Towards an Efficient Tile Matrix Inversion of Symmetric Positive Definite Matrices on Multicore Architectures

Candidate to the Best Student Paper Award

Emmanuel Agullo[1,3], Henricus Bouwmeester[2,4], Jack Dongarra[1,5], Jakub Kurzak[1,6], Julien Langou[2,7], and Lee Rosenberg[2,8]

[1] Dpt of Electrical Engineering and Computer Science, University of Tennessee, 1122 Volunteer Blvd, Claxton Building, Knoxville, TN 37996-3450, USA
[2] Dpt of Mathematical and Statistical Sciences, University of Colorado Denver, Campus Box 170, P.O. Box 173364, Denver, Colorado 80217-3364, USA,
Research was supported by the National Science Foundation grant no. NSF CCF-811520.
[3] Emmanuel.Agullo@eecs.utk.edu
[4] Henricus.Bouwmeester@email.ucdenver.edu
[5] Jack.Dongarra@eecs.utk.edu
[6] Jakub.Kurzak@eecs.utk.edu
[7] Julien.Langou@ucdenver.edu
[8] Lee.Rosenberg@email.ucdenver.edu

Abstract. The algorithms in the current sequential numerical linear algebra libraries (*e.g.* LAPACK) do not parallelize well on multicore architectures. A new family of algorithms, the *tile algorithms*, has recently been introduced. Previous research has shown that it is possible to write efficient and scalable tile algorithms for performing a Cholesky factorization, a (pseudo) LU factorization, a QR factorization, and computing the inverse of a symmetric positive definite matrix. In this extended abstract, we revisit the computation of the inverse of a symmetric positive definite matrix. We observe that, using a dynamic task scheduler, it is relatively painless to translate existing LAPACK code to obtain a ready-to-be-executed tile algorithm. However we demonstrate that, for some variants, non trivial compiler techniques (array renaming, loop reversal and pipelining) need then to be applied to further increase the parallelism of the application. We present preliminary experimental results.

1 Introduction

The appropriate direct method to compute the solution of a symmetric positive definite system of linear equations consists of computing the Cholesky factorization of that matrix and then solving the underlying triangular systems. It is not recommended to use the inverse of the matrix in this case. However some applications need to explicitly form the inverse of the matrix. A canonical example

is the computation of the variance-covariance matrix in statistics. Higham [15, p.260,§3] lists more such applications.

With their advent, multicore architectures [21] induce the need for algorithms and libraries that fully exploit their capacities. A class of such algorithms – called tile algorithms [8, 9] – has been developed for one-sided dense factorizations (Cholesky, LU and QR) and made available as part of the Parallel Linear Algebra Software for Multicore Architectures (PLASMA) library [2]. In this paper, we study this class of algorithms to the case of the (symmetric positive definite) matrix inversion. An identical study has already been performed in 2008 [11], and the associated software is present in the libflame library [23].

Besides constituting an important functionality for a library such as PLASMA, the study of the matrix inversion on multicore architectures represents a challenging algorithmic problem. Indeed, first, contrary to standalone one-sided factorizations that have been studied so far, the matrix inversion exhibits many anti-dependences [4] (Write After Read). Those anti-dependences can be a bottleneck for parallel processing, which is critical on multicore architectures. It is thus essential to investigate (and adapt) well known techniques used in compilation such as using temporary copies of data to remove anti-dependences to enhance the degree of parallelism of the matrix inversion. This technique is known as *array renaming* [4] (or *array privatization* [14]). Second, *loop reversal* [4] is to be investigated. Third, the matrix inversion consists of three successive steps (first of which is the Cholesky decomposition). In terms of scheduling, it thus represents an opportunity to study the effects of *pipelining* [4] those steps on performance.

The current version of PLASMA (version 2.1) is scheduled statically. Initially developed for the IBM Cell processor [17], this static scheduling relies on POSIX threads and simple synchronization mechanisms. It has been designed to maximize data reuse and load balancing between cores, allowing for very high performance [3] on today's multicore architectures. However, in the case of the matrix inversion, the design of an ad-hoc static scheduling is a time consuming task and raises load balancing issues that are much more difficult to address than for a stand-alone Cholesky decomposition, in particular when dealing with the pipelining of multiple steps. Furthermore, the growth of the number of cores and the more complex memory hierarchies make executions less deterministic. In this paper, we rely on an experimental in-house dynamic scheduler named QUARK [16]. This scheduler is based on the idea of expressing an algorithm through its sequential representation and unfolding it at runtime using data hazards (Read after Write, Write after Read, Write after Write) as constraints for parallel scheduling. The concept is rather old and has been validated by a few successful projects. We could as well have used schedulers from the Jade project from Stanford University [20], from the SMPSs project from the Barcelona Supercomputer Center [18], or from the SuperMatrix framework [10].

