Blocked and Scalable Matrix Computations – Packed Cholesky, In-Place Transposition, and Two-Sided Transformations

Lars Karlsson

Licentiate Thesis, April 2009

Department of Computing Science
SE-901 87 Umeå

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

larsk@cs.umu.se

ISBN 978-91-7264-788-6
ISSN 0348-0542
UMINF 09.11

Printed by Print & Media, Umeå University, 2009

Abstract

Dense linear algebra algorithms are often regular and have a high arithmetic intensity. This allows them to achieve performance close to the peak performance even on the largest computing systems in the world. Factors that affect the performance of an implementation include the presence of a deep memory hierarchy, pipelines, SIMD units, and multiple processing cores. On a larger scale, several processors and memories are physically distributed and communicates over a high speed network. Techniques such as blocking, vectorization, and asynchronous communication are used in state-of-the-art implementations. One aim of the research in this area is to develop portable software libraries with high performance implementations of scalable algorithms.

High level linear algebra algorithms can be built on top of a relatively small set of fundamental operations, such as matrix multiplication, that perform the majority of the floating point operations. These operations, once tuned to a particular machine, provide portable performance to the higher level algorithms. This observation led to the standardization of an interface to three levels of operations known as the Basic Linear Algebra Subprograms (BLAS) and eventually to software libraries such as LAPACK and ScaLAPACK.

One topic studied in this thesis is alternative schemes for packed storage of symmetric or triangular matrices in a distributed memory environment. In particular, several implementations of the Cholesky factorization of dense matrices were designed and evaluated. Related to this topic is how to convert a matrix stored in a canonical row- or column-major storage format to a block format. A prototype library for conversion based on in-place matrix transposition is developed in connection with an evaluation of several known algorithms for the closely connected problem of in-place transposition.

This thesis also investigates ways of moving beyond the limitations of extracting parallelism at the BLAS levels of abstraction. One of our Cholesky algorithms interleaves computation with a complex communication algorithm and thereby eliminates the distinct boundaries between computation and communication that is common in synchronous parallel algorithms.

Two-sided transformations offer further challenges due to their often much smaller degree of concurrency. This thesis explores how priority-based dynamic scheduling on each node of a distributed memory system can be used to achieve a faster execution by eliminating synchronization overhead. The advantages of being able to express diverse and complex schedules within a sequential node program are clear, but the inherent overheads and limitations are topics for further research.

Sammanfattning

Algoritmer inom tät linjär algebra är ofta reguljära med en hög aritmetisk intensitet. Detta medger en prestandanivå som ligger mycket nära den maximala prestandan även på världens största datorsystem. Faktorer som påverkar en implementations prestanda innefattar djupa minneshierarkier, pipelines, SIMD-enheter, och flera beräkningskärnor. I större skala är flera processorer och minnen ihopkopplade med ett högpresterande nätverk. Programmeringstekniker såsom blockning, vektorisering, och asynkron kommunikation används i moderna implementationer. Ett mål med forskningen inom detta område är att utveckla portabla programbibliotek med högpresterande implementationer av skalbara algoritmer.

Linjär algebra-algoritmer kan byggas utifrån en relativt liten mängd grundläggande operationer, såsom matrismultiplikation, som utför majoriteten av flyttalsoperationerna. Dessa enklare operationer ger portabel prestanda åt komplexa algoritmer när de väl har optimerats för ett specifikt datorsystem. Denna observation ledde till standardiseringen av ett gränssnitt känt som Basic Linear Algebra Subprograms (BLAS) och så småningom till mjukvarubibliotek som LAPACK och ScaLAPACK.

En frågeställning som studeras i denna uppsats är alternativa packade lagringsformat för symmetriska eller triangulära matriser i en miljö med distribuerat minne. Flera implementationer av Cholesky-faktorisering av täta matriser designades och utvärderades. En relaterad frågeställning är hur konvertering *in situ* mellan kanoniska lagringsformat och block-format kan implementeras. Ett programbibliotek för sådan konvertering har utvecklats. Det baseras på kända algoritmer för matristransponering *in situ*.

Ytterligare ett ämne som studeras i denna uppsats är olika sätt att överbrygga de begränsningar som uppstår när parallellitet uttrycks på BLAS-nivå. En av Cholesky-algoritmerna väver samman beräkningar med en komplex kommunikationsalgoritm och på så sätt elimineras den tydliga gräns mellan kommunikation och beräkningar som är vanlig i synkrona parallella algoritmer.

