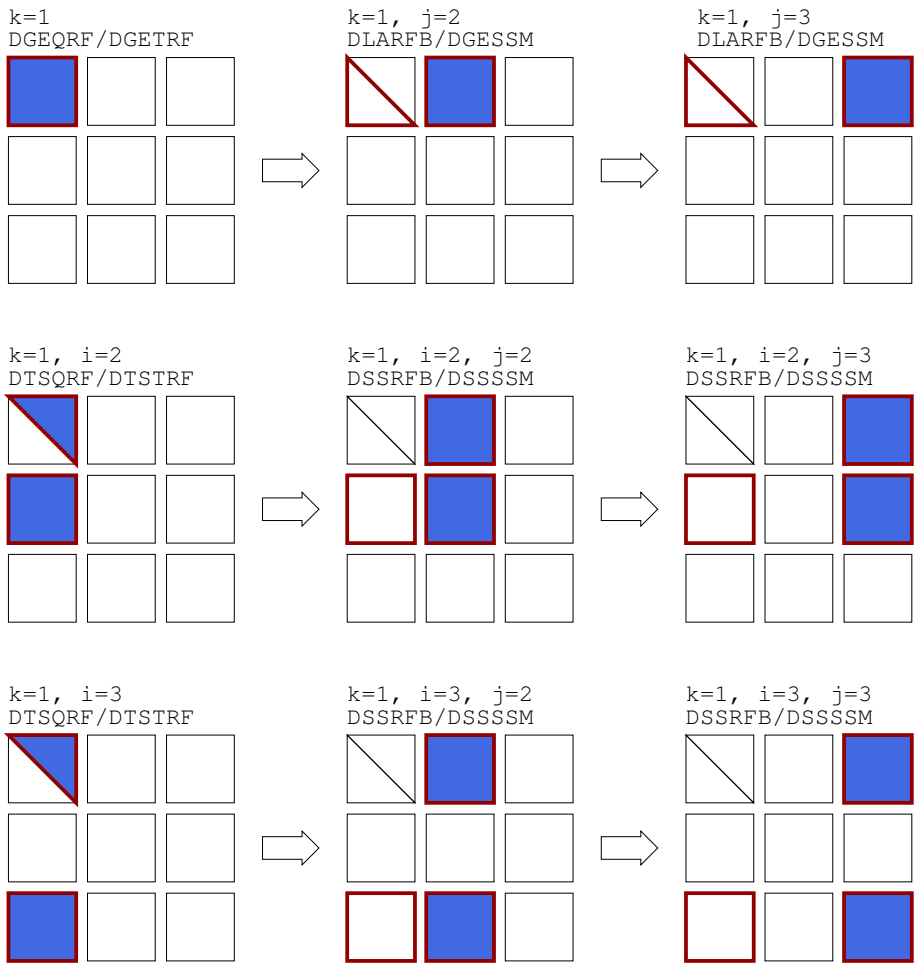

Fig. 2. Graphical representation of one repetition of the outer loop in Algorithms 2 and 3 on a matrix with $p = q = 3$. As expected the picture is very similar to the out-of-core algorithm presented in [17].

4 Graph driven asynchronous execution

Following the approach presented in [4, 7, 20], Algorithms 1, 2 and 3 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 3 show the DAG for the tiled QR factorization when Algorithm 2 is executed on a matrix with $p = q = 3$. Note that these DAGs have 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.

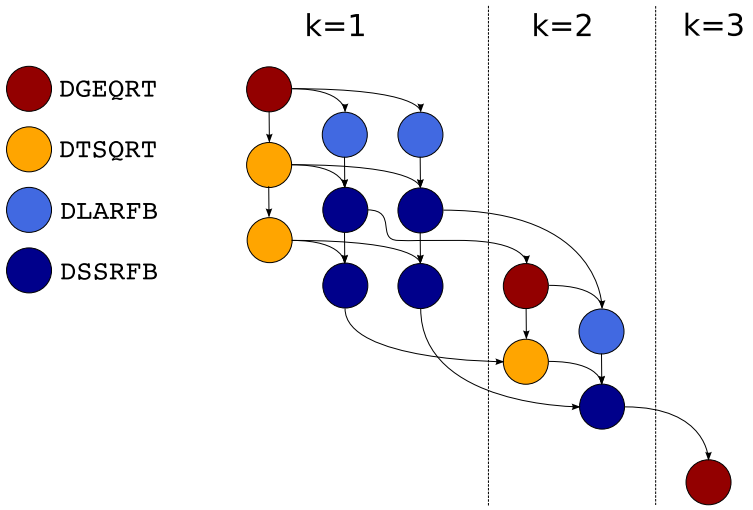

Fig. 3. The dependency graph of Algorithm 2 on a matrix with $p = q = 3$.