Our discussions are illustrated with experiments conducted on a dual-socket quad-core machine based on an Intel Xeon EMT64 processor operating at 2.26 GHz. Composed of 8 cores, the theoretical peak is equal to 9.0 Gflop/s per

core or 72.3 Gflop/s for the whole machine. The machine is running Mac OS X 10.6.2 and is shipped with the Apple vecLib v126.0 multithreaded BLAS [1] and LAPACK vendor library. We have installed reference LAPACK [5] v3.2.1, reference ScaLAPACK [7] v1.8.0, and libflame v3935 [23].

The rest of the paper is organized as follows. In Section 2, we explain a possible algorithmic variant for matrix inversion based on tile algorithms; we explain how we articulated it with our dynamic scheduler to take advantage of multicore architectures and we compare its performance against state-of-the-art libraries. In Section 3, we investigate the impact on parallelism and performance of different well known techniques used in compilation: loop reversal, array renaming and pipelining. We conclude and present future work directions in Section 4.

2 Tile in-place matrix inversion

Tile algorithms are a class of Linear Algebra algorithms that allow for fine granularity parallelism and asynchronous scheduling, enabling high performance on multicore architectures [3, 8, 9, 11, 19]. The matrix of order n is split into $t \times t$ square submatrices of order b ($n = b \times t$). Such a submatrix is of small granularity (we fixed $b = 200$ in this paper) and is called a *tile*. So far, tile algorithms have been essentially used to implement one-sided factorizations [3, 8, 9, 11, 19].

Algorithm 1 extends this class of algorithms to the case of the matrix inversion. As in state-of-the-art libraries (LAPACK, ScaLAPACK), the matrix inversion is performed *in-place*, *i.e.*, the data structure initially containing matrix A is directly updated as the algorithm is progressing, without using any significant temporary extra-storage; eventually, A^{-1} substitutes A. Algorithm 1 is composed of three steps. Step 1 is a Tile Cholesky Factorization computing the Cholesky factor L (lower triangular matrix satisfying $A = LL^T$). This step was studied in [9]. Step 2 computes L^{-1} by inverting L. Step 3 finally computes the inverse matrix $A^{-1} = {L^{-1}}^T L^{-1}$.

A more detailed description is beyond the scope of this extended abstract and is not essential to the understanding of the rest of the paper. However, we want to point out that the stability analysis of the block (or tile) triangular inversion is quite subtle and one should not replace too hastily "TRSM-then-TRTRI" by "TRTRI-then-TRMM" See [13] for a comprehensive exaplanation.

Each step is composed of multiple fine granularity tasks (since operating on tiles). These tasks are part of the BLAS (SYRK, GEMM, TRSM, TRMM) and LAPACK (POTRF, TRTRI, LAUUM) standards. Indeed, from a high level point of view, an operation based on tile algorithms can be represented as a Directed Acyclic Graph (DAG) [12] where nodes represent the fine granularity tasks in which the operation can be decomposed and the edges represent the dependences among them. For instance, Figure 1(a) represents the DAG of Step 3 of Algorithm 1.

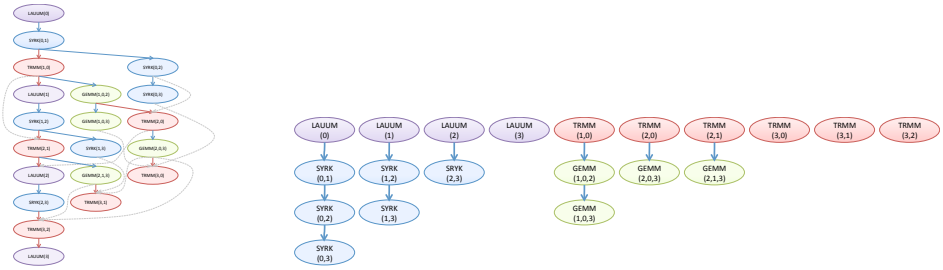
Algorithm 1 is based on the variants used in LAPACK 3.2.1. Bientinesi, Gunter and van de Geijn [6] discuss the merit of algorithmic variations in the case of the computation of the inverse of a symmetric positive definite matrix.