Två-sidiga transformationer ställer större krav på schemaläggning av deluppgifter på grund av en låg grad av parallellitet. Vi har studerat hur dynamisk prioritetsbaserad schemaläggning på varje nod i ett system med distribuerat minne kan användas för att ge snabbare exekvering genom att eliminera synkroniseringskostnader. Fördelarna med att uttrycka vitt skilda och komplexa scheman från ett sekventiellt nodprogram är tydliga, men de inneboende kostnader och begränsningar som metoden innebär är föremål för framtida forskning.

Preface

This Licentiate Thesis consists of an introduction and the following four papers.

Paper I	F. G. Gustavson, L. Karlsson, and B. Kågström. Three Algorithms for Cholesky Factorization on Distributed Memory using Packed Storage[1]. In *Applied Parallel Computing: State of the Art in Scientific Computing*, Lecture Notes in Computer Science, LNCS4699, pp. 550-559. Springer-Verlag, 2007.
Paper II	F. G. Gustavson, L. Karlsson, and B. Kågström. Distributed SBP Cholesky Factorization Algorithms with Near-Optimal Scheduling[2]. *ACM Transactions on Mathematical Software*, Vol. 36, No. 2, 2009.
Paper III	L. Karlsson. Blocked In-Place Transposition with Application to Storage Format Conversion. UMINF 09.01, Department of Computing Science, Umeå University, Sweden, 2009.
Paper IV	L. Karlsson and B. Kågström. A Framework for Dynamic Node-Scheduling of Two-Sided Blocked Matrix Computations. In *Proceedings of PARA'08*, accepted for publication, 2008.

[1] Reprinted by permission of Springer-Verlag.
[2] Reprinted by permission of ACM, Inc.

Acknowledgements

I am grateful for the kind support offered by my supervisor Bo Kågström. Bo, thank you for your thorough reading of this thesis and your encouraging and constructive advice over the past couple of years. I look forward to further collaboration in the years to come. Bo is a co-author of Papers I, II, and IV, as well as a major contributor to Paper III.

I would like to thank Fred Gustavson for detailed technical discussions and for introducing me to some of the research topics in this thesis. Thank you for carefully reading our drafts, particularly all of the figures, tables, and algorithms. Several subtle mistakes and ambiguous statements would have gone unnoticed without your help. Fred is a co-author of Papers I-II.

Thanks to all the colleagues at the Department of Computing Science. Besides providing me with an excellent education you have also created an interesting and open working place with excellent technical and administrative support.

Privately, I want to thank my parents Kjell and Berit for giving me not only a healthy upbringing but never ending support in all aspects of life and throughout the years. I also want to thank my siblings John and Linnea for all the happy moments we have experienced together.

A special thanks goes to Hannah, my dearest friend and partner in life. Thank you for all the joy you bring!

This research was conducted using the resources of High Performance Computing Center North (HPC2N), and supported in part by the Faculty of Science and Technology, Umeå University, by the *Swedish Research Council* under grant VR 70625701, and by the *Swedish Foundation for Strategic Research* under the frame program grant A3 02:18.

Thank You!

Umeå, April 2009

Lars Karlsson

Contents

Chapter 1

High Performance Dense Linear Algebra Research

High Performance Computing (HPC) has expanded in recent years to include not only its traditional scientific and engineering applications but also search engines, databases, online multiplayer gaming, and more. In large part, this expansion has been made possible by the rise of the Internet, global communication, and the massive digitalization of information. Underlying many HPC applications is a small set of performance critical kernels, usually implemented as portable libraries which are tuned for specific architectures.

In this thesis, we consider a specific set of such kernels: namely dense linear algebra operations. Libraries such as BLAS [7], LAPACK [2], and ScaLAPACK [6] provide portable high performance for a large class of operations on both shared memory and distributed memory architectures. We first give a brief account to some major aspects of the research in this area with a focus on implementation issues. We then summarize the four papers that constitute this thesis. They concern various topics such as matrix storage formats and conversion, scheduling of two-sided matrix computations, and packed Cholesky factorization on distributed memory architectures.

1.1 Algorithms

The numerical algorithms offered by HPC libraries should be efficient as well as robust and numerically stable. This enables high accuracy and reliability to be obtained as fast as possible. Numerical analysts have developed mathematical theories that establish the numerical properties of many algorithms, see [21] for a collection of key results.

