Implementing Linear Algebra Routines on Multi-Core Processors with Pipelining and a Look Ahead

Jakub Kurzak[1] and Jack Dongarra[2]

[1] University of Tennessee, Knoxville TN 37996, USA,
kurzak@cs.utk.edu, http://www.cs.utk.edu/~kurzak
[2] University of Tennessee, Knoxville TN 37996, USA,
Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA,
dongarra@cs.utk.edu, http://www.netlib.org/utk/people/JackDongarra/

Abstract. Linear algebra algorithms commonly encapsulate parallelism in Basic Linear Algebra Subroutines (BLAS). This solution relies on the fork-join model of parallel execution, which may result in suboptimal performance on current and future generations of multi-core processors. To overcome the shortcomings of this approach a pipelined model of parallel execution is presented, and the idea of the look ahead is utilized in order to suppress the negative effects of sequential formulation of the algorithms. Application to one-sided matrix factorizations, LU, Cholesky and QR, is described. Shared memory implementation using POSIX threads is presented.

1 Introduction

The standard approach to parallelization of numerical linear algebra algorithms for shared memory systems, utilized by the LAPACK library [1, 2], is to rely on parallel implementation of BLAS [3]. It is a proven method to extract parallelism from Level 3 BLAS routines. However, as the number of processors or cores grow, practice shows that parallelization of Level 1 and 2 routines is unlikely to yield speedups and can even result in slowdowns. By the same token Level 1 and 2 BLAS portions of the algorithms effectively decrease the benefits of parallelization and limit achievable performance. A more flexible approach is required to address new generations of multi-core processors, which are expected to have tens, and possibly hundreds, of cores in near future.

The technique of the *look ahead* can be used to remedy the problem by overlapping the execution of less efficient operations with the more efficient ones. Also, the use of different levels of the *look ahead* is investigated and the idea of a dynamic *look ahead* is introduced, where the algorithm execution path is decided at runtime. The following section describes briefly three one-sided matrix factorizations, LU, Cholesky and QR, more precisely their block versions implemented in the LAPACK library. Then the technique of the *look ahead* is introduced. Static and dynamic variants are discussed. Presentation of performance results and their discussion follows.

2 Factorizations

The LU factorization with partial row pivoting of an $m \times n$ real matrix A has the form

$$A = PLU,$$

where L is an $m \times n$ real unit lower triangular matrix, U is an $n \times n$ real upper triangular matrix and P is a permutation matrix. The description of the block algorithm can be found in [4, 5]. In LAPACK the double precision algorithm is implemented by the DGETRF routine. A single step of the algorithm is implemented by a sequence of calls to the following LAPACK and BLAS routines: DGETF2, DLASWP, DTRSM, DGEMM, where DGETF2 factorizes a block of columns of the matrix (the panel) and DLASWP, DTRSM, DGEMM apply appropriate transformations to the submatrix to the right from the panel.

The Cholesky factorization of an $n \times n$ real symmetric positive definite matrix A has the form

$$A = LL^T,$$

where L is an $n \times n$ real lower triangular matrix with positive diagonal elements. The formulation of the block algorithm is analogous to the one for LU factorization. In LAPACK the double precision algorithm is implemented by the DPOTRF routine. A single step of the algorithm is implemented by a sequence of calls to the following LAPACK and BLAS routines: DSYRK, DPOTF2, DGEMM, DTRSM. Here, for simplicity, only the case of lower triangular coefficient matrix is considered. In this case the routines DSYRK and DPOTF2 factorize a block of rows of the matrix (the panel) and the DGEMM and DTRSM apply appropriate transformations to the submatrix below the panel.

