Parallel Tiled QR Factorization for Multicore Architectures

Alfredo Buttari[1], Julien Langou[3], Jakub Kurzak[1], Jack Dongarra[12]

[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] 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 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 algorithm for QR factorization where parallelism can only be exploited at the level of the BLAS operations.

1 Introduction

In the last twenty years, microprocessor manufacturers have been driven towards higher performance rates only by the exploitation of higher degrees of *Instruction Level Parallelism* (ILP). Based on this approach, several generations of processors have been built where clock frequencies were higher and higher and pipelines were deeper and deeper. As a result, applications could benefit from these innovations and achieve higher performance simply by relying on compilers that could efficiently exploit ILP. Due to a number of physical limitations (mostly power consumption and heat dissipation) this approach cannot be pushed any further. For this reason, chip designers have moved their focus from ILP to *Thread Level Parallelism* (TLP) where higher performance can be achieved by replicating execution units (or *cores*) on the die while keeping the clock rates in a range where power consumption and heat dissipation do not represent a problem. 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 released in June 2007 shows that the number of 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 **?** architecture or the Intel Polaris**?** prototype). This requires splitting an operation into tasks that operate on small portions of data in order to reduce bus traffic and improve data locality.

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

The LAPACK **?** and ScaLAPACK **?** 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 **?** and GotoBLAS **?** or other vendor BLAS like Intel MKL **?** are popular choices. These parallel BLAS libraries use common techniques for shared memory parallelization like pThreads **?** or OpenMP **?**. This is represented in Figure 1 (*left*).

In the ScaLAPACK case, parallelism is exploited by PBLAS **?** which is a parallel BLAS implementation that uses the Message Passing Interface **?** (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

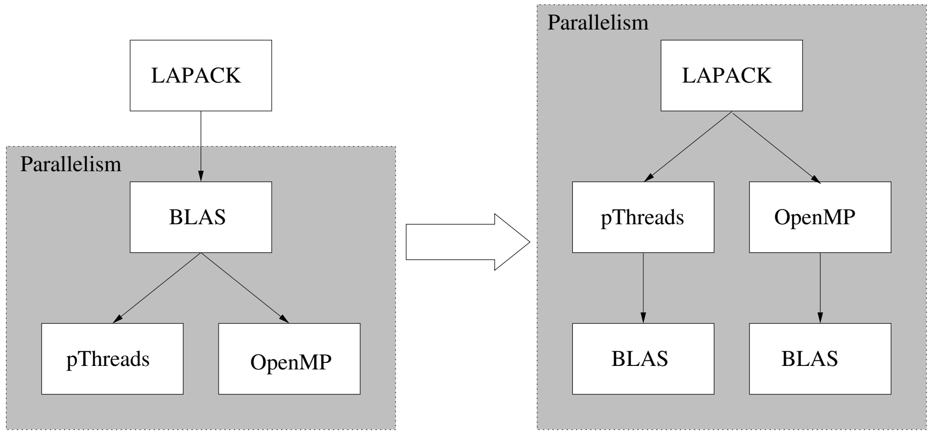

Fig. 1. Transition from sequential algorithms that rely on parallel BLAS to parallel algorithms.

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 **???** 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 **?**.

The rest of this document shows how this can be achieved for the QR factorization. Section 2 describes the algorithm for block QR factorization used in the LAPACK library; Section 3 describes the tiled QR factorization that provides both fine granularity and high level of asynchronicity; performance results for this algorithm are shown in Section 4. Finally future working directions are illustrated in Section 5.

2 Block QR Factorization

2.1 Description of the block QR Factorization

The QR factorization is a transformation that factorizes an $m \times n$ matrix A into its factors Q and R where Q is a unitary matrix of size $n \times n$ and R is a triangular matrix of size $m \times m$. This factorization is operated by applying $min(m,n)$ Householder reflections to the matrix A. Since Householder reflections are orthogonal transformations, this factorization is stable as opposed to the LU one; however, stability comes at the price of a higher flop count: QR requires $2n^2(m-n/3)$ as opposed to the $n^2(m-n/3)$ needed for LU. A detailed discussion of the QR factorization can be found in **???**. LAPACK uses a particular version of this algorithm which achieves higher performance on architectures with memory hierarchies thanks to blocking. This algorithm is based on accumulating a number of Householder transformations in what is called a *panel factorization* which are, then, applied all at once by means of high performance Level 3 BLAS operations. The technique used to accumulate Householder transformation was introduced in **?** and is known as the compact WY technique.

