

# A class of parallel tiled linear algebra algorithms for multicore architectures

Alfredo Buttari<sup>a</sup>, Julien Langou<sup>b</sup>, Jakub Kurzak<sup>a</sup>, Jack Dongarra<sup>a,c,d,\*</sup>

<sup>a</sup> Department of Electrical Engineering and Computer Science, University Tennessee, Knoxville, TN, United States

<sup>b</sup> Department of Mathematical and Statistical Sciences, University of Colorado Denver, CO, United States

<sup>c</sup> Oak Ridge National Laboratory, Oak Ridge, TN, United States

<sup>d</sup> University of Manchester, Manchester, UK

## ARTICLE INFO

### Article history:

Received 6 October 2007

Received in revised form 12 October 2008

Accepted 13 October 2008

Available online 1 November 2008

### Keywords:

Linear algebra

Mathematical software

High performance computing

Multicore

## 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 algorithms 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 out of order execution of tasks which will completely hide the presence of intrinsically sequential tasks in the factorization. Performance comparisons are presented with LAPACK algorithms where parallelism can only be exploited at the level of the BLAS operations and vendor implementations.

Published by Elsevier B.V.

## 1. Introduction

In the last 20 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 list [1] 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 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 run on

\* Corresponding author. Address: Department of Electrical Engineering and Computer Science, University Tennessee, Knoxville, TN, United States.  
E-mail addresses: [dongarra@eecs.utk.edu](mailto:dongarra@eecs.utk.edu), [dongarra@utk.edu](mailto:dongarra@utk.edu) (J. Dongarra).

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 task-level parallelism 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 this type of architecture will scale 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 Cell Broadband Engine Architecture [39] or the Intel Polarix prototype [4]). This requires splitting an operation into tasks that operate on small portions of data in order to reduce bus traffic and improve data locality. Moreover, for those architectures where cache memories are replaced by local memories, like the Cell Broadband Engine, fine granularity is the only means to achieve parallelism as suggested in previous work by the authors [35,37];
- asynchronicity: as the degree of TLP grows and the granularity of the operations becomes smaller, the presence of synchronization points in a parallel execution seriously affects the efficiency of an algorithm. Moreover, using asynchronous execution models it is possible to hide the latency of access to memory. The use of dynamic tasks execution in dense linear algebra was studied as long ago as early 1980s [38].

Section 2 shows why such properties cannot be achieved on algorithms implemented in commonly used linear algebra libraries due to their scalability limits in the context of multicore computing. Section 3 describes fine granularity, tiled algorithms for the Cholesky, LU and QR factorizations and presents a programming model for their asynchronous and dynamic execution; performance results for these algorithms are shown in Section 5.

## 2. The LAPACK and ScaLAPACK libraries and their scalability limits

The LAPACK [8] and ScaLAPACK [14] 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.<sup>1</sup> In both cases exploitation of parallelism comes from the availability of parallel BLAS.

The algorithms implemented in these two packages 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. This is achieved by recasting linear algebra algorithms (like those implemented in LINPACK) in a way that, for the most part, computation is done by Level-3 BLAS operations, where data reuse is guaranteed by the so called *surface-to-volume* effect, and only a small part in Level-2 BLAS, for which memory bus speed constitutes a performance upper bound [18]. As a result, such algorithms can be roughly 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 provide high performance on memory hierarchy systems.

Both LAPACK and ScaLAPACK only exploit parallelism at the BLAS level, i.e., by means of multithreaded BLAS libraries (GotoBLAS [26], MKL [2], ATLAS [46], ESSL [5], ...) in the former case and by means of the PBLAS [15] library in the latter. Because Level-2 BLAS operations cannot be efficiently parallelized on shared memory (multicore) architectures due to the bus bottleneck, exploitation of parallelism only at the BLAS level introduces a fork-join execution pattern where:

- scalability is limited by the fact that the relative cost of strictly sequential operations (i.e., the panel factorization) increases when the degree of parallelism grows,
- asynchronicity cannot be achieved because multiple threads are forced to wait in an idle state for the completion of sequential tasks.

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

<sup>1</sup> Here and in what follows, with LAPACK and ScaLAPACK we refer exclusively to the libraries reference implementations.

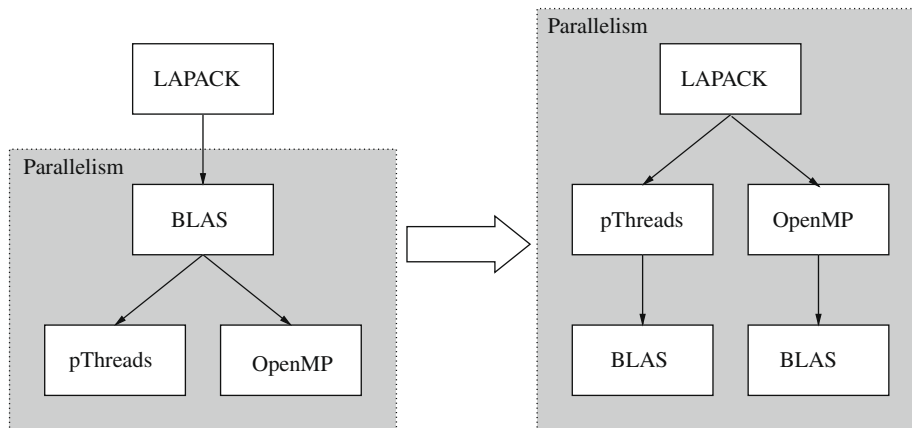


Fig. 1. Transition from sequential algorithms that rely on parallel BLAS to parallel algorithms.

these new architectures by exploiting parallelism at a higher level. This transition is shown in Fig. 1. Approaches along these lines have already been studied in the 1980s [6,7,38].

The technique described in [11,36] 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. Even if this approach provides significant speedup, as shown in [11,36], 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. Moreover, such a 1-D partitioning of the computational tasks is not suited for such architectures like the Cell processor where memory requirements impose a much smaller granularity. The work described here aims at overcoming these limitations based on the usage of the *tiled* algorithms described in Section 3.

The following sections describe the application of the idea of dynamic scheduling and out of order execution to a class of algorithms for the Cholesky, LU and QR factorizations, where finer granularity of the operations and higher flexibility for the scheduling can be achieved. Fine granularity is obtained by using algorithms where the whole factorization can be described as a sequence of tasks that operate on small, square, portions of a matrix (i.e., tiles). Asynchronicity is obtained by executing such algorithms according to a dynamic, graph driven model.

### 3. Fine granularity algorithms for the Cholesky, LU and QR factorizations

As described in Section 1, fine granularity is one of the main requirements that is demanded from 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.

It is crucial to note that the technique of processing the matrix by square blocks only provides performance in tandem with data organization by square blocks, a fact initially observed by Gustavson [29,30] and recently investigated in depth by Gustavson et al. [32]. The layout is referred to as *square block* (SB) format by Gustavson et al. and as *block data layout* (BDL) in this work. The paper by Elmroth et al. [22] gives an excellent introduction to many of the important issues concerning deep memory hierarchies and the use of recursion and hybrid data structures and also contains a section on the QR factorization.