The numerically critical operations are typically localized to a small subset of the algorithm such as the pivoting in LU factorization and the deflation detection in eigenvalue computations. These parts require careful engineering to avoid numerical problems such as underflow/overflow and severe cancellation. However, most of the computations can be expressed as highly parallel and numerically simpler operations such as matrix multiplication and back substitution. From a numerical viewpoint, high performance implementations are often adaptations of sequential algorithms rather than completely different. Experience with high performance implementations has resulted in the development of concepts such as blocked and recursive blocked algorithms (e.g., see [2, 32, 34]), communication minimization, schedule optimization, alternative storage formats (e.g., see [25, 12, 28]), automatic tuning (e.g., see [5, 42, 40, 20]), asynchronous execution (e.g., see [10, 11, 30]), etc. These concepts are general enough to apply to many different algorithms and careful studies have furthered our understanding of their impact on performance.

There is a continuous development of new and increasingly sophisticated numerical algorithms, such as the MRRR algorithm for the symmetric tridiagonal eigenvalue problem [16], Jacobi SVD for highly accurate singular value decomposition [17, 18], a QR algorithm with aggressive early deflation and level 3 performance for the nonsymmetric eigenvalue problem [9], and a QZ algorithm which applies similar ideas to the generalized eigenvalue problem [31, 1]. Other recent novel parallel algorithms include reordering of eigenvalues in computed Schur forms [23] and the solution of matrix equations [22]. Even though new algorithms such as the ones above represent major advances in capability, many need further development before they can efficiently take advantage of the available parallel hardware.

1.2 Multicore Processors

The tremendous increase in processor performance has been linked with Moore's Law, which famously predicts an exponential increase in the number of transistors per chip. However, translating transistor count into performance has required a lot of research and innovation. Pipelines, SIMD units, superscalar and out-of-order execution have provided a large boost in performance. The clock frequency has steadily increased and with it the power consumption. Hierarchies of fast but expensive cache memories have been put in place to partially overcome the increasing gap between processor performance and memory bandwidth and latency. All of these technologies suffer from diminishing returns. Therefore, processor manufacturers have all embraced a so called multicore architecture where several independent processing units, or cores, are placed on the same chip. A multicore architecture has traditionally been reserved for special purpose hardware such as graphics cards, routers, and high-end servers. Today, this type of architecture is widely viewed as the most viable option to continue on the exponential performance development curve in the near future [3].

The HPC community is currently investigating how best to harness the power of multicore architectures and other closely related technologies such as graphics processing units and numerical accelerators. The most immediate problem is to find algorithms that can reach near peak performance on these architectures. Then comes the problem of producing and maintaining large libraries with hundreds of different algorithms and variants. The programming cost required to implement one such algorithm must be brought down to manageable levels before such a task can be undertaken. This will likely require adding abstraction layers and/or adopting new programming models [14, 36, 37].

1.3 Hybrid and Heterogeneous Distributed Memory Architectures

A supercomputer typically exploits parallelism on all levels of architecture design from pipelines, SIMD units, and multicore processors, up to high performance networks with large bisectional bandwidths. The memory in such a large system is necessarily physically distributed. However, the nodes connected by the network often have a shared memory architecture built from one or more multicore processors. We say that such a system has a hybrid architecture.

The high performance offered by graphics cards and numerical accelerators makes it economically attractive to build heterogeneous HPC systems by connecting general purpose processors to accelerators.

To attain optimal performance it is necessary to take into account the primary features of the underlying hardware. The two de-facto standard parallel programming environments are OpenMP [37] and MPI [35] which are designed for the shared memory and distributed memory architecture models, respectively. Since most HPC systems are actually hybrid architectures, several efforts have been made to combine MPI with OpenMP to improve performance, with varying results and conclusions.

The recent shift to multicore processors and the improved programmability of graphics cards have sparked renewed interest in programming models for hybrid and heterogeneous architectures. Several papers address issues related to dense linear algebra algorithms, see for example [10, 11, 41, 15].

1.4 Matrix Storage Formats

A matrix can be stored in computer memory in many different ways. The way in which a matrix is stored, i.e., how each element is mapped to a unique memory address, is called a matrix storage format (or data layout). The standard row-major (RM) and column-major (CM) formats, typically used by compilers to store multi-dimensional arrays, are preferred for their simplicity and low overhead in both direct and incremental index calculations.