The LAPACK subroutine that performs the QR factorization is called `DGEQRF` and is explained below. Consider a matrix A of size $m \times n$ that can be represented as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

where A_{11} is of size $b \times b$, A_{12} of size $b \times (n-b)$, A_{21} of size $(m-b) \times b$ and A_{22} of size $(m-b) \times (n-b)$.

The LAPACK algorithm for QR factorization can be described as a sequence of steps where, at each step, the transformation in Equation (1) is performed.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \Longrightarrow \begin{pmatrix} V_{11} \\ V_{21} \end{pmatrix}, \begin{pmatrix} R_{11} & R_{12} \\ 0 & \tilde{A}_{22} \end{pmatrix} \tag{1}$$

The transformation in Equation (1) is obtained in two steps:

1. **Panel Factorization.** At this step a QR transformation of the panel (A_{*1}) is performed as in Equation (2).

$$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} \Longrightarrow \begin{pmatrix} V_{11} \\ V_{21} \end{pmatrix}, (T_{11}), (R_{11}) \tag{2}$$

   This operation produces b Householder reflectors $(V_{*,1})$ and an upper triangular matrix R_{11} of size $b \times b$, which is a portion of the final R factor, by means of the `DGEQR2` LAPACK subroutine; also, at this step, a triangular matrix T_{11} of size $b \times b$ by means of the `DLARFT` LAPACK subroutine[4]. Please note that V_{11} is a unit lower triangular matrix of size $b \times b$. The arrays V_{*1} and R_{11} do not need extra space to be stored since they overwrite A_{*1}. A temporary workspace is needed to store T_{11}.
2. **Trailing submatrix update.** At this step, the transformation that was computed in the panel factorization is applied to the rest of the matrix, also called *trailing submatrix* as shown in Equation (3).

$$\begin{pmatrix} R_{12} \\ \tilde{A}_{22} \end{pmatrix} = \left(I - \begin{pmatrix} V_{11} \\ V_{21} \end{pmatrix} \cdot (T_{11}) \cdot \left(V_{11}^T \; V_{21}^T \right) \right) \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} \tag{3}$$

   This operation, performed by means of the `DLARFB` LAPACK subroutine, produces a portion R_{12} of the final R factor of size $b \times (n-b)$ and the matrix $\tilde{A}_{22}$.

[4] for the meaning of the matrix T_{11} please refer to **?**

The QR factorization is continued by applying the transformation (1) to the submatrix $\tilde{A}_{22}$ and, then, iteratively, until the end of the matrix A is reached. The value of $b \ll m, n$ (the so called *block size*) is set by default to 32 in LAPACK-3.1.1 but different values may be more appropriate, and provide higher performance, depending on the architecture characteristics.

2.2 Scalability limits of the LAPACK implementation

The LAPACK algorithm for QR factorization 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 Equation (2), i.e., the panel factorization. Both the `DGEQR2` and the `DLARFT` are 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 $O(n^2)$ floating-point operations are performed on $O(n^2)$ 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 algorithm for QR factorization is, thus, characterized by the presence of a sequential operation (i.e., the panel factorization) 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 the block QR factorization 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 the QR factorization would show a sequence of sequential operations (i.e. the panel factorizations) interleaved to parallel ones (i.e., the trailing submatrix updates).

Table 1 shows the scalability limits of the panel factorization and how this affects the scalability of the whole QR factorization on an 8-way dual-core AMD Opteron system with MKL-9.1 and GotoBLAS-1.15 parallel BLAS libraries.

In **??**, a solution to this scalability problem is presented. The approach described in **??** 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

	Intel MKL-9.1		GotoBLAS-1.15	
# cores	`DGEQR2` Gflop/s	`DGEQRF` Gflop/s	`DGEQR2` Gflop/s	`DGEQRF` Gflop/s
1	0.4106	2.9	0.4549	3.31
2	0.4105	4.95	0.4558	5.51
4	0.4105	8.79	0.4557	9.69
8	0.4109	14.3	0.4549	10.58
16	0.4103	16.89	0.4558	13.01

Table 1. Scalability of the fork-join parallelization on a 8-way dual Opteron system (sixteen cores total).