Section 4 shows how the idea of dynamic scheduling and out of order execution, already discussed in [11,36], can be applied to these algorithms in order to achieve the other important property described in Section 1, i.e. asynchronicity. These ideas are not new and have been proposed a number of times in the past [6,7,16,38].

#### 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 [13,28,31,35].

The tiled algorithm for Cholesky factorization will be based on the following set of kernel subroutines:

**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*  $\rightarrow$  *output*, the call  $\text{DPOTF2}(A_{kk}, L_{kk})$  will perform

$$A_{kk} \rightarrow L_{kk} = \text{Cholesky}(A_{kk}).$$

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

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

**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.  $A_{ij}$  tile where  $i = j$ , this subroutine takes advantage of the symmetry of the matrix. The call  $\text{DGSMM}(L_{ik}, L_{jk}, A_{ij})$

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

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

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

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

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

Although the same approach as for Cholesky can also be applied to the LU and QR factorizations [30], this method, which consists of simply rearranging the LAPACK algorithm in terms of operations by tiles, incurs into efficiency problems due to the fact that, in a panel factorization step, each of the tiles that compose the panel is accessed multiple times.

#### Algorithm 1. Tiled Cholesky factorization.

```

1. for  $k = 1, \dots, p$  do
2.    $\text{DPOTF2}(A_{kk}, L_{kk})$ 
3.   for  $i = k + 1, \dots, p$  do
4.      $\text{DTRSM}(L_{kk}, A_{ik}, L_{ik})$ 
5.   endfor
6.   for  $i = k + 1, \dots, p$  do
7.     for  $j = k + 1, \dots, i$  do
8.        $\text{DGSMM}(L_{ik}, L_{jk}, A_{ij})$ 
9.     endfor
10.  endfor
11. endfor

```

The algorithms presented here rely on *updating factorizations* [25,44]. Methods for updating matrix factorizations were first discussed by Gill et al. [24]. Yip [48] applied the idea to the LU factorization to produce an efficient out-of-core solver. Recently, this approach was re-introduced, again in the context of out-of-core solvers [27,34,40], for the LU and QR factorizations (along with stability analysis [40]). A similar idea has also been proposed in [9] for Hessenberg reduction in the parallel distributed context.

#### 3.2.1. Tiled algorithm for the QR factorization

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

**DGEQRT.** This subroutine was developed to perform the block QR 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 compact WY technique

for accumulating Householder transformations [42]. This kernel subroutine is based on the LAPACK `DGEQRF` one and, thus, it consists mostly of Level-3 BLAS operations; in addition to the LAPACK subroutine, `DGEQRT` also computes the  $T_{kk}$  matrix. Thus, using the notation *input*  $\rightarrow$  *output*, the call `DGEQRT`( $A_{kk}, V_{kk}, R_{kk}, T_{kk}$ ) performs

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

**DLARFB.** This LAPACK subroutine, based exclusively on Level-3 BLAS operations, will be used to apply the transformation  $(V_{kk}, T_{kk})$  computed by subroutine `DGEQRT` to a tile  $A_{kj}$  producing a  $R_{kj}$  tile. Thus, `DLARFB`( $A_{kj}, V_{kk}, T_{kk}, R_{kj}$ ) performs

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

**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 compact 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

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

**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

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

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. Further details on the implementation of the `DTSQRT` and `DSSRFB` are provided in Section 3.2.3.

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  $k = 1, \dots, \min(p, q)$  do
2.   DGEQRT( $A_{kk}, V_{kk}, R_{kk}, T_{kk}$ )
3.   for  $j = k + 1, \dots, q$  do
4.     DLARFB( $A_{kj}, V_{kk}, T_{kk}, R_{kj}$ )
5.   endfor
6.   for  $i = k + 1, \dots, p$  do
7.     DTSQRT( $R_{kk}, A_{ik}, V_{ik}, T_{ik}$ )
8.     for  $j = k + 1, \dots, q$  do
9.       DSSRFB( $R_{kj}, A_{ij}, V_{ik}, T_{ik}$ )
10.    endfor
11.  endfor
12. endfor

```

### 3.2.2. Tiled algorithm for the LU factorization

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

**DGETRF.** This LAPACK subroutine, consisting mostly of Level-3 BLAS operations, performs 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*  $\rightarrow$  *output*, the call `DGETRF`( $A_{kk}, L_{kk}, U_{kk}, P_{kk}$ ) will perform

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

**DGESSM.** This routine, based on Level-3 BLAS operations, 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

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

**DTSTRF.** This subroutine was developed to perform the block LU factorization of a matrix that is formed by coupling the 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

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

**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

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

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.<sup>2</sup> Further details on the implementation of the `DTSTRF` and `DSSSSM` are provided in Section 3.2.3.

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  $k = 1, \dots, \min(p, q)$  do
2.   DGETRF( $A_{kk}$ ,  $L_{kk}$ ,  $U_{kk}$ ,  $P_{kk}$ )
3.   for  $j = k + 1, \dots, q$  do
4.     DGEESM( $A_{kj}$ ,  $L_{kk}$ ,  $P_{kk}$ ,  $U_{kj}$ )
5.   endfor
6.   for  $i = k + 1, \dots, p$  do
7.     DTSTRF( $U_{kk}$ ,  $A_{ik}$ ,  $P_{ik}$ )
8.     for  $j = k + 1, \dots, q$  do
9.       DSSSSM( $U_{kj}$ ,  $A_{ij}$ ,  $L_{ik}$ ,  $P_{ik}$ )
10.    endfor
11.  endfor
12. endfor

```

Since the only difference between Algorithms 2 and 3 is in the kernel subroutines, 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 Fig. 2.

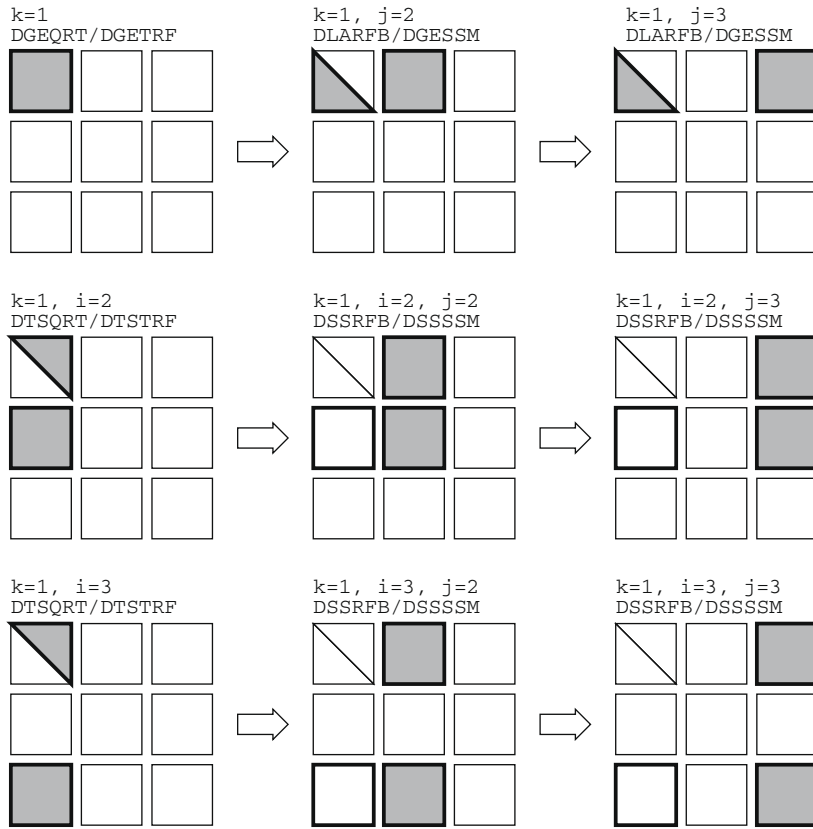
### 3.2.3. Reducing the cost of the tiled algorithms for the QR and LU factorization

This section presents the discussion on reduction of extra floating point operations resulting from aggregation of transformations in the LU and QR factorizations. Since the technique is almost identical for both factorizations, the QR factorization is discussed here.