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 `DGEQRT` subroutine have the highest priority and then three other priority levels can be defined for `DTSQRT`, `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. Figure 4 shows part of the execution flow of Algorithm 2 on a 8-cores machine (2-way Quad Clovertown) when tasks are dynamically sched-

uled based on dependencies in the DAG. Each line in the execution flow shows which tasks are performed by one of the threads involved in the factorization.

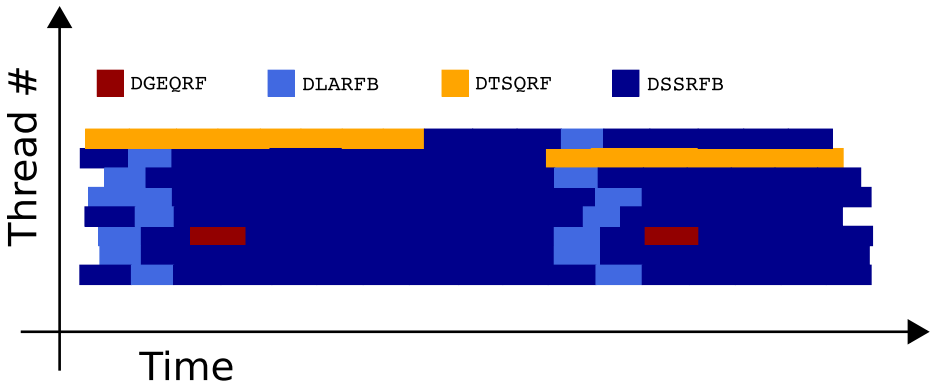

Fig. 4. The execution flow for dynamic scheduling, out of order execution of Algorithm 2.

Figure 4 shows that all the idle times, which represent the major scalability limit of the fork-join approach, can be removed thanks to the very low synchronization requirements of the graph driven execution. The graph driven execution also provides some degree of adaptivity since tasks are scheduled to threads depending on the availability of execution units.

5 Performance Results

The performance of the tiled algorithms for Cholesky, QR ad LU factorizations with dynamic scheduling of tasks has been measured on the system described in Table 2 and compared to the performance of the MKL-9.1 implementations and to the fork-join approach, i.e., the standard algorithm for block factorizations of LAPACK associated with multithreaded BLAS (MKL-9.1).

Figures 5, 6, 7 report the performance of the Cholesky, QR and LU factorizations for the tiled algorithms with dynamic scheduling, the MKL-9.1 implementation and the LAPACK block algorithms with multithreaded BLAS. For the tiled algorithms, the tile size and (for QR and LU) the internal blocking size have been chosen in order to achieve the best performance possible. As a reference, the tile size is in the range of 200 and the internal blocking size in the range of 20-40. In the case of the LAPACK block algorithms, the block size [7] has been tuned in order to achieve the best performance.

[7] the block size in the LAPACK algorithm sets the width of the panel.

	2-way quad Clovertown
Architecture	Intel®Xeon®CPU X5355
Clock speed	2.66 GHz
# cores	$2 \times 4 = 8$
Peak performance	85.12 Gflop/s
Memory	16 GB
Compiler suite	Intel 9.1
BLAS library	MKL-9.1.023

Table 2. Details of the system used for the following performance results.

The graphs on the right part of each figure show the performance measured using the maximum number of cores available on the system (i.e. eight) with respect to the problem size. The graphs on the left 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. The axis of ordinates has been scaled to reflect the theoretical peak performance of the system (i.e. the top value is 85.12 Gflop/s) and, also, as a reference, the performance of the matrix-matrix multiply (DGEMM) has been reported in the right-side graphs. In order to reflect the time to completion, for each operation the operation count of all the algorithms is assumed to be the same as that of the LAPACK block algorithm. In the case of the tiled LU and QR factorizations, this assumption is only slightly false since the amount of extra flops can be considered negligible for a correct choice of the internal blocking size s.