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 **?**, 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 an algorithm for QR factorization where finer granularity of the operations and higher flexibility for the scheduling can be achieved.

3 Tiled QR Factorization

The idea of dynamic scheduling and out of order execution can be applied to a class of algorithms that allow the parallelization of common Linear Algebra operations. Previous work in this direction includes SYRK, CHOL, block LU, and block QR **???**. For those four factorizations, no algorithmic change is needed however, while CHOL and SYRK can be naturally "tiled", the algorithms for block LU and block QR factorizations involved tall and skinny panel factorization that represents the bottlenecks of the computation (see pervious Section). In order to have a finer granularity in LU and QR, we need to "tile" the operations. To do so we will need a major algorithmic change in LU and QR.

The algorithmic change we propose is actually well-known and takes its roots in updating factorizations **??**. Using updating techniques to tile the algorithms have first[5] been proposed by Yip **?** for LU to improve the efficiency of out-of-core solvers, and were recently reintroduced in **??** for LU and QR, once more in the out-of-core context. A similar idea has also been proposed in **?** for Hessenberg reduction in the parallel distributed context.

The originality of this paper is to study this techniques in the multicore context, where they enable us to schedule operations of very fine granularity.

[5] to our knowledge

3.1 A Fine-Grain Algorithm for QR Factorization

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

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

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

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

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

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

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

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

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

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

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

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

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 1.

Algorithm 1 The block algorithm for QR factorization.

```
1: for $k = 1, 2..., min(p, q)$ do
2:    DGEQT2($A_{kk}$, $T_{kk}$);
3:    for $j = k + 1, k + 2, ..., q$ do
4:       DLARFB($A_{kj}$, $V_{kk}$, $T_{kk}$);
5:    end for
6:    for $i = k + 1, k + 1, ..., p$ do
7:       DTSQT2($R_{kk}$, $A_{ik}$, $T_{ik}$);
8:       for $j = k + 1, k + 2, ..., q$ do
9:          DSSRFB($A_{kj}$, $A_{ij}$, $V_{ik}$, $T_{ik}$);
10:      end for
11:   end for
12: end for
```

Figure 2 gives a graphical representation of one repetition (with $k = 1$) of the outer loop in Algorithm 1 with $p = q = 3$. The red, thick borders show what blocks in the matrix are being read and the light blue fill shows what blocks are being written in a step. The T_{kk} matrices are not shown in this figure for clarity purposes.

3.2 Operation count

This section shows that Algorithm 1 has a higher operation count than the LAPACK algorithm discussed in Section 2. Performance results in Section 4 will demonstrate that it is worth paying this cost for the sake of scaling. The operation count of the block algorithm for QR factorization can be derived starting from the operation count of each elementary operation; assuming that b is the block size:

`DGEQT2`**:** this operation is a standard non blocked QR factorization of a $b \times b$ matrix where, in addition, the T_{kk} triangular matrix is computed. Thus, $4/3b^3$ floating point operations are performed for the factorization plus $2/3b^3$ for computing T_{kk}. This subroutine accounts for $2b^3$ floating point operations total.