The most important issue in performance optimization of orthogonal transformations is aggregation of transformations leading to efficient use of the memory system. The idea was first demonstrated by Dongarra et al. [17], later by Bischof and van Loan [10], and yet later by Schreiber and van Loan [42], resulting in the compact  $WY$  technique for accumulating Householder reflectors.

Elmroth and Gustavson [19–21] generalized this work to produce high performance recursive QR factorization. In this work the problem of reducing the amount of extra floating point operations was addressed by the introduction of *mini blocking/s register blocking*, referred to as *inner blocking* in this article. Other implementations of updating factorizations also rely on this concept [27,40].

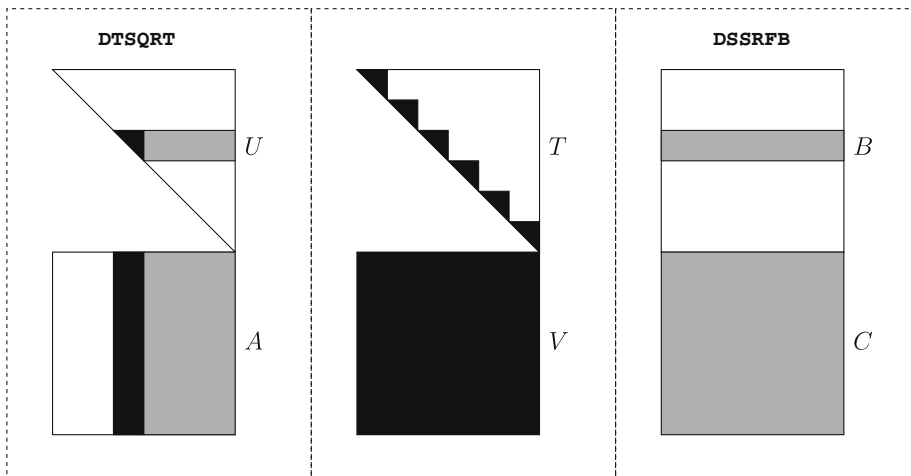
<sup>2</sup> The upper part of  $L_{ij}$  is, actually, a group of  $b/s$  unit, lower triangular matrices each of size  $s \times s$  and, thus, only a small memory area is required to store it.



**Fig. 2.** Graphical representation of one repetition of the outer loop in Algorithms 2 and 3 on a matrix with  $p = q = 3$ . A thick border shows the tiles that are being read and a gray fill shows the tiles that are being written at each step. As expected the picture is very similar to the out-of-core algorithm presented in [27].

Based on the observation that the `DGEQRT`, `DLARFB` and `DTSQRT` kernels only contribute lower order terms (only  $O(n^2)$ ,  $n$  being the size of the problem), the cost of the tiled algorithm for the QR factorization is determined by the cost of the `DSSRFB` kernel. It is, thus, important to pay attention to the way the transformations applied by `DSSRFB` are computed and accumulated in `DTSQRT`. These transformations can be accumulated in sets of  $s$ ; assuming  $s \ll b$ , where  $b$  is the tile size, the extra cost introduced by blocking is limited to a negligible amount, as demonstrated below. This technique is illustrated in Fig. 3.

Assuming  $b/s = t$ , the `DTSQRT`( $U, A, V, T$ ) performs a loop of  $t$  repetitions where, at each step  $i$ , a set of  $s$  Householder reflectors  $V_i = (v_{i1} v_{i2} \dots v_{is})$  are computed and accumulated according to the `WY` technique already mentioned above. As a result



**Fig. 3.** Details of the computation and accumulation of transformations in `DTSQRT` and their application in `DSSRFB`.

of the accumulation, an upper triangular matrix  $T_i$  of size  $s \times s$  is formed ( $V_i$  and  $T_i$  are highlighted in black in Fig. 3(center)). This amounts to a blocking QR factorization (with block size  $s$ ) of the couple formed by the  $U$  and  $A$  tiles (in Fig. 3(left) the panel and the trailing submatrix are highlighted in black and grey, respectively). By the same token, the  $\text{DSSRFB}(B, C, V, T)$  performs a loop of  $t$  repetitions where, at step  $i$ , a portion of the  $B$  and  $C$  tiles is updated by the application of the transformations computed in  $\text{DTSQRT}$  and accumulated in  $V_i$  and  $T_i$ . The data updated at each step of  $\text{DSSRFB}$  is highlighted in grey in Fig. 3 (right).

The cost for a single call of the  $\text{DSSRFB}$  kernel is, ignoring the lower order terms,  $4b^3 + sb^2$ ; consequently, the cost of the whole QR factorization is

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

assuming that  $q < p$  and that  $p$  and  $q$  are big enough so that it is possible to ignore the  $O(n^2)$  contributions from the  $\text{DGEQRT}$ ,  $\text{DLARFB}$  and  $\text{DTSQRT}$  kernels. It must be noted that, when  $s = b$ , the cost of the tiled algorithm is 25% higher than that of the standard LAPACK one; the choice  $s = b$  may help overcoming the limitations of commonly used BLAS libraries on small size data [12] but, as performance results show (see Section 5) it is possible to define values for  $b$  and  $s$  capable of reducing the extra cost to a negligible amount while providing a good level of performance.

This tile level blocking technique can be applied to the  $\text{DTSRFT}$  and  $\text{DSSSM}$  kernel subroutines for the tiled LU factorization as well. This leads to a cost of  $2b^3 + sb^2$  for the  $\text{DSSSM}$  kernel and

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

for the whole factorization under the same assumption as before.

It has to be noted that, too small values for  $s$  may hurt the performance of the Level-3 BLAS operations used in the kernel subroutines. It is, thus, very important to carefully choose the correct values for  $b$  and  $s$  that offer the better compromise between extra cost minimization and efficiency of Level-3 BLAS operations.