The QR factorization of an $m \times n$ real matrix A has the form

$$A = QR,$$

where Q is an $m \times m$ real orthogonal matrix and R is an $m \times n$ real upper triangular matrix. The traditional algorithm for QR factorization applies a series of elementary Householder matrices of the general form

$$H = I - \tau v v^T,$$

where v is a column vector and τ is a scalar. In the block form of the algorithm a product of nb elementary Householder matrices is represented in the form[6, 7]

$$H_1 H_2 \ldots H_{NB} = I - VTV^T,$$

where V is an $n \times n$ real matrix those columns are the individual vectors v and T is an $nb \times nb$ real upper triangular matrix. For the derivation of the blocked algorithm the reader is referred to the original papers mentioned above. In LAPACK the double precision algorithm is implemented by the DGEQRF routine. A single step of the algorithm is implemented by a sequence of calls to the following LAPACK routines: DGEQR2, DLARFT, DLARFB, where DGEQR2 and DLARFT operate on a block of columns of the matrix (the panel) and DLARFB operates on the submatrix to the right from the panel.

3 Parallelization

Block formulations of the three factorizations discussed, as well as many other one-sided factorizations, follow a common scheme. In a single step of each algorithm, first operations are applied to a single block of rows or columns, referred to as the panel, then the result is applied to the remaining portion of the matrix. The panel operations are usually implemented with Level 1 and 2 BLAS, and, in most cases, achieve the best performance when executed on a single processor. By the same token, it is most straightforward to use one dimensional partitioning of work for parallel implementation, by cyclic assignment of blocks of rows or blocks of columns to processors, depending on the orientation of the panel; this is the approach used here.

It is a well known fact that matrix factorizations have left-looking and right-looking formulations [4]. It has even been observed that transition between the two can be done by automatic code transformations [8], although more powerful methods than simple dependency analysis is necessary. Another well known fact is that the technique of the *look ahead* can be used to significantly improve the performance of matrix factorizations, a method based on performing panel factorizations in parallel with the update to the remaining submatrix from previous step of the algorithm [9]. Also, the *look ahead* can be of arbitrary depth and an example of software utilizing this idea is the high performance LINPACK benchmark (HPL) [10, 11]. The *look ahead* is nothing else, but altering the order of operations in the factorization. A great number of permutations are legal, as long as algorithmic dependencies are not violated (Figure 1). It can be observed that the right-looking and left-looking formulation of a matrix factorization are on two opposite ends of a wide spectrum of possible execution paths, with the *look ahead* providing a transition between them. If the straight right-looking formulation is regarded as one with the *look ahead* of zero, then the left-looking formulation is equivalent to the right looking formulation with the maximum possible *look ahead* for a given problem.

4 Arbitrary (Static) Look Ahead

Classic implementation of a one-sided matrix factorization follows the execution pattern where the panel factorizations and the updates to the submatrix to the right from the panel are sequentially ordered. Panel factorization is performed by a single processor, while other processors are idle. Alternatively, when the first block of columns (block of rows) from step N of the factorization has been updated, one processor can perform panel factorization from step $N + 1$ on that block, when the remaining processors continue applying the update from step N, which decreases the idle time.

An arbitrary number of panels can be factorized ahead of updates to the submatrix on the right. Panels can be factorized up to an arbitrary depth. When this depth is reached an update has to be finished before another panel can be factorized. The order of execution of operations is determined by the depth of