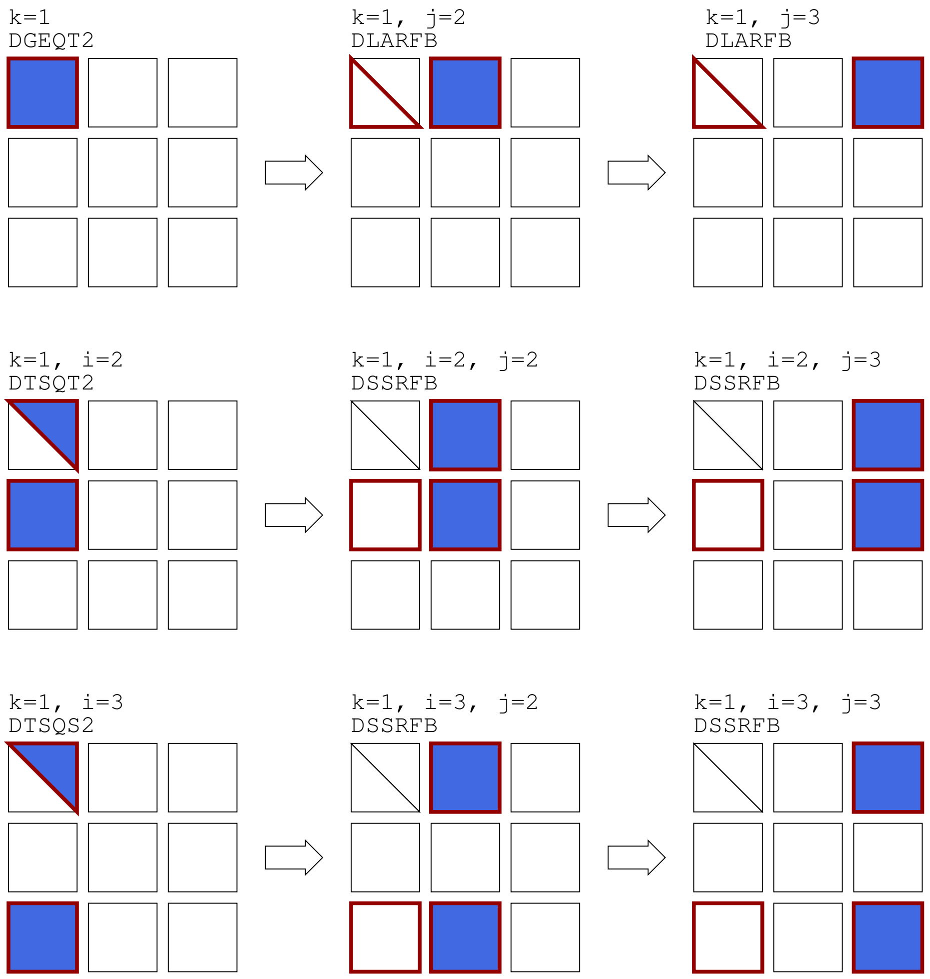

Fig. 2. Graphical representation of one repetition of the outer loop in Algorithm 1 on a matrix with $p = q = 3$. As expected the picture is very similar to the out-of-core algorithm presented in **?**.

`DLARFB`: since both V_{kk} and T_{kk} are triangular matrices, $3b^3$ floating point operations are done in this subroutine.

`DTSQT2`: taking advantage of the triangular structure of R_{kk}, the factorization can be computed in $2b^3$ floating point operations. The computation of the triangular T_{ik} matrix can also be performed exploiting the structure of the Householder vectors built during the factorization (remember that the b reflectors can be represented as an identity block on top of a square full block). Since $4/3b^3$ are needed to compute T_{ik}, the DTSQT2 accounts for $10/3b^3$ floating point operations.

DSSRFB: exploiting the structure of the Householder reflectors and of the T_{ik} matrix computed in DTSQT2, this subroutine needs $5b^3$ floating point operations.

For each repetition of the outer loop in Algorithm 1, one DGEQT2, $q-k$ DLARFB, $p-k$ DTSQT2 and $(p-k)(q-k)$ DSSRFB are performed for a total of $2b^3+3(q-k)b^3+10/3(p-k)b^3+5(p-k)(q-k)b^3$. Assuming, without loss of generality, that $q<p$ and integrating this quantity over all the q repetitions of the outer loop in Algorithm 1, the total operation count for the QR factorization is

$$\begin{aligned} &\textstyle\sum_{k=1}^{q}(2b^3+3(q-k)b^3+\frac{10}{3}(p-k)b^3+5(p-k)(q-k)b^3) \\ &\simeq \tfrac{5}{2}q^2(p-\tfrac{q}{3})b^3 \qquad (4) \\ &= \tfrac{5}{2}n^2(m-\tfrac{n}{3}). \end{aligned}$$

Equation (4) shows that the block algorithm for QR factorization needs 25% more floating point operations than the standard LAPACK algorithm.

Note that if we have used the nonblocked version of the Householder application (DLARF instead of DLARFB) then the number of FLOPS for the tiled algorithm and the block algorithm would have been exactly the same.

An easy twist to reduce the 25% overhead of the block version is to use nonsquare blocks. For example, using $2b\times b$ blocks would reduce the overhead at 12.5%.