### 3.2.4. Stability of the tiled algorithm for the LU factorization

Algorithm 3 performs eliminations with different pivots than *Gaussian elimination with partial pivoting* (GEPP). For eliminating the  $(n-k)$  entries in column  $k$ , partial pivoting chooses a unique pivot while, Algorithm 3 potentially uses up to  $(n-k)/b$  pivots. The pivoting strategy considered in Algorithm 3 is indeed a tiled version of *Gaussian elimination with pairwise pivoting* (GEWP) where GEPP is used at the block level. For this reason, we call the pivoting strategy used by Algorithm 3: *Gaussian elimination with tiled pairwise pivoting* (GETWP). When  $b = 1$  (a  $n$ -by- $n$  tiled matrix with 1-by-1 tiles), GETWP reduces to GEWP. When  $b = n$  (a 1-by-1 tiled matrix with  $n$ -by- $n$  tiles), GETWP reduces to GEPP.

GEWP dates back to Wilkinson's work [47]. Wilkinson's motivation was to cope with limited amount of memory in contemporary computers. The approach has been since successfully used in out-of-core solvers (e.g. [34,40,41,48]) or in the parallel context (see [23, Section 4.2.2], for a summary of references).

The stability analysis of GEPP is not well understood, the accepted idea is that GEPP is *practically stable*. It is only our experience that makes us conjecture that the practical behavior is stable and indeed far different from a few contrived unstable examples [33,45].

Unfortunately and unsurprisingly, the stability analysis of GETWP is as badly understood. In this section, we build experience with this pivoting strategy and conclude that

1. GETWP is less stable than GEPP; the smaller  $b$  is, the less stable the method is;
2. we highly recommend to check the backward error for the solution after a linear solve (i.e. do not trust the answer, check it) and perform a few steps of iterative refinement if needed;
3. our observations have lead to more questions than answers.

GEPP and GETWP consists in the successive applications onto  $A$  of both an elementary unit lower triangular matrix ( $L$ ) and an elementary permutation matrix ( $P$ ). For GEPP, there are  $(n-1)$  couples and we write

$$U = L_{n-1}P_{n-1} \cdots L_1P_1A. \quad (1)$$

For GETWP, with  $p = n/b$ , there are  $(p)(p-1)/2$  couples and we write:

$$U = L_{p,p}P_{p,p}L_{p,p-1}P_{p,p-1} \cdots L_{2,p}P_{2,p} \cdots L_{2,2}P_{2,2}L_{1,p}P_{1,p} \cdots L_{1,1}P_{1,1}A. \quad (2)$$

In GEPP and GETWP, a pivot can eliminate an element only if the eliminator (pivot) is larger in absolute value than the eliminatee. Consequently, the multipliers (the off-diagonal elements of  $L_k$  (GEPP) or  $L_{ij}$  (GETWP)) are smaller or equal than 1 in absolute value.

In the case of GEPP, Eq. (1) can be rearranged in the form:

$$LU = PA, \quad (3)$$



where  $P = P_{n-1} \cdots P_1$  and  $L$  is obtained by taking nonzeros off-diagonal elements in the elementary transformations  $L_k$ , changing their signs, and applying the permutations accordingly. Therefore, in GEPP, all the entries below the diagonal of  $L$  are smaller than 1 in absolute value. Consequently, the norm of  $L$  is bounded independently of  $A$ , we have  $\|L\|_\infty \leq n$ . This observation is crucial in the study of the stability of GEPP and explains the focus in the literature on  $\|U\|_\infty/\|A\|_\infty$ .

In the case of GETWP, we define

$$N = (L_{p,p}P_{p,p}L_{p,p-1}P_{p,p-1}L_{p-1,p-1}P_{p-1,p-1} \cdots L_{2,p}P_{2,p} \cdots L_{2,2}P_{2,2}L_{1,p}P_{1,p} \cdots L_{1,1}P_{1,1})^{-1}, \tag{4}$$

so that we can write from Eq. (2):

$$NU = A. \tag{5}$$

Eq. (5) is to GETWP what Eq. (3) is to GEPP. We note that  $N$  is a mathematical artifact and in practice  $N$  is not computed but manipulated through  $L_{i,k}$  and  $P_{i,k}$ .

Three main differences occur between  $L$  from GEPP and  $N$  from GETWP:

1.  $N$  is the combination of permutations and elementary transformations, the effect of which cannot be dissociated in two matrices  $L$  and  $P$  as it is for GEPP, this complicates notably any analysis,
2. although we need  $n(n-1)/2$  elements to store all the  $(L_{ij})$ 's, the matrix  $N$  is not unit lower triangular and has in general a lot more than  $n(n-1)/2$  entries,
3. the absolute values of the entries of the off-diagonal elements of  $N$  can be greater than 1 and in practice they are notably larger, therefore a stability analysis of GETWP requires us not only to monitor  $\|U\|_\infty$  but also  $\|L\|_\infty$ .

In term of related theoretical work, Sorensen [43] has proved that the worst case behavior for the growth in  $U$  for GETWP is  $2^{n-1}$  (same bound for GEPP) while the worst case behavior for the growth in  $L$  is  $2^{n-1}$  (GEPP is  $\sqrt{n}$ ). We know that these worth case scenarios come from contrived examples and so our present analysis tries to clarify what the general case behavior is.

Experimental results of the stability of GEWP are given in [45] where Trefethen and Schreiber experimentally showed that the growth factor in  $U$  is smaller than  $n$  for a set of random matrices ( $n \leq 1024$ ). Quintana-Ortí and van de Geijn [40] have experimentally studied GETWP on random matrices with two tiles (case  $b = n/2$ ).

The present section details results for GETWP with various block sizes on matrices coming from random matrices and applications.

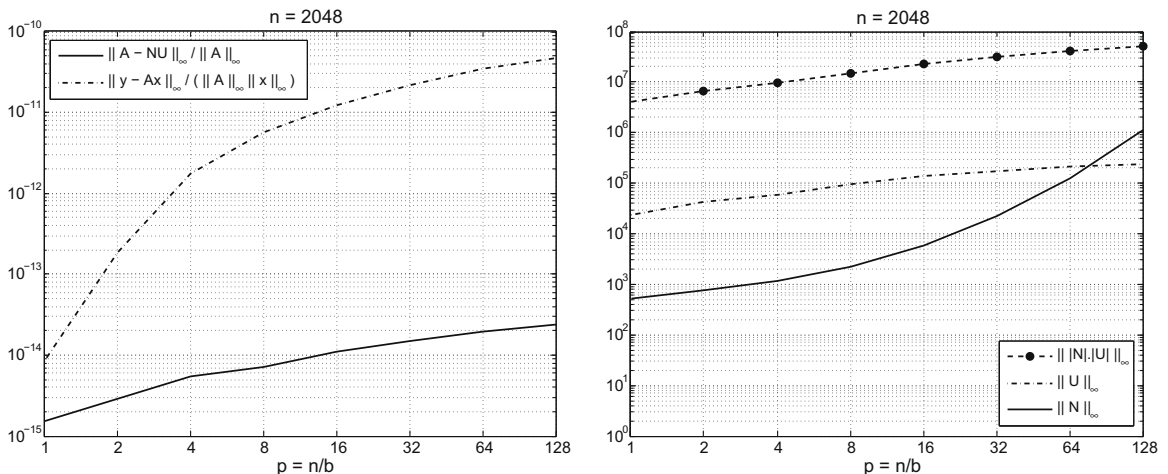
*Random matrices.* We take 10 random matrices of size  $n = 2048$  ( $A = \text{randn}(n)$ ). Any reported quantity is indeed the mean obtained from this sample.