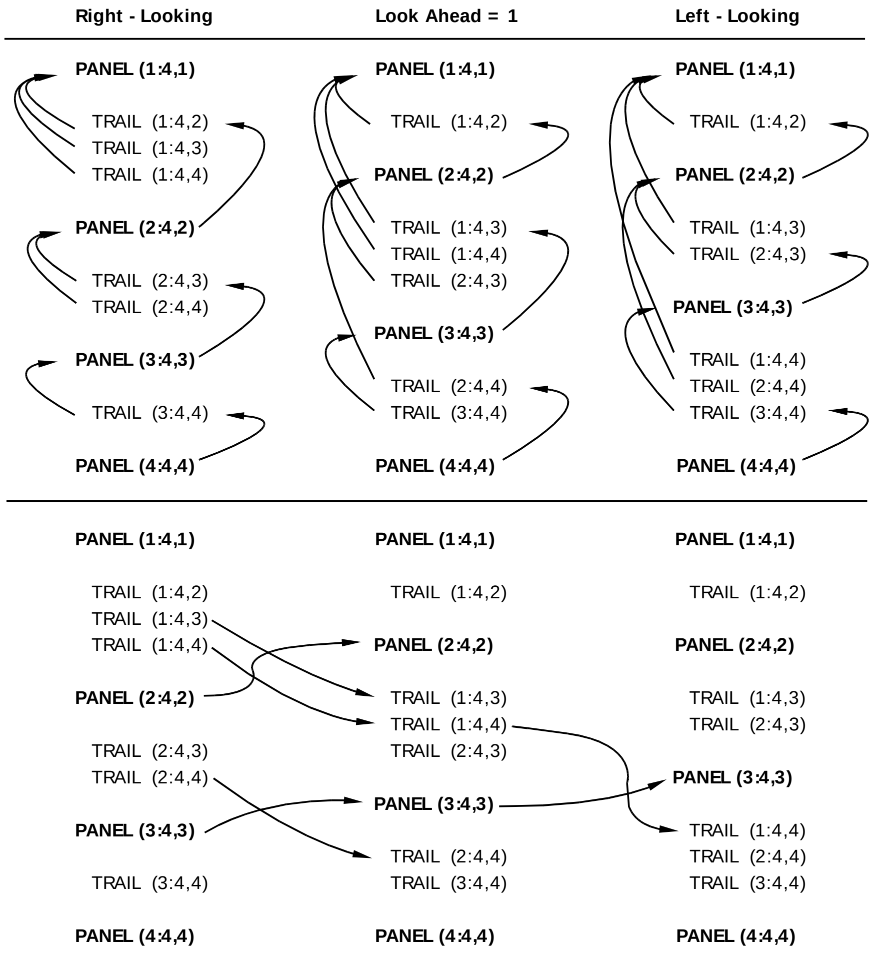

Fig. 1. Different variants of block LU factorization with 1D block cyclic work partitioning for a problem with the coefficient matrix of size 4 by 4 blocks. The top part shows dependencies, the bottom part illustrates the rearrangement of execution stages.

the *look ahead* and static throughout the execution of the factorization. Figure 2 shows the simplified code implementing arbitrary (static) *look ahead.* At each step the cycle is followed by checking dependencies and stalling if necessary, executing the operation, updating the progress, and making a transition to a next operation. The transition is always known a priori based on the current stage and the depth of the *look ahead*, since the static nature of the algorithm.

The deeper the depth, the less the processors stall at the end of the factorization. This is because the necessary panel is readily available, when the time comes to apply the update. At the same time, with deeper *looks ahead*, stalls occur at the beginning of execution, when processors wait for a block of columns (of rows) to be updated in order to apply a forerunning panel factorization.

```
while (1) {
    switch (task.type) {

        case PANEL:
            check_dependencies();
            dgetf2();       dsyrk();        dgeqr2();
                            dpotf2();       dlarft();
            update_progress();
            make_transition();
            break;

        case COLUMN:
            check_dependencies();
            dlaswp();       dgemm();       dlarfb();
            dtrsm();        dtrsm();
            dgemm();
            update_progress();
            make_transition();
            break;

        case END:
            check_dependencies();
            for ()
                dlaswp();
            return;
        }
    }
```

Fig. 2. Simplified code for one-sided factorizations with an arbitrary (static) *look ahead.*

5 Dynamic Look Ahead

The idea of the dynamic *look ahead* comes from the observation that the benefits of a deep *look ahead* are obliterated by the stalls, or bubbles, at the beginning of the factorization. The basic idea behind the dynamic *look ahead* is to implement the left-looking variant of the algorithm, where the panel factorizations are performed as soon as possible, with the modification that if the panel factorization introduces a stall, then an update to a block of columns (or rows) of the right submatrix is performed instead. The updating continues only until next panel factorization is possible. By the same token, dynamic *look ahead* is implemented by dynamic scheduling of work at runtime. Figure 3 shows the simplified code implementing the dynamic *look ahead*. Here the steps of checking dependencies and making a transition are merged into the step of fetching next task, where the choice of transition is made dynamically at run-time depending on the progress of the execution.

```
while(1) {
    fetch_task();

    switch(task.type) {

        case PANEL:
            dgetf2();       dsyrk();        dgeqr2();
                            dpotf2();       dlarft();
            update_progress();
            break;

        case COLUMN :
            dlaswp();       dgemm();        dlarfb();
            dtrsm();        dtrsm();
            dgemm();
            update_progress();
            break;

        case END:
            for ()
                dlaswp();
            return;
        }
    }
```