3.3 Graph driven asynchronous execution

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

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

This dynamic scheduling results in an out of order execution where idle time is almost completely eliminated since only very loose synchronization is required

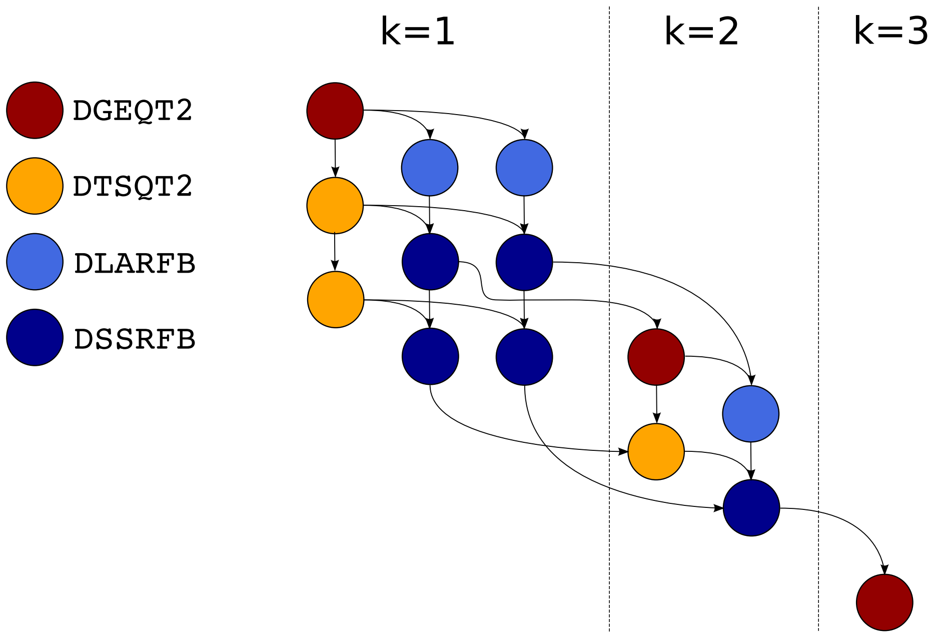

Fig. 3. The dependency graph of Algorithm 1 on a matrix with $p = q = 3$.

between the threads. Figure 4 shows part of the execution flow of Algorithm 1 on a 16-cores machine (8-way Dual Opteron) 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.

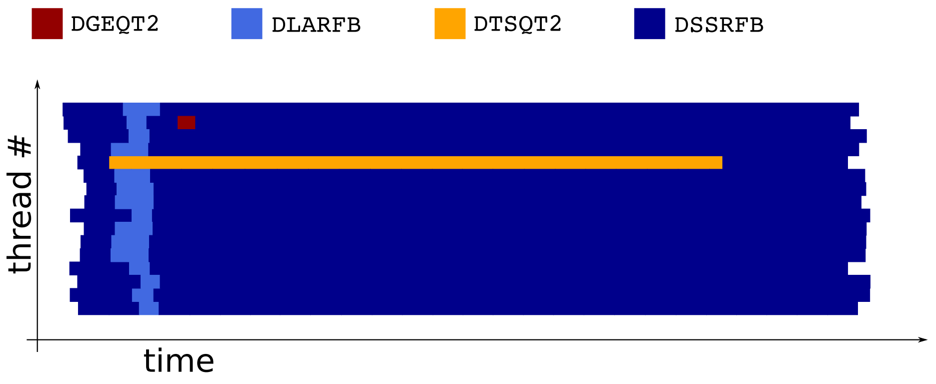

Fig. 4. The execution flow for dynamic scheduling, out of order execution of Algorithm 1.

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.

3.4 Block Data Layout

The major limitation of performing very fine grain computations, is that the BLAS library generally have very poor performance on small blocks. This situation can be considerably improved by storing matrices in Block Data Layout (BDL) instead of the Column Major Format that is the standard storage format for FORTRAN arrays.

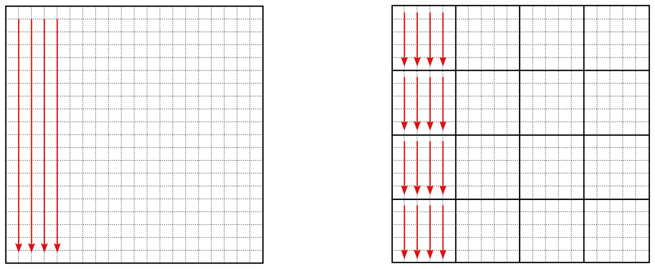

Fig. 5. A comparison of Column Major storage format (*left*) and Block Data Layout (*right*).