To evaluate the backward error for the factorization of GETWP, we need to compute the  $N$  factor. From Eq. (4), we get

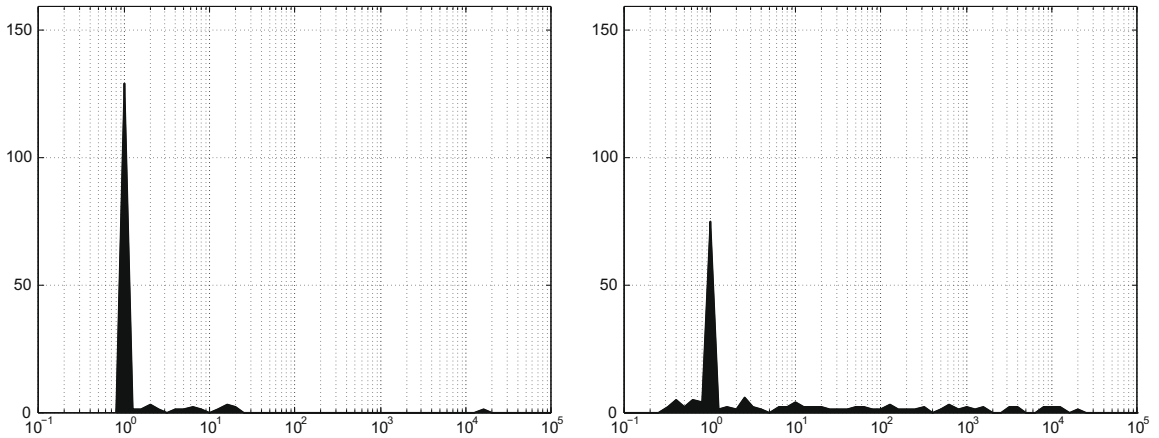
$$N = P_{1,1}^{-1}L_{1,1}^{-1} \cdots P_{1,p}^{-1}L_{1,p}^{-1}P_{2,2}^{-1}L_{2,2}^{-1} \cdots P_{2,p}^{-1}L_{2,p}^{-1} \cdots P_{p-1,p-1}^{-1}L_{p-1,p-1}^{-1}P_{p,p-1}^{-1}L_{p,p-1}^{-1}P_{p,p}^{-1}L_{p,p}^{-1}.$$

On the left of Fig. 4, we plot the backward error for the factorization obtained with GETWP

$$\left( \frac{\|A - N_{wp}U_{wp}\|_\infty}{\|A\|_\infty} \right)$$



**Fig. 4.** We take a sample of 10 random matrices of size  $n = 2048$ . On the left, we plot the mean backward error obtained for the GETPW factorization and the mean backward error for the solution when solving a linear system of equations with the GETPW factorization with a random right-hand side. In the x-axis, we make the tile size ( $b$ ) decrease from  $b = n = 2048$  (corresponds to  $p = 1$ ) to  $b = 16$  (corresponds to  $p = 128$ ). On the right, we plot the mean of various relevant quantities for the GETPW factorization.



**Fig. 5.** Histogram representing the distribution of the ratio  $\frac{\|y - Ax_{wp}\|_\infty / (\|A\|_\infty \|x_{wp}\|_\infty)}{\|y - Ax_{pp}\|_\infty / (\|A\|_\infty \|x_{pp}\|_\infty)}$  for GETWP over  $\frac{\|y - Ax_{pp}\|_\infty / (\|A\|_\infty \|x_{pp}\|_\infty)}{\|y - Ax_{wp}\|_\infty / (\|A\|_\infty \|x_{wp}\|_\infty)}$  for GEPP (left) and the ratio of  $\frac{\|A - N_{wp}U_{wp}\|_\infty / \|A\|_\infty}{\|PA - L_{pp}U_{pp}\|_\infty / \|A\|_\infty}$  for GETWP over  $\frac{\|PA - L_{pp}U_{pp}\|_\infty / \|A\|_\infty}{\|A - N_{wp}U_{wp}\|_\infty / \|A\|_\infty}$  for GEPP (right). Matrices are taken from the matrix-market collection [3] and right-hand sides are random.

and the backward error for the solution when solving a linear system of equations with the GETPW factorization and a random right-hand side

$$\left( \frac{\|y - Ax_{wp}\|_\infty}{\|A\|_\infty \|x_{wp}\|_\infty} \right).$$

The horizontal axis represents various numbers of tiles ( $p$ ). For  $p = 1$ , there is one tile so the algorithm is indeed GEPP. For  $p = 2$ , there are four 1024-by-1024 tiles, etc. As the number of tiles increases, the stability of GETWP decreases. We note that there is a significant difference between the backward error for the solution and the backward error for the factorization.

On the right of Fig. 4, we plot the three quantities:

$$\|N_{wp}\|_\infty, \quad \|U_{wp}\|_\infty, \quad \text{and} \quad \|N_{wp}\|_\infty \cdot \|U_{wp}\|_\infty.$$

The relevant quantity for the stability of the factorization being  $\|N_{wp}\|_\infty \cdot \|U_{wp}\|_\infty$ .  $\|N_{wp}\|_\infty$  and  $\|U_{wp}\|_\infty$  being good indicators of how large this first quantity might be. We observe that the growth in  $U_{wp}$  ( $\|U_{wp}\|_\infty$ ) is almost constant as we increase the number of tiles, unfortunately the growth in  $N_{wp}$  ( $\|N_{wp}\|_\infty$ ) is increasing quite significantly with  $p$ . We note however that  $\|N_{wp}\|_\infty \cdot \|U_{wp}\|_\infty$  is significantly smaller than  $\|N_{wp}\|_\infty \|U_{wp}\|_\infty$  which means that, hopefully, all the growth observed in  $N$  does not end up in the error in the factorization. We acknowledge that the mechanism behind this observation is not yet understood.

We report a last experiment that is worth noting. Since we are working with random matrices, a reasonable pivoting strategy to consider is *Gaussian elimination with no pivoting* (GENP). In this context, we would hope that GETWP is at least better than GENP. It turns out that this is not the case for the backward error for the factorization. We report for GENP a mean error of  $2 \times 10^{-11}$  while the mean is  $7 \times 10^{-11}$  for GETWP and  $p = 128$ . Once more, we acknowledge that the mechanism behind this observation is not yet understood.

*Matrix market matrices.* In Fig. 5, we present stability results for GETWP compared to GEPP on matrices from Matrix Market [3]. At the date of May 2008, we took all the matrices from Matrix Market with size ( $n$ ) between 1 and 6000 which are square, which are associated with *Linear System*, and which are in Matrix Market format (.mtx.gz).<sup>3</sup> This methodology provides us 159 matrices. For all these matrices, we assign a random right-hand side  $y$  ( $y = \text{randn}(n, 1)$ ;) whether or not the matrix had a prescribed right-hand side on Matrix Market. The tile size  $b$  is a function of the matrix size  $n$ . In this experiment, we want to keep  $p = n/b$  constant with  $p = 32$ .