Algorithm 1: Tile In-place Cholesky Inversion (lower format). Matrix A is the on-going updated matrix (in-place algorithm).

Input: A, Symmetric Positive Definite matrix in tile storage ($t \times t$ tiles).
Result: A^{-1}, stored in-place in A.
Step 1: Tile Cholesky Factorization (compute L such that $A = LL^T$);
Variant 2;
for $j = 0$ **to** $t - 1$ **do**
  for $k = 0$ **to** $j - 1$ **do**
    $A_{j,j} \leftarrow A_{j,j} - A_{j,k} * A_{j,k}^T$ (SYRK(j,k)) ;
  $A_{j,j} \leftarrow CHOL(A_{j,j})$ (POTRF(j)) ;
  for $i = j + 1$ **to** $t - 1$ **do**
    for $k = 0$ **to** $j - 1$ **do**
      $A_{i,j} \leftarrow A_{i,j} - A_{i,k} * A_{j,k}^T$ (GEMM(i,j,k)) ;
  for $i = j + 1$ **to** $t - 1$ **do**
    $A_{i,j} \leftarrow A_{i,j}/A_{j,j}^T$ (TRSM(i,j)) ;

Step 2: Tile Triangular Inversion of L (compute L^{-1})
Variant 4;
for $j = t - 1$ **to** 0 **do**
  for $i = t - 1$ **to** $j + 1$ **do**
    $A_{i,j} \leftarrow A_{i,i} * A_{i,j}$ (TRMM(i,j)) ;
    for $k = i - 1$ **to** $j + 1$ **do**
      $A_{i,j} \leftarrow A_{i,j} + A_{i,k} * A_{k,j}$ (GEMM(i,j,k)) ;
    $A_{i,j} \leftarrow -A_{i,j}/A_{j,j}$ (TRSM(i,j)) ;
  $A_{j,j} \leftarrow TRINV(A_{j,j})$ (TRTRI(j)) ;

Step 3: Tile Product of Lower Triangular Matrices (compute $A^{-1} = {L^{-1}}^T L^{-1}$)
Variant 2;
for $i = 0$ **to** $t - 1$ **do**
  for $j = 0$ **to** $i - 1$ **do**
    $A_{i,j} \leftarrow A_{i,i}^T * A_{i,j}$ (TRMM(i,j)) ;
  $A_{i,i} \leftarrow A_{i,i}^T * A_{i,i}$ (LAUUM(i)) ;
  for $j = 0$ **to** $i - 1$ **do**
    for $k = i + 1$ **to** $t - 1$ **do**
      $A_{i,j} \leftarrow A_{i,j} + A_{k,i}^T * A_{k,j}$ (GEMM(i,j,k)) ;
  for $k = i + 1$ **to** $t - 1$ **do**
    $A_{i,i} \leftarrow A_{i,i} + A_{k,i}^T * A_{k,i}$ (SYRK(i,k)) ;



(a) In-place (Algorithm 1)

(b) Out-of-place (variant introduced in Section 3)

Fig. 1. DAGs of Step 3 of the Tile Cholesky Inversion ($t = 4$).

Although of definite interest, this is not the focus of this extended abstract. We will use the same variant enumerations as in [6]. With these notations, Algorithm 1 is called 242: variant 2 for POTRF, variant 4 for TRTRI and variant 2 for LAUUM. Variant 4 of TRTRI is identical to variant 1 of TRTRI but it starts from the bottom-right corner and ends at the top left corner. (Variant 1 is the reverse.)

We will see in the experimental section, Section 3, that this choice of variants is not optimal, however it gives an interesting case study.