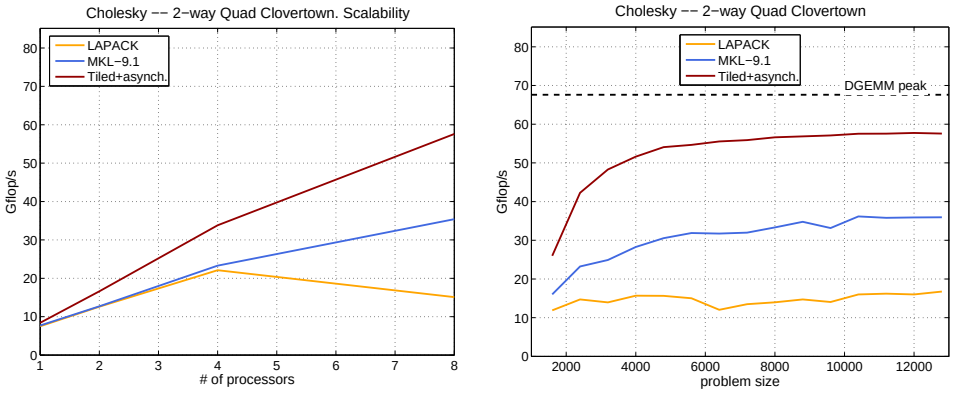

Fig. 5. Cholesky factorization: comparison between the performance of the tiled algorithm with dynamic scheduling, the MKL-9.1 implementation and the LAPACK block algorithm with MKL-9.1 multithreaded BLAS.

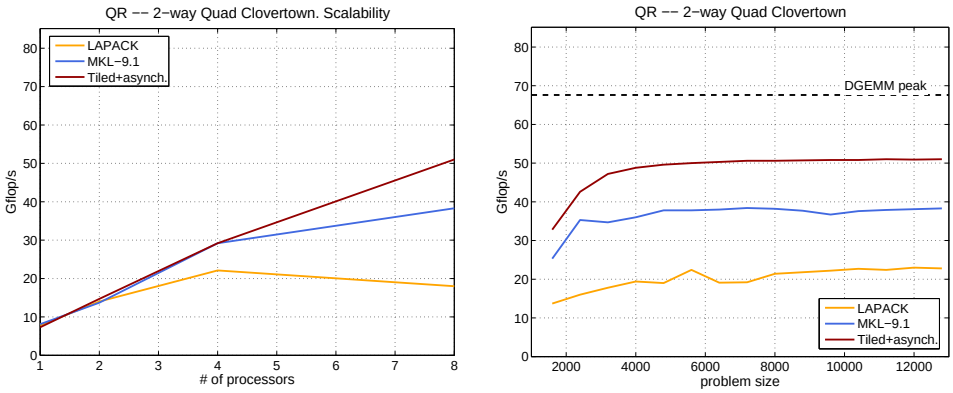

Fig. 6. QR factorization: comparison between the performance of the tiled algorithm with dynamic scheduling, the MKL-9.1 implementation and the LAPACK block algorithm with MKL-9.1 multithreaded BLAS.

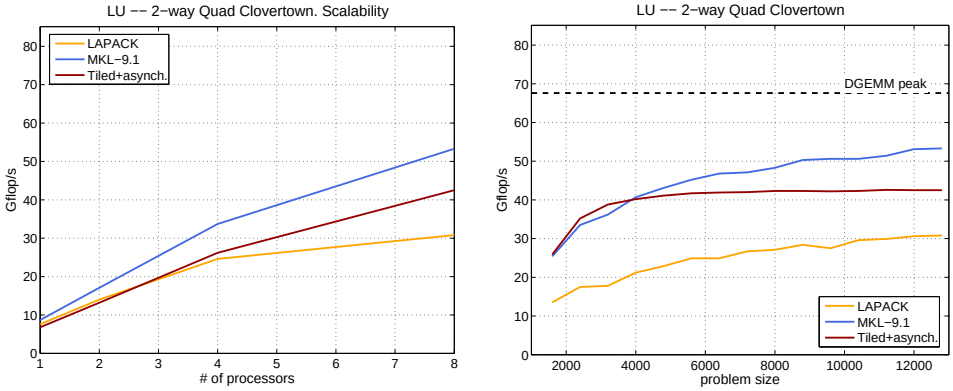

Fig. 7. LU factorization: comparison between the performance of the tiled algorithm with dynamic scheduling, the MKL-9.1 implementation and the LAPACK block algorithm with MKL-9.1 multithreaded BLAS.

Figures 5 and 6 provide roughly the same information: the tiled algorithm combined with asynchronous graph driven execution delivers higher execution rates than the fork-join approach (i.e. LAPACK block algorithm with multithreaded BLAS) and also than a vendor implementation of the operation. An important remark has to me made for the Cholesky factorization: the *left-looking* variant (see [12] for more details) of the block algorithm is implemented in LAPACK. This variant delivers very poor performance when compared to the *right-looking* one; a sequential *right-looking* implementation of the Cholesky factorization that uses multithreaded BLAS would run at higher speed than that measured on the LAPACK version.

In the case of the LU factorization, even if it still provides a considerable speedup with respect to the fork-join approach, the tiled algorithm is still slower than the vendor implementation. This is mostly due to two main reasons:

1. pivoting: in the block LAPACK algorithm, entire rows are swapped at once and, at most, n swaps have to be performed where n is the size of the problem. With pairwise pivoting, which is the pivoting scheme adopted in the tiled algorithm, at most $n^2/(2b)$ can happen and all the swaps are performed in a very inefficient way since rows are swapped in pieces of size b.
2. internal blocking size: as shown by Equation (2), the flop count of the tiled algorithm grows by a factor of $1 + s/(2b)$. To keep this extra cost limited to a negligible amount, a very small internal block size s has to be chosen. This results in a performance loss due to the limitations of BLAS libraries on small size data.

6 Conclusions

Even if a definition of multicore processor is still lacking, with some speculation it is possible to define a limited set of characteristics that a software should have in order to efficiently take advantage of multiple execution units on a chip. Early work [4, 7, 19, 20] on this subject suggested that fine granularity and asynchronous execution models are desirable properties in order to achieve high performance on multicore architectures due to high degrees of parallelism, increased importance of local data reuse and the necessity to hide the latency of access to memory. Performance results presented in Section 5 support this reasoning by showing how the usage of fine granularity, tiled algorithms together with a graph driven, asynchronous execution model can provide considerable benefits over the traditional fork-join approach and also vendor implementations where usually a significant amount of work is done to address cache behavior. Even if performance results are presented here for a single architecture, the proposed approach has a much wider scope and portability. Other work by the authors shows, for example, how tiled algorithms perfectly match the architectural features of the Cell Broadband Engine processor [19]. In addition it is important to note that blocking techniques can still be applied to tiled algorithms which makes them rich in Level-3 BLAS operations and, thus, efficient on systems equipped with memory hierarchies.

Bibliography

[1] http://top500.org.

[2] http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.

[3] Teraflops research chip. http://www.intel.com/research/platform/terascale/teraflops.htm.

[4] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel Tiled QR Factorization for Multicore Architectures. Technical report, University of Tennessee Knoxville, 2007. UT-CS-07-598. 23-July-2007.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, 3. edition, 1999.

[6] M. W. Berry, J. J. Dongarra, and Y. Kim. A parallel algorithm for the reduction of a nonsymmetric matrix to block upper-hessenberg form. *Parallel Comput.*, 21(8):1189–1211, 1995.

[7] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. In *PARA 2006, Umeå Sweden*, June 2006.

[8] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. Supermatrix out-of-order scheduling of matrix operations for SMP and multicore architectures. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 116–125, New York, NY, USA, 2007. ACM Press.

[9] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. 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).

[10] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. W. Walker, and R. Clinton Whaley. A proposal for a set of parallel basic linear algebra subprograms. In *PARA '95: Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science*, pages 107–114, London, UK, 1996. Springer-Verlag.

[11] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 05(1):46–55, 1998.

[12] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. Vander Vorst. *Numerical Linear Algebra for High Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[13] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[14] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

[15] K. Goto and R. van de Geijn. High-performance implementation of the level-3 blas. Technical Report TR-2006-23, The University of Texas at Austin, Department of Computer Sciences., 2006. FLAME Working Note 20.

[16] Gregorio Quintana Orti, Enrique Quintana Orti, Ernie Chan Field G. Van Zee, and Robert van de Geijn. Scheduling of QR Factorization Algorithms on SMP and Multi-Core Architectures. Technical report, The University of Texas at Austin, Department of Computer Sciences, 2007. Flame Working Note 24. 31-July-2007.

[17] B. C. Gunter and R. A. van de Geijn. Parallel out-of-core computation and updating of the QR factorization. *ACM Trans. Math. Softw.*, 31(1):60–78, 2005.

[18] F. G. Gustavson. New generalized data structures for matrices lead to a variety of high performance algorithms. In *PPAM '01: Proceedings of the th International Conference on Parallel Processing and Applied Mathematics-Revised Papers*, pages 418–436, London, UK, 2002. Springer-Verlag.

[19] J. Kurzak, A. Buttari, and J. Dongarra. 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.

[20] J. Kurzak and J. Dongarra. 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.

[21] F. Mueller. Pthreads library interface, 1993.

[22] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *IEEE International Solid-State Circuits Conference*, pages 184–185, 2005.

[23] E. Quintana-Orti and R. van de Geijn. 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.

[24] R. Schreiber and C. van Loan. A storage-efficient WY representation for products of Householder transformations. *SIAM J. Sci. Stat. Comput.*, 10(1):53–57, 1989.

[25] G. W. Stewart. *Matrix Algorithms*, volume 1. SIAM, Philadelphia, 1. edition, 1998.

[26] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–25, 2001.

[27] E. L. Yip. FORTRAN Subroutines for Out-of-Core Solutions of Large Complex Linear Systems. Technical Report CR-159142, NASA, November 1979.