We note that some of these matrices will provide us with  $U$  factors that have exact 0's on their diagonals. This will result in NaN or Inf results.

On the left in Fig. 5, we give the histogram of the ratio of the backward error for the solution when solving a linear system of equations

$$\left( \frac{\|y - Ax_{wp}\|_\infty}{\|A\|_\infty \|x_{wp}\|_\infty} \right) / \left( \frac{\|y - Ax_{pp}\|_\infty}{\|A\|_\infty \|x_{pp}\|_\infty} \right).$$

On the right in Fig. 5, we give the histogram of the ratio of the backward error for the factorization

$$\left( \frac{\|A - N_{wp}U_{wp}\|_\infty}{\|A\|_\infty} \right) / \left( \frac{\|PA - L_{pp}U_{pp}\|_\infty}{\|A\|_\infty} \right).$$

<sup>3</sup> This last restriction implies that we have discarded the five matrices that were in Harwell Boeing format (.pse.gz).

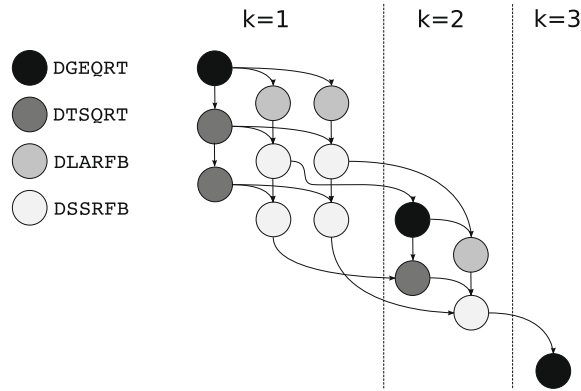


Fig. 6. The dependency graph of Algorithm 2 on a matrix with  $p = q = 3$ .

We have set any backward error (for the solution or for the factorization) smaller than the machine precision at the level of the machine precision.

For 146 matrices out of 147,<sup>4</sup> we solve the linear system with a backward error for the solution lower than the one of GEPP times 25 (Fig. 5 left).

For 121 matrices out of 147, we obtain a backward error for the factorization lower than the one of GEPP times 25 (Fig. 5 right).

The worst case matrix is in both case the matrix named `orani`. Its condition number is about  $10^4$  and its order is  $n = 2529$ . The ratio of backward error is  $4.6 \times 10^7$  for the factorization and  $1.9 \times 10^4$  for the solution. If we compare the norm of the factors, we get:

$$\|N_{wp}\|_{\infty} = 2 \times 10^5, \quad \|U_{wp}\|_{\infty} = 6 \times 10^4, \quad \text{and} \quad \|L_{pp}\|_{\infty} = 20, \quad \|U_{pp}\|_{\infty} = 10,$$

where we initially had  $\|A\|_{\infty} = 9$ . We see that, for this special case, GETWP suffers of growth in the  $N$  factor and growth in the  $U$  factor.

#### 4. Graph driven asynchronous execution

Following the approach presented in [11,12,36], Algorithms 1–3 can be represented as a Directed Acyclic Graph (DAG) where nodes are computational tasks performed in kernel subroutines and where edges represent the dependencies among them. Fig. 6 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.

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; this non conventional definition of critical path stems from the observation that anticipating the execution of nodes with an higher number of outgoing edges maximises the number of tasks in a “ready” state. 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. Fig. 7 shows part of the execution flow of Algorithm 2 using 8 cores machine when tasks are dynamically scheduled 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. 7 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.

##### 4.1. Implementation details

The approach based on the combination of tiled algorithms, Block Data Layout and graph driven, dynamic execution (as described, respectively, in Sections 3 and 4) has been validated in a software implementation based on the pThreads POSIX standard. The graph of dependencies is implicitly represented in a shared progress table. Each thread in the pool is self-

<sup>4</sup> Twelve matrices are indeed structurally singular and produce a 0 on the diagonal of the  $U$  factor for both GEPP and GETWP.

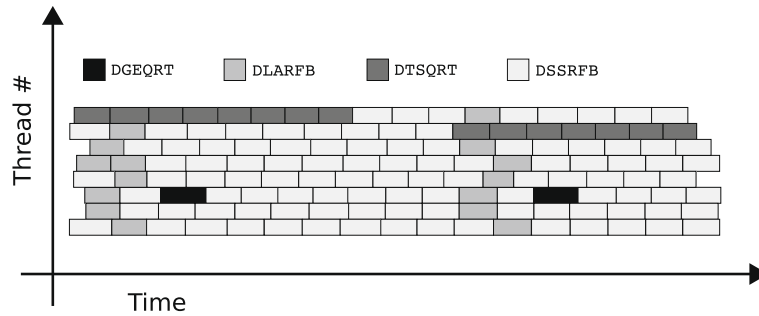


Fig. 7. The execution flow for dynamic scheduling, out of order execution of Algorithm 2.

Table 1

Details of the system used for the following performance results.

	8-way dual Opteron
Architecture	AMD® Opteron® 8214
Clock speed	2.2 GHz
# Cores	$2 \times 8 = 16$
Peak performance	70.4 Gflop/s
Memory	65 GB
Compiler suite	Intel 9.1
BLAS libraries	MKL-9.1.023, ACML-4.0.0
DGEMM performance	57.5 Gflop/s

scheduled: it check the shared progress table to identify a set of doable tasks and then picks one of them according to a priority policy. Once a thread terminates the execution of a task, it updates the progress table accordingly. Because the centralized progress table may represent a bottleneck and may imply more synchronization as the degree of parallelism grows, the object of future work will be to distribute the handling of the dependency graph.

Despite the choice of using the pThreads standard, the presented approach may also be implemented by means of other technologies like, for examples OpenMP or MPI or even an hybrid combination of them.

## 5. Performance results

The performance of the tiled algorithms for Cholesky, QR and LU factorizations with dynamic scheduling of tasks (using ACML-4.0.0 BLAS for tile computations) has been measured on the system described in Table 1 and compared to the performance of the MKL-9.1 and ACML-4.0.0 implementations and to the fork-join approach, i.e., the standard algorithm for block factorizations of LAPACK associated with multithreaded BLAS (ACML-4.0.0).<sup>5</sup> In the following figures, the tiled algorithms with dynamic scheduling are referred to as PLASMA (Parallel Linear Algebra for Scalable Multicore Architectures), the name of the project inside which the presented work was developed.

Figs. 8–10 report the performance of the Cholesky, QR and LU factorizations for the tiled algorithms with dynamic scheduling, the MKL-9.1 and ACML-4.0.0 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<sup>6</sup> has been tuned in order to achieve the best performance; specifically the block size was set to 100. These graphs show the performance measured using the maximum number of cores available on the system (i.e., 16) with respect to the problem size. The axis of ordinates has been scaled to reflect the theoretical peak performance of the system (i.e. the top value is 70.4 Gflop/s) and, also, as a reference, the performance of the matrix–matrix multiply (DGEMM) has been reported.