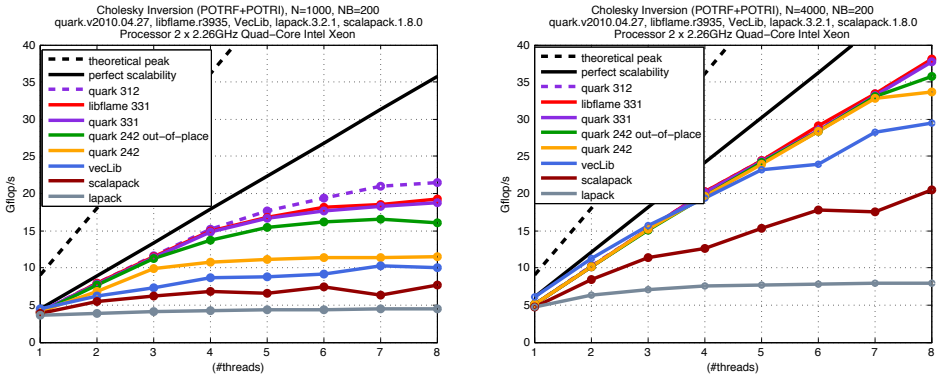
We have implemented Algorithm 1 using our dynamic scheduler QUARK introduced in Section 1. Figure 2 shows its performance against state-of-the-art libraries and the vendor library on the machine described in Section 1. For a matrix of small size while keeping the tile size reasonably large, it is difficult to extract parallelism and have full use of all the cores [3, 8, 9, 19]. We indeed observe a limited scalability ($N = 1000$, Figure 2(a)). However, tile algorithms (Algorithm 1) still benefit from a higher degree of parallelism than blocked algorithms [3, 8, 9, 19]. Therefore Algorithm 1 (in place) consistently achieves a significantly better performance than vecLib, ScaLAPACK and LAPACK libraries. With the tile size kept constant, a larger matrix size ($N = 4000$, Figure 2(b)) allows for a better use of parallelism. In this case, an optimized implementation of a blocked algorithm (vecLib) competes well against tile algorithms (in place) on few cores (left part of Figure 2(a)). However, only tile algorithms scale to a larger number of cores (rightmost part of Figure 2(b)) due to a higher degree of parallelism. In other words, the tile Cholesky inversion achieves a better *strong scalability* than the blocked versions, similarly to what had been observed for the factorization step [3, 8, 9, 19].

We see that the performance of the 242 variant (green and yellow lines) is mediocre and a more appropriate variant would be 331 (red and purple lines) or even better 312 (dashed purple). The reason for this is that the combinantion of variants in the 242 variant does not lend itself well to interleaving. Variant 2 of POTRF starts from the top-left corner and ends bottom-right, then variant 4 of TRTRI starts from the bottom-right corner and ends at the top-left corner being

followed by variant 2 of LAUUM which starts from the top-left corner and ends at the bottom-right. Due to the progression of each step within this combination, it is very difficult to interleave the tasks. More appropriately, a variant of TRTRI which progresses from top-left to bottom-right would afford a more aggressive interleaving of POTRF, TRTRI, and LAUUM. Variants 331 and 312 provide this combination. This corroborates the observation of Bientinesi, Gunter and van de Geijn: the 331 variant (e.g.) allows for a "*one-sweep*" algorithm [6].

We note that we obtain similar performance as the libflame libraries when we use the same 331 algorithmic variant. (See red and plain purple curves.) The main difference in this case being the schedulers (QUARK vs Supermatrix) which are performing equally for these experiments. We did not try to tune the parameters of either of these schedulers.

The best algorithmic combination of variants for our experimental conditions was 312 as observed in the plot with the dashed purple curve.



(a) $n = 1000$ (b) $n = 4000$

Fig. 2. Scalability of Algorithm 1 (in place) and its out-of-place variant introduced in Section 3, using our dynamic scheduler against libflame, vecLib, ScaLAPACK and LAPACK libraries.

3 Algorithmic study

In the previous section, we compared the performance of the tile Cholesky inversion against state-the-art libraries. In this section, we focus on tile Cholesky inversion and we discuss the impact of several possible optimizations of Algorithm 1 on its performance.