The memory hierarchy designs favor locality of reference in two ways. First, since cache memories have a limited capacity, data that has not been recently used has a high probability of being evicted from the cache. Thus, temporal locality of reference increases the likelihood that the referenced data is in the cache. Second, since cache memories transfer memory in blocks (so called cache lines), typically of the order of 64 to 128 bytes long, spatial locality of reference increases the likelihood that the referenced data is found in the cache. HPC algorithms must be carefully designed to exhibit high levels of both temporal and spatial locality of reference in order to use the memory hierarchy efficiently. Algorithms that fail to do so will be memory bound and hence unable to utilize the full potential of the processor.

There is maximum spatial locality when a column-major matrix is accessed column-wise, but (for a large enough matrix) the poor spatial locality observed when the matrix is accessed row-wise may result in no cache reuse at all. The performance difference can be as large as an order of magnitude. Many alternative matrix storage formats have been proposed to alleviate this problem. These improve the spatial data locality by matching the access patterns of some particular class of algorithms. For example, block storage formats arrange submatrices (so called blocks) so that they are stored contiguously. Thus, a block can be accessed with perfect spatial locality and block formats therefore match blocked matrix algorithms well [25, 38, 19].

Recursive storage formats have also been intensively studied due to their theoretical locality-preserving properties. Recursive formats are related to space-filling curves and hence they tend to borrow their names. Examples include the U-Morton, Z-Morton, X-Morton, Hilbert, and Peano orderings [12, 4, 19]. One drawback of recursive formats is the indexing overhead, especially if the recursion is carried down to the element level. Therefore, practical recursive storage formats are hybrid and the recursion stops when the submatrices (blocks) are small enough and these are then stored in a conventional storage format.

There are recursive blocked algorithms for several important linear algebra operations (see the review in [19]). The ever smaller submatrix accesses that are generated by such algorithms amounts to a multi-level blocking with the ability to automatically adapt to deep memory hierarchies [24]. The primary benefit of recursive algorithms is an increased temporal data locality.

There are also various storage formats for band, triangular, and symmetric matrices. These have received attention due to the possibly large reduction in the required amount of memory. Packed storage formats for symmetric or triangular matrices do not store all of the redundant/zero elements. One way to store a lower triangular matrix is to store the relevant portion of each column one after the other. This format is similar to column-major and is used in the LAPACK library. However, the lack of support for this format in the level 3 BLAS has meant that packed LAPACK routines perform much worse than their full storage equivalents. Alternative storage formats such as the Square Block Packed (SBP) [27] and Rectangular Full Packed (RFP) [28] formats allow packed routines to reach level 3 performance. The RFP format has recently been included in the LAPACK library [2].

1.5 Automatic Code Generation and Tuning

Optimizing performance critical kernels for specific architectures is a difficult and time consuming task. The frequent introduction of new processors and compilers results in a continuous demand for optimization. However, the time and skill required to do the optimizations are prohibitive for all but experts. As a result, there is usally a substantial delay between the introduction of a new architecture and the availability of optimized kernels. Furthermore, the code has likely been optimized for general use and not for a user's particular needs.

These issues motivate research into automatic optimization [5, 42]. The research on compiler optimization is extensive. However, when faced with complex algorithms such as matrix computations, typical compiler optimizations do not come close to the best manually tuned implementations. There are many reasons for this, such as limited information about the code, and time restrictions; few users would tolerate a compiler that spends minutes or hours optimizing three nested loops.

A popular complement to compiler optimization, which is typically based on models and heuristics, is empirical optimization based on automatic code generation and tuning. In empirical optimization, several functionally equivalent implementations are automatically generated and empirically evaluated. Certain classes of implementations can be readily generalized to a code template from which automatic code generation can produce a multitude of variants. A search space is constructed from code generators and manually tuned implementations. Sophisticated search algorithms are required when the search space is too large to allow for an exhaustive search [42]. Examples of projects that employ automatic code generation and tuning include the PHiPAC [5] and ATLAS [42] projects (both inspired by the GEMM-based approach [33, 32]) for linear algebra operations, and Spiral [40] and FFTW [20] for signal processing and fast Fourier transforms.

Automatic tuning is useful even without code generation. Many high-level algorithms have parameters such as block sizes, thresholds, machine parameters, etc that affect performance. By automatically tuning these parameters, e.g., during installation [42] or at runtime