Fig. 11 shows the weak scalability, i.e. the flop rates versus the number of cores when the local problem size is kept constant ( $n_{loc} = 5000$ ) as the number of cores increases.

In order to reflect the time to completion, in all the figures the operation count of the tiled algorithms for QR and LU factorizations is assumed to be the same as that of the LAPACK block algorithm; for what discussed in Section 3.2.3, this assumption is only slightly inaccurate since the amount of extra flops can be considered negligible for a correct choice of the internal blocking size  $s$ .

<sup>5</sup> For both tiled algorithms and LAPACK, the choice of the underlying BLAS library is such that the highest performance possible is achieved.

<sup>6</sup> The block size in the LAPACK algorithm sets the width of the panel.

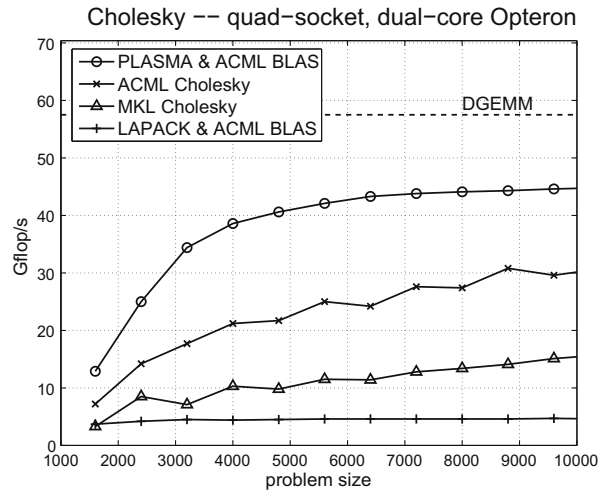


Fig. 8. Cholesky factorization.

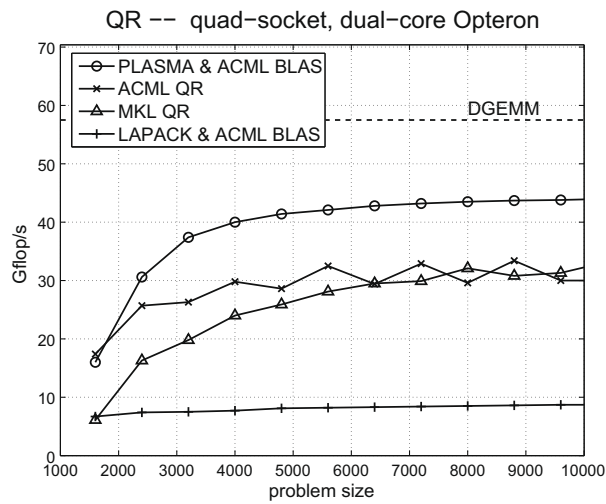


Fig. 9. QR factorization.

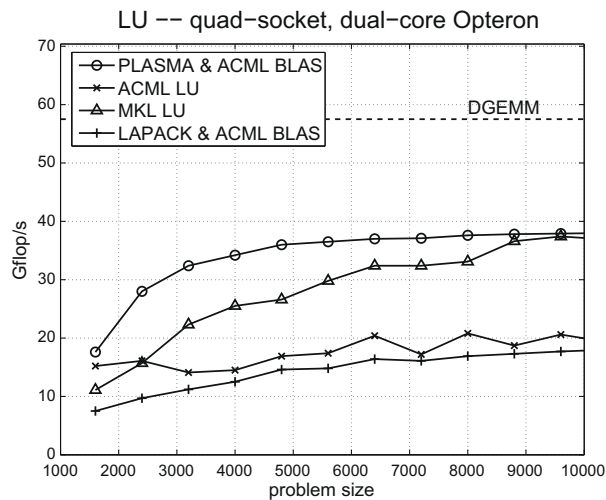


Fig. 10. LU factorization.

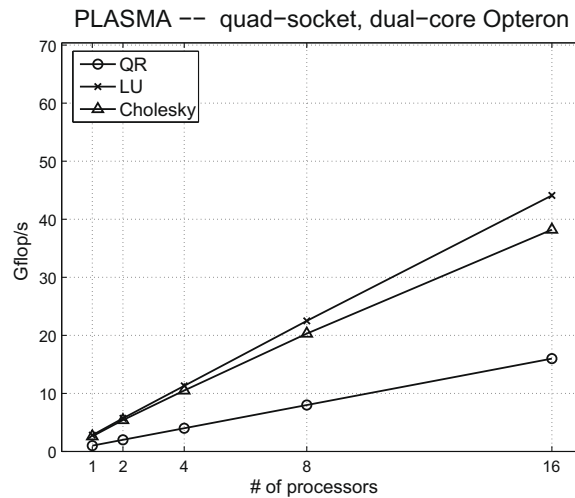


Fig. 11. PLASMA software: scalability.

Figs. 8 and 9 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 performs around 50% better than a vendor implementation of the operation. An important remark has to be made for the Cholesky factorization: the *left-looking* variant (see [18] 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 delivers, asymptotically, roughly the same performance as the MKL-9.1 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 in Section 3.2.3, 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.

It must be noted, however, that the tiled algorithm for LU factorization, reaches the asymptotic performance faster thus providing considerable performance benefit for lower size problems. This is a consequence of the fact that, once the values for the tile size  $b$  and blocking factor  $s$  are fixed, the performance of the BLAS operations is constant. The dynamic execution model reduces the overhead of parallelization yielding the relatively steep growth for the curve related to the tiled algorithm.

## 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.

The work presented here follows a path established by the same authors in [11,12,36,37] exploiting and reinterpreting ideas already studied in the past [6,7,16,38].

The discussed approach suggests 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.

## Acknowledgement

The authors would like to thank Fred Gustavson for his insightful comments, which helped significantly improve the quality of this publication.