Array renaming (removing anti-dependences). The dependence between SYRK(0,1) and TRMM(1,0) in the DAG of Step 3 of Algorithm 1 (Figure 1(a)) represents the constraint that the SYRK operation (l. 28 of Algorithm 1) needs to read $A_{k,i} = A_{1,0}$ before TRMM (l. 22) can overwrite $A_{i,j} =$

	In-place case	Out-of-place case
Step 1	$3t-2$ (up)	$3t-2$ (up)
Step 2	$3t-2$ (down)	$2t-1$ (down)
Step 3	$3t-2$ (up)	t (up)

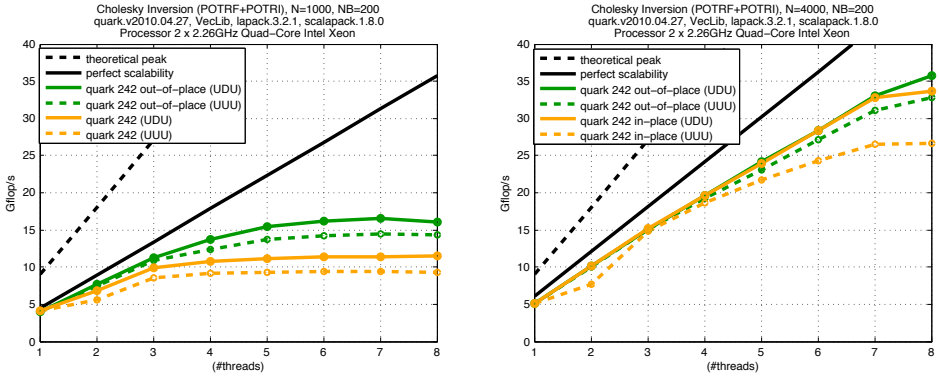
Table 1. Length of the critical path as a function of the number of tiles t.

$A_{1,0}$. This anti-dependence (Write After Read) can be removed by use of a temporary copy of $A_{1,0}$. Similarly, all the SYRK-TRMM anti-dependences, as well as TRMM-LAUMM and GEMM-TRMM anti-dependences can be removed. We have designed a variant of Algorithm 1 that removes all the anti-dependences by usage of a large working array (this technique is called *array renaming* [4] in compilation [4]). The subsequent DAG (Figure 1(b)) is split into multiple pieces (Figure 1(b)), leading to a shorter critical path (Table 1). We also implemented the out-of-place algorithm, within the framework of our dynamic scheduler. Figure 2(a) shows that our dynamic scheduler exploits this higher degree of parallelism to achieve a higher strong scalability even on small matrices ($N = 1000$). For a larger matrix (Figure 2(b)), the in-place algorithm already achieved very good scalability. Therefore, using up to 7 cores, their performance are similar. However, there is not enough parallelism with a 4000×4000 matrix to efficiently use all 8 cores with the in-place algorithm; thus the higher performance of the out-of-place version in this case (leftmost part of Figure 2(b)).

Loop reversal (exploiting commutativity). The most internal loop of each step of Algorithm 1 (l. 8, l. 17 and l. 26) consists in successive commutative GEMM operations. Therefore they can be performed in any order, among which increasing order and decreasing order of the loop index. Their ordering impacts the length of the critical path. Algorithm 1 orders those three loops as increasing (U) for POTRF, decreasing (D) for TRTRI, and increasing (U) for LAUUM. We had manually chosen these respective orders (UDU) because they minimize the critical path of each step (values reported in Table 1). A naive approach would have, for example, been comprised of consistently ordering the loops in increasing order (UUU). In this case (UUU), the critical path of TRTRI would have been equal to $t^2 - 2t + 4$ (in-place) or $(\frac{1}{2}t^2 - \frac{1}{2}t + 2)$ (out-of-place) instead of $3t - 2$ (in-place) or $2t - 1$ (out-of-place) for (UDU). Figure 3 shows how loop reversal impacts performance.

This optimization is important for libraries relying on a tile BLAS (e.g. libflame [23]). While any loop ordering is fine for a tile GEMM (e.g.) in term of correctness, the loop ordering has an influence on how fast tiles are freed by the tile GEMM operation. It is therefore critical that tile GEMM (e.g.) has the ability of switching the ordering of the tasks depending on the context. In our case, the optimal loop ordering for the 331 variant of Cholesky inversion is UUU and so the good loop ordering comes "naturally".