Figure 5 compares Column Major Format (*left*) and Block Data Layout (*right*). In BDL a matrix is split into blocks and each block is stored into contiguous memory locations. Each block is stored in Column Major Format and blocks are stored in Column Major Format with respect to each other. As a result the access pattern to memory is more regular and BLAS performance is considerably improved. The benefits of BDL have been extensively studied in the past, for example in **?**, and recent studies like **?** demonstrate how fine-granularity parallel algorithms can benefit from BDL.

4 Performance Results

The performance of the tiled QR factorization with dynamic scheduling of tasks has been measured on the systems listed in Table 2 and compared to the performance of the fork-join approach, i.e., the standard algorithm for block QR factorization of LAPACK associated with multithreaded BLAS.

Figures 6, 7, 8, 9 report the performance of the QR factorization for both the block algorithm with dynamic scheduling and the LAPACK algorithm with

	8-way dual Opteron	2-way quad Clovertown
Architecture	Dual-Core AMD Opteron$^{\text{TM}}$8214	Intel®Xeon®CPU X5355
Clock speed	2.2 GHz	2.66 GHz
# cores	$8 \times 2 = 16$	$2 \times 4 = 8$
Peak performance	70.4 Gflop/s	85.12 Gflop/s
Memory	62 GB	16 GB
Compiler suite	Intel 9.1	Intel 9.1
BLAS libraries	GotoBLAS-1.15 MKL-9.1	GotoBLAS-1.15 MKL-9.1

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

multithreaded BLAS. A block size of 200 has been used for the block algorithm while the block size for the LAPACK algorithm[6] has been tuned in order to achieve the best performance for all the combinations of architecture and BLAS library.

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

The graphs on the left part of each figure show the performance measured using the maximum number of cores available on each system with respect to the problem size. The graphs on the right part of each figure show the weak scalability, i.e. the flop rates versus the number of cores when the local problem size is kept constant (nloc=5,000) as the number of cores increases.

Figures 6, 7, 8, 9 show that, despite the higher operation count, the block algorithm with dynamic scheduling is capable of completing the QR factorization in less time than the LAPACK algorithm when the parallelism degree is high enough that the benefits of the asynchronous execution overcome the penalty of the extra flops. For lower numbers of cores, in fact, the fork-join approach has a good scalability and completes the QR factorization in less time than the block algorithm because of the lower flop count. Note that the actual execution rate of the block algorithm for QR factorization with dynamic scheduling (i.e., the dashed curves) is always higher than that of the LAPACK algorithm with multithreaded BLAS even for low numbers of cores. The actual performance of the block algorithm, even if considerably higher than that of the fork-join one, is still far from the peak performance of the systems used for the measures. This is mostly due to two factors. First the nature of the BLAS operations involved; the `DGEQR2` and the `DLARFT` in the LAPACK algorithm and the `DGEQT2` and `DTSQT2` in the block algorithm are based on Level 2 BLAS operations that, being memory

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

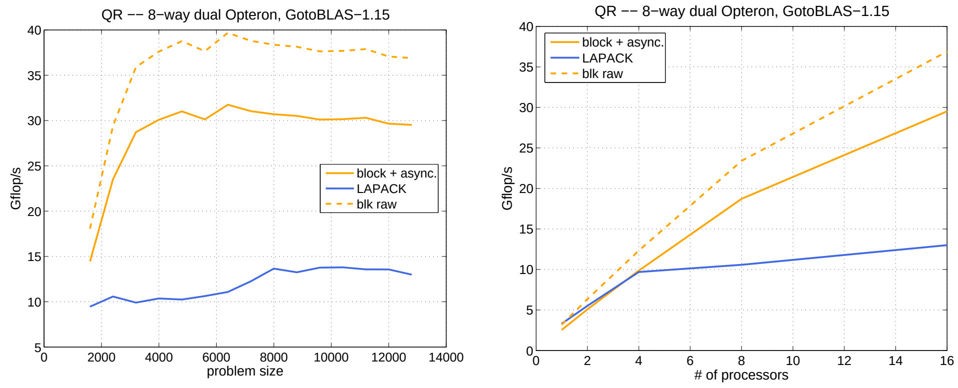

Fig. 6. Comparison between the performance of the block algorithm with dynamic scheduling using GotoBLAS-1.15 on an 8-way dual Opteron system. The dashed curve reports the raw performance of the block algorithm with dynamic scheduling, i.e., the performance as computed with the true operation count in Equation (4).