Fig. 3. Simplified code for one-sided factorizations with a dynamic *look ahead*.

6 Results

Results presented here were collected on a shared memory system with two 1.8 GHz dual-core AMD Opteron^{TM} 265 processors. GOTO BLAS [12] version 1.05 was used, the most recent one at the time of writing this paper. Block size of 64 was used in all cases.

Figure 4 shows Gantt chart of execution of LU factorization using different approaches ordered from top to bottom according to their relative performance.

On top is the right-looking version, where all but one processor are stalled by the panel factorization. Below is the left-looking version, which can also be considered the right-looking version with maximum possible amount of *look ahead*. Here stalls are eliminated at the end of factorization, but multiple bubbles are introduced at the beginning. Next are right-looking factorizations with the *look ahead* of depth one and two. Lastly, at the bottom is the factorization with the dynamic *look ahead*.

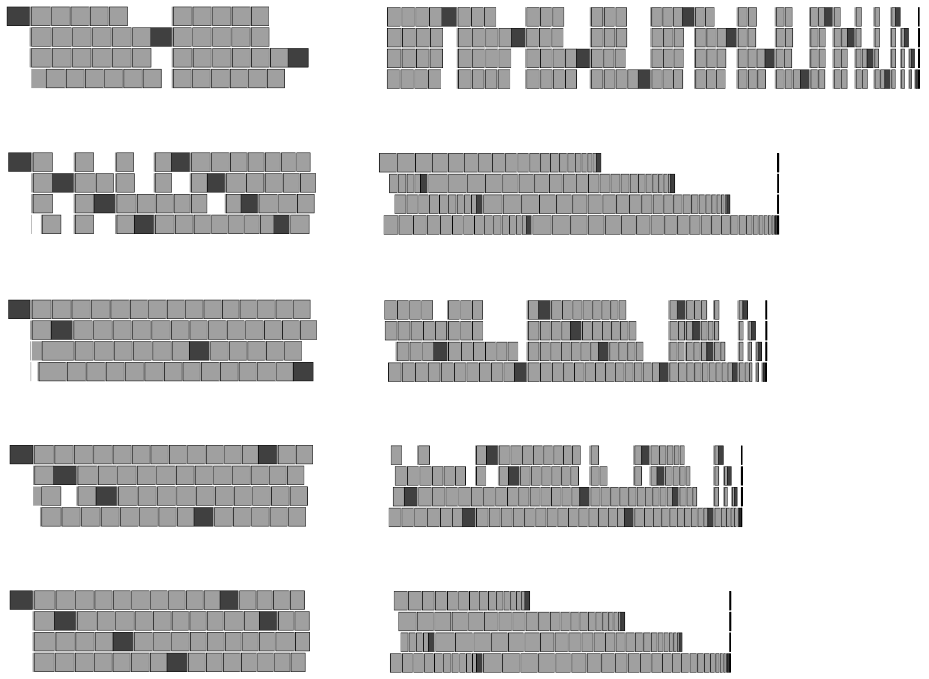

Fig. 4. Gantt chart for illustration of execution of different variants of one-sided factorizations. From top to bottom: right-looking (*look ahead* of depth zero), left-looking (right-looking with maximum *look ahead*), right-looking with *look ahead* of depth one, right-looking with *look ahead* of depth two, dynamic *look ahead*.

Figure 5, 6 and 7 show performance results for LU factorization, Cholesky factorization and QR factorization respectively.

The code with 1D partitioning suffers from two main performance disadvantages. First problem is that, on most processors, Level 3 BLAS routines like DGEMM do not achieve high performance unless called on a relatively big matrix. In the case of 1D partitioning with arbitrary or dynamic *look ahead*, those operations are always performed on relatively narrow slices of width equal to the block size. Fortunately it was not a major problem on the system used here,

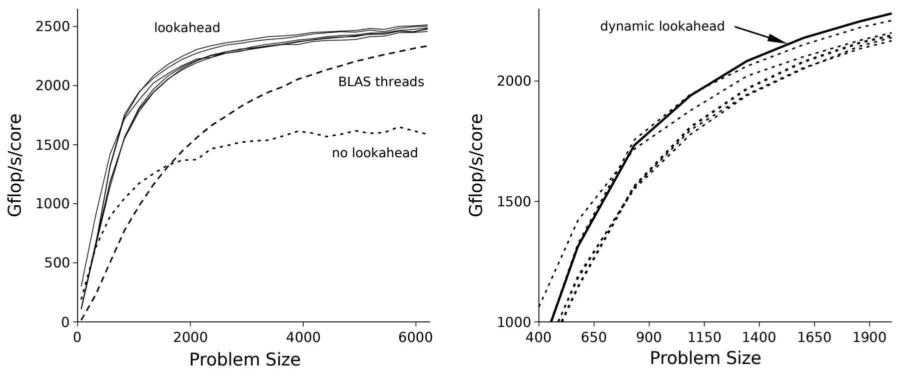

Fig. 5. Performance of different variants of LU factorization.

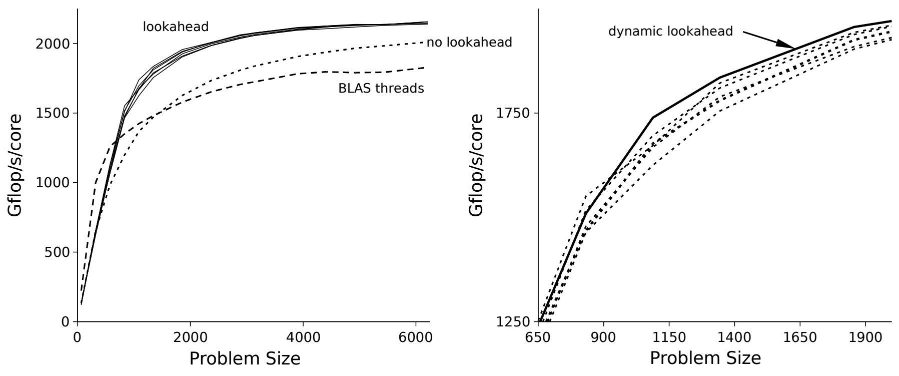

Fig. 6. Performance of different variants of Cholesky factorization.

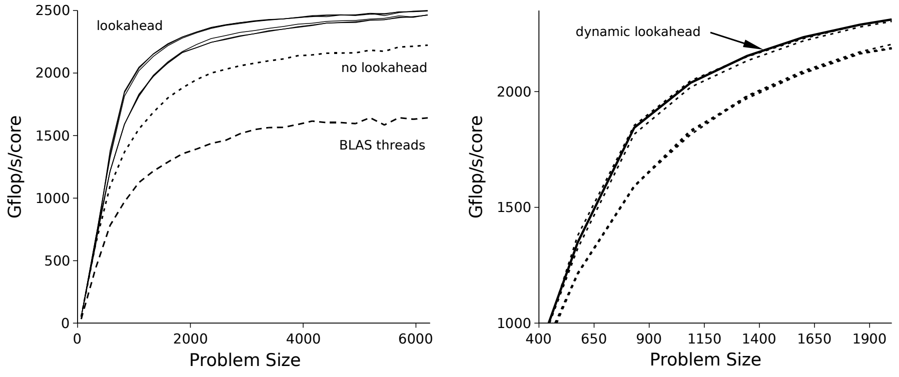

Fig. 7. Performance of different variants of QR factorization.

but the issue deserves a closer look on a wider range of hardware platforms. Second, 1D partitioning introduces serious load imbalance, which is most clearly visible in the "tail" at the end of execution of the left-looking algorithm and the dynamic *look ahead* algorithm. The disadvantages of 1D partitioning without the *look ahead* show clearly for LU factorization, where the BLAS-parallel code performs much better, especially for larger matrices.

The significant performance advantage of the code with 1D partitioning comes from replacing the fork-join model of execution of threaded BLAS. All BLAS operations applying the update to the submatrix on the right from the panel can be performed by a single processor without synchronization with the others, whereas synchronization (forking and joining) between each BLAS call is required if BLAS-parallel model is used. The advantages of 1D partitioning without the *look ahead* are most visible for QR factorization, where the BLAS-parallel code performs much worse, than the code with 1D partitioning.

At the same, time the introduction of the *look ahead* brings excellent performance gain compared to either the code without the *look ahead*, as well as the code with parallelism in the BLAS. Also, it can be concluded that the depth of the *look ahead* is not performance critical, at least on shared memory systems with small numbers of cores. The impact of the depth of the *look ahead* is expected to get bigger with growing number of cores. The *looka head* is also expected to play an important role on distributed memory systems, where it will provide means for hiding the communication latency.

Although at this number of cores the performance of dynamic *look ahead* can be matched by static *look ahead*, the dynamic *look ahead* sets the upper bound of the performance gains achievable from the *look ahead*, and at the same time, eliminates the guesswork of setting the optimal *look ahead* depth.

7 Conclusions

The application of the technique of the *look ahead* was discussed in two algorithmic variants, with arbitrary (static) *look ahead*, where the algorithmic path is predetermined, and with dynamic *look ahead*, where the path is decided at runtime. The results collected on a modern multi-core system showed strong advantages of the idea of the *look ahead* in general. At the same time, it was shown that dynamic *look ahead* sets the upper bound on achievable performance gains.

8 Future Plans

The most important work envisioned in the future is application of the ideas presented here to the case of 2D partitioning, which will become necessary for load balancing with rapidly growing number of cores in multi-core processors.

9 Acknowledgements

The authors would like to thank Dr. Julien Langou for many interesting conversations and helpful comments about the work presented here.

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J.W., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide. SIAM (1992)
2. Barker, V.A., Blackford, L.S., Dongarra, J., Du Croz, J., Hammarling, S., Marinova, M., Wasniewski, J., Yalamov, P.: LAPACK95 Users' Guide. SIAM (1992)
3. Basic Linear Algebra Technical Forum: Basic Linear Algebra Technical Forum Standard. (2001)
4. Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A.: Numerical Linear Algebra for High-Performance Computers. SIAM (1998)
5. Demmel, J.W.: Applied Numerical Linear Algebra. SIAM (1997)
6. Bischof, C., van Loan, C.: The WY representation for products of householder matrices. J. Sci. Stat. Comput. **8** (1987) 2–13
7. Schreiber, R., van Loan, C.: A storage-efficient WY representation for products of householder transformations. J. Sci. Stat. Comput. **10** (1991) 53–57
8. Menon, V., Pingali, K.: Look left, look right, look left again: An application of fractal symbolic analysis to linear algebra code restructuring. Int. J. Parallel Comput. **32**(6) (2004) 501–523
9. Strazdins, P.E.: A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Int. J. Parallel Distrib. Systems Networks **4**(1) (2001) 26–35
10. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK Benchmark: past, present and future. Concurrency Computat.: Pract. Exper. **15** (2003) 803–820
11. Petitet, A., Whaley, R.C., Dongarra, J.J., Cleary, A.: HPL - A portable implementation of the high-performance linpack benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl/ (2006)
12. Goto, K., van de Geijn, R.: On reducing TLB misses in matrix multiplication. Technical Report TR-02-55, Department of Computer Sciences, University of Texas at Austin (2002)