Pipelining. Pipelining (interleaving) the multiple steps of the inversion reduces the length of its critical path. For the in-place case, the critical path cannot be reduced since the final task of Step 1 must be completed before the first task



(a) $n = 1000$ (b) $n = 4000$

Fig. 3. Impact of loop reversal on performance.

of Step 2 can proceed and similarly for Step 2 to Step 3. (This is because we have chosen to study the 242 variant.) For the out-of-place case, it is reduced from $6t - 3$ to $5t - 2$ tasks. We studied the effect of pipelining on the performance of the inversion of a 8000×8000 matrix with an artificially large tile size ($b = 2000$ and $t = 4$). For the out-of-place case, the elapsed time grows from 16.4 to 19.0 seconds (16 % overhead) when pipelining is prevented.

4 Conclusion and future work

We have studied the problem of the computation of the inverse of a symmetric positive definite matrix on multicore architectures. This problem was already presented by Chan, Van Zee, Bientinesi, Quintana-Ortí, Quintana-Ortí, and van de Geijn in 2008 [11]. We are essentially following the same approach: starting from standard algorithms, we derive tile algorithms whose tasks are then scheduled dynamically.

Our experimental study has shown both an excellent scalability of these algorithms and a significant performance improvement compared to LAPACK and ScaLAPACK based libraries.

In perspective of [11], our contribution is to bring back to the fore well known issues in the domain of compilation. Indeed, we have shown the importance of loop reversal, array renaming and pipelining. The optimization of these are very important in the sense that they influence dramatically the shape of the DAG of tasks that is provided to our dynamic scheduler and consequently determine the degree of parallelism (and scalability) of the application.

The use of a dynamic scheduler allowed an out-of-the-box pipeline of the different steps whereas loop reversal and array renaming required a manual change to the algorithm. The future work directions consist of enabling the scheduler to

automatically perform loop reversal and array renaming. We exploited the commutativity of GEMM operations to perform array renaming. Their associativity would furthermore allow them to be processed in parallel (e.g. following a binary tree). Actually, the commutative and associative nature of addition allows one to execute the operations in the fashion of a DOANY loop [22]. The subsequent impact on performance is to be studied. Array renaming requires extra-memory, thus it will be interesting to address the problem of the maximization of performance under memory constraints.

Thanks

The authors would like to thank Anthony Danalis for his insights on compiler techniques, Robert van de Geijn for pointing us to reference [11], (we missed this very relevant reference in the first place!), and Ernie Chan for making the tuning and benchmarking of libflame as easy as possible.

References

1. BLAS: Basic linear algebra subprograms. http://www.netlib.org/blas/.
2. E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, and H. Ltaief. PLASMA Users' Guide. Technical report, ICL, UTK, 2009.
3. E. Agullo, B. Hadri, H. Ltaief, and J. Dongarrra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.
4. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.
5. E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1992.
6. P. Bientinesi, B. Gunter, and R. van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw.*, 35(1):1–22, 2008.
7. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.
8. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, 20(13):1573–1590, 2008.
9. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.
10. E. Chan. Runtime data flow scheduling of matrix computations. FLAME Working Note #39. Technical Report TR-09-22, The University of Texas at Austin, Department of Computer Sciences, August 2009.

11. E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 123–132, New York, NY, USA, 2008. ACM.
12. N. Christofides. *Graph Theory: An algorithmic Approach*. 1975.
13. J. J. Du Croz and N. J. Higham. Stability of methods for matrix inversion. *IMA Journal of Numerical Analysis*, 12:1–19, 1992.
14. R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks®. *IEEE Trans. Parallel Distrib. Syst.*, 9(1):5–23, 1998.
15. N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.
16. J. Kurzak and J. Dongarra. Fully dynamic scheduler for numerical computing on multicore processors. *University of Tennessee CS Tech. Report, UT-CS-09-643*, 2009.
17. J. Kurzak and J. Dongarra. QR factorization for the Cell Broadband Engine. *Sci. Program.*, 17(1-2):31–42, 2009.
18. J. M. Perez, R. M Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of IEEE Cluster Computing 2008*, 2008.
19. G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and E. Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*, 36(3), 2009.
20. M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 6:28–38, 1993.
21. H. Sutter. A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.
22. Michael Wolfe. Doany: Not just another parallel loop. In *Proceedings of the 5th International Workshop on Languages andCompilers for Parallel Computing*, pages 421–433, London, UK, 1993. Springer-Verlag.
23. Field G. Van Zee. `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.