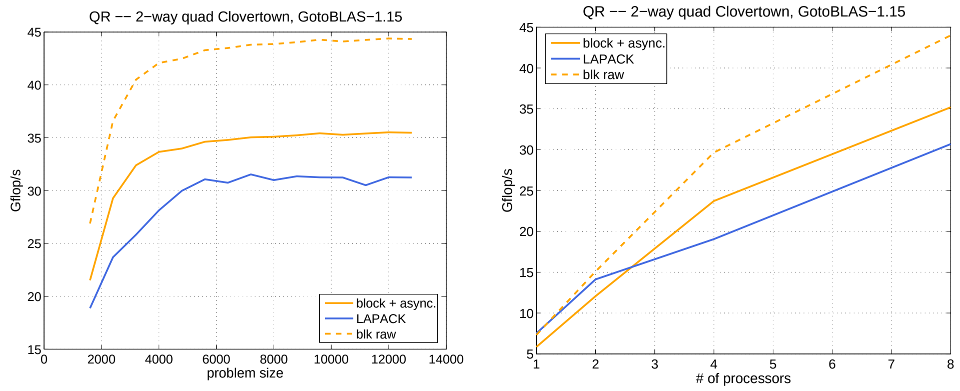

Fig. 7. Comparison between the performance of the block algorithm with dynamic scheduling using GotoBLAS-1.15 on an 2-way quad Clovertown system. The dashed curve reports the raw performance of the block algorithm with dynamic scheduling, i.e., the performance as computed with the true operation count in Equation (4).

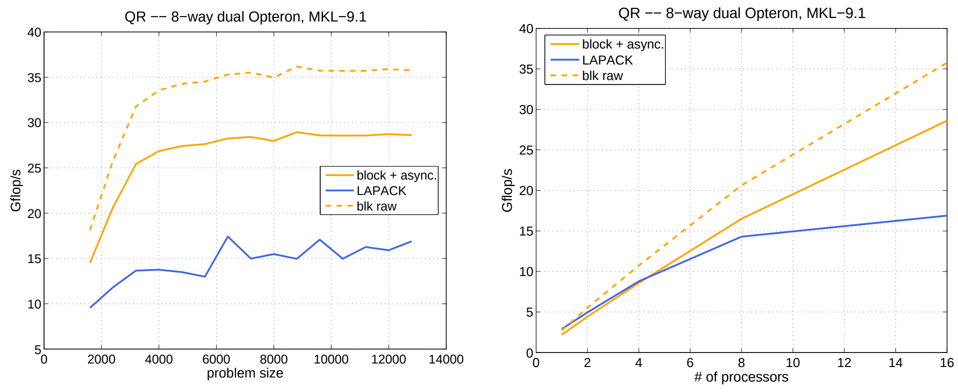

Fig. 8. Comparison between the performance of the block algorithm with dynamic scheduling using MKL-9.1 on an 8-way dual Opteron system. The dashed curve reports the raw performance of the block algorithm with dynamic scheduling, i.e., the performance as computed with the true operation count in Equation (4).

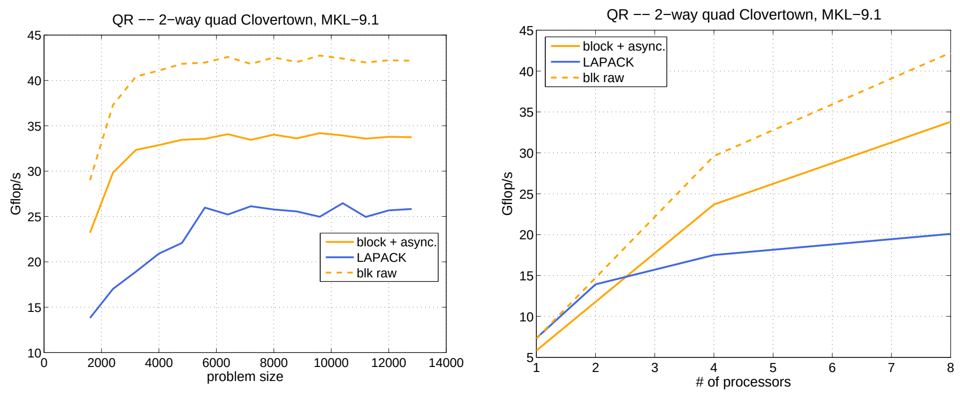

Fig. 9. Comparison between the performance of the block algorithm with dynamic scheduling using MKL-9.1 on an 2-way quad Clovertown system. The dashed curve reports the raw performance of the block algorithm with dynamic scheduling, i.e., the performance as computed with the true operation count in Equation (4).