## References

- [1] <http://top500.org>.
- [2] <http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm>.
- [3] <http://math.nist.gov/MatrixMarket/>.
- [4] Teraflops research chip. <http://www.intel.com/research/platform/terascale/teraflops.htm>.
- [5] IBM engineering and scientific subroutine library for AIX version 3, release 3., December 2001. IBM Pub. No. SA22-7272-04.
- [6] R.C. Agarwal, F.G. Gustavson, 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 Asynchronous Parallel Processors*, Stanford CA, 22–26 Aug, 1988, North Holland, pp. 217–221.
- [7] R.C. Agarwal, F.G. Gustavson, Vector and parallel algorithms for Cholesky factorization on IBM 3090, in: *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pp. 225–233, ACM, New York, NY, USA, 1989.
- [8] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, *LAPACK Users' Guide*, 3rd ed., SIAM, Philadelphia, 1999.
- [9] M.W. Berry, J.J. Dongarra, Y. Kim, A parallel algorithm for the reduction of a nonsymmetric matrix to block upper-Hessenberg form, *Parallel Comput.* 21 (8) (1995) 1189–1211.
- [10] C. Bischof, C. van Loan, The WY representation for products of householder matrices, *J. Sci. Stat. Comput.* 8 (1987) 2–13.
- [11] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, S. Tomov, The impact of multicore on Math software, in: B. Kågström et al. (Eds.), *State of the Art in Scientific Computing. 8th International Workshop, PARA 2006*, Umea, Sweden, June 18–21, 2006, volume 4699 of *Lecture Notes in Computer Science*, pp. 1–10, 2007.
- [12] A. Buttari, J. Langou, J. Kurzak, J.J. Dongarra, Parallel tiled QR factorization for multicore architectures, *Concurrency Computat.: Pract. Exper.* 20 (13) (2008) 1573–1590.
- [13] E. Chan, E.S. Quintana-Ortí, G. Quintana-Ortí, R. van de Geijn, Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures, in: *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, ACM Press, New York, NY, USA, 2007, pp. 116–125.
- [14] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, 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).
- [15] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D.W. Walker, R.C. 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*, Springer-Verlag, London, UK, 1996, pages 107–114.
- [16] J.J. Dongarra, R.E. Hiromoto, A collection of parallel linear equations routines for the Denelcor HEP, *Parallel Computing* 1 (2) (1984) 133–142. December.
- [17] J.J. Dongarra, L. Kaufman, S. Hammarling, Squeezing the most out of eigenvalue solvers on high-performance computers, *Lin. Alg. Applic.* 77 (1986) 113–136.
- [18] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, Henk A. van der Vorst, *Numerical Linear Algebra for High Performance Computers*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.
- [19] E. Elmroth, F.G. Gustavson, New serial and parallel recursive QR factorization algorithms for SMP systems, in: *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, 4th International Workshop, PARA'98, Umeå, Sweden, June 14–17 1998, *Lecture Notes in Computer Science* 1541, 120–128.
- [20] E. Elmroth, F.G. Gustavson, Applying recursion to serial and parallel QR factorization leads to better performance, *IBM J. Res. & Dev.* 44 (4) (2000) 605–624.
- [21] E. Elmroth, F.G. Gustavson, High-performance library software for QR factorization, in: *Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, 5th International Workshop, PARA 2000, Bergen, Norway, June 18–20 2000. *Lecture Notes in Computer Science* (1947) 53–63.
- [22] E. Elmroth, F.G. Gustavson, I. Jonsson, B. Kågström, Recursive blocked algorithms and hybrid data structures for dense matrix library software, *SIAM Review* 46 (1) (2004) 3–45.
- [23] K.A. Gallivan, R.J. Plemmons, A.H. Sameh, Parallel algorithms for dense linear algebra computations, *SIAM Review* 32 (1) (1990) 54–135. March.
- [24] P.E. Gill, G.H. Golub, W.A. Murray, M.A. Saunders, Methods for modifying matrix factorizations, *Mathematics of Computation* 28 (126) (1974) 505–535.
- [25] G. Golub, C. Van Loan, *Matrix Computations*, 3rd ed., Johns Hopkins University Press, Baltimore, MD, 1996.
- [26] K. Goto, 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.
- [27] B.C. Gunter, R.A. van de Geijn, Parallel out-of-core computation and updating of the QR factorization, *ACM Trans. Math. Softw.* 31 (1) (2005) 60–78.
- [28] F. Gustavson, L. Karlsson, B. Kågström, Three algorithms for Cholesky factorization on distributed memory using packed storage, in: B. Kågström et al. (Eds.), *State of the Art in Scientific Computing. 8th International Workshop, PARA 2006*, Umea, Sweden, June 18–21, 2006, volume 4699 of *Lecture Notes in Computer Science*, 2007, pp. 550–559.
- [29] F.G. Gustavson, Recursion leads to automatic variable blocking for dense linear-algebra algorithms, *IBM J. Res. & Dev.* 41 (6) (1997) 737–756.
- [30] F.G. Gustavson, New generalized matrix data structures lead to a variety of high-performance algorithms, in: *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, Kluwer Academic Publishers, Ottawa, Canada, October 2–4 2000, pp. 211–234.
- [31] F.G. Gustavson, High-performance linear algebra algorithms using new generalized data structures for matrices, *IBM J. Res. Dev.* 47 (1) (2003) 31–55.
- [32] F.G. Gustavson, J.A. Gunnels, J.C. Sexton, Minimal data copy for dense linear algebra factorization, in: *Applied Parallel Computing, State of the Art in Scientific Computing, 8th International Workshop, PARA 2006*, Umeå, Sweden, June 18–21, 2006. *Lecture Notes in Computer Science* 4699, pp. 540–549. [http://dx.doi.org/10.1007/978-3-540-75755-9\\_66](http://dx.doi.org/10.1007/978-3-540-75755-9_66). doi:10.1007/978-3-540-75755-9\_66.
- [33] N.J. Higham, D.J. Higham, Large growth factors in Gaussian elimination with pivoting, *SIAM J. Matrix Anal. Appl.* 10 (2) (1989) 155–164. April.
- [34] Thierry Joffrain, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Rapid development of high-performance out-of-core solvers, in: Jack Dongarra, Kaj Madsen, Jerzy Wasniewski (Eds.), *PARA*, volume 3732 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 413–422.
- [35] J. Kurzak, A. Buttari, J.J. Dongarra, Solving systems of linear equation on the CELL processor using Cholesky factorization, *Trans. Parallel Distrib. Syst.* 19 (9) (2008) 1175–1186.
- [36] J. Kurzak, 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.
- [37] J. Kurzak, J.J. Dongarra, QR factorization for the CELL processor, *Scientific Programming*, special issue on High Performance Computing on CELL B.E. Processors. (accepted).
- [38] R.E. Lord, J.S. Kowalik, S.P. Kumar, Solving linear algebraic equations on an MIMD computer, *J. ACM* 30 (1) (1983) 103–117.
- [39] 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. Suzuki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, K. Yazawa, The design and implementation of a first-generation CELL processor, in: *IEEE International Solid-State Circuits Conference*, 2005, pp. 184–185.
- [40] E. Quintana-Ortí, 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.
- [41] J.K. Reid, A note on the stability of Gaussian elimination, *IMA Journal of Applied Mathematics* 8 (3) (1971) 374–375.
- [42] R. Schreiber, C. van Loan, A storage-efficient WY representation for products of householder transformations, *J. Sci. Stat. Comput.* 10 (1991) 53–57.
- [43] D.C. Sorensen, Analysis of pairwise pivoting in Gaussian elimination, *IEEE Trans. Comput.* 34 (3) (1985) 274–278.

- [44] G.W. Stewart, Matrix Algorithms, 1st ed., vol. 1, SIAM, Philadelphia, 1998.
- [45] L.N. Trefethen, R.S. Schreiber, Average-case stability of Gaussian elimination, *SIAM J. Matrix Anal. Appl.* 11 (3) (1990) 335–480. July.
- [46] R.C. Whaley, A. Petitet, J. Dongarra, Automated empirical optimization of software and the ATLAS project, *Parallel Computing* 27 (1–2) (2001) 3–25.
- [47] J.H. Wilkinson, The Algebraic Eigenvalue Problem, Oxford University Press, 1965.
- [48] E.L. Yip, Fortran subroutines for out-of-core solutions of large complex linear systems. Technical Report CR-159142, NASA, November 1979.