bound, represent a limit for performance. Second, the performance of BLAS routines on small size blocks. The block size used in the experiments reported above is 200; this block size represents a good compromise between flexibility of the scheduler and performance of the BLAS operations but it is far from being ideal. Such a block size, in fact, does not allow a good task scheduling for smaller size problems and still the performance of BLAS operations is far from what can be achieved for bigger size blocks.

5 Conclusion

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

We note that the 25% overhead can be easily reduced by using nonsquare blocks. For example, using $2b \times b$ blocks, the overhead reduces to 12.5%.

While this paper only addresses the QR factorization, it is straightforward to derive with the same ideas the two important computational routines that consists in applying the Q factor to a set of vectors (see DORMQR in LAPACK) and constructing the Q-factor (see DORGQR in LAPACK).
The ideas behind this work can be extended in many directions:

Implement other linear algebra operations. The LU factorization can be performed with an algorithm that is analogous to the QR one described in Section 3. This algorithm has been discussed in **??** as a way of improving the out-of-core LU factorization. Even though the only difference between the block algorithms for the LU and QR factorizations is in the elementary operations, in the LU case the cost of block algorithm is 50% higher than the LAPACK algorithm. For this reason, the benefits of the improved scalability may be visible only at very high processor counts or may not be visible at all. Techniques must be investigated to eliminate or reduce the extra cost.
The same to blocking may also be applied to other two sided transformations like Hessenberg reduction, Tridiagonalization and Bidiagonalization. In these transformations, Level 2 BLAS operations are predominant and panel reductions account for almost 50% of the time of a sequential execution. Breaking the panel into smaller tasks that can be executed in parallel with other tasks may yield considerable performance improvements.

Enforcing data locality. The results proposed in **?** show that enforcing data locality and CPU affinity may provide considerable benefits. It must be noted that the improvements that can be expected on non-multicore SMPs are higher than on currently available multicore systems and this is due to the fact that on multicores, some of the higher level memories are shared between multiple cores. Moreover enforcing data locality has a major drawback in the fact that it seriously limits the scheduling of tasks since each core can only be assigned tasks that operate on data that resides on the memory associated with it. Preliminary results show that enforcing data locality and CPU affinity provides a slight speedup on the 8-way Dual Opteron system which is a NUMA architecture. These techniques require further investigation.

Implement the same algorithms in distributed memory systems. The fact that the block algorithms for QR and LU factorizations only require loose synchronization between tasks make them also good candidates for the implementation on distributed memory systems based on MPI communications.

Implement the same algorithms on the STI Cell architecture. In the STI Cell processor, no caches are present but a small, explicitly managed memory is associated to each core. Due to the small size of these local memories (only 256 KB), the LAPACK algorithms for LU and QR factorizations cannot be efficiently implemented. The block algorithms for LU and QR factorizations represent ideal candidates for the STI Cell architecture since they can be parallelized with a very fine granularity.

Explore the usage parallel programming environments. The task of implementing Linear Algebra operations with dynamic scheduling of tasks on multicore architectures can be considerably simplified by the use of graph driven parallel programming environments. One such environment is SMP Superscalar**?** developed at the Barcelona Supercomputing Center. SMP Superscalar addresses the automatic exploitation of the functional parallelism of a sequential program in multicore and SMP environments. The focus in on the portability, simplicity and flexibility of the programming model. Based on a simple annotation of the source code, a source to source compiler generates the necessary code and a runtime library exploits the existing parallelism by building at runtime a task dependency graph. The runtime takes care of scheduling the tasks and handling the associated data. Besides, a temporal locality driven task scheduling can be implemented.

Acknowledgment

We would like to thank John Gilbert from University of California at Santa Barbara for sharing with us its 8-way dual Opteron system.

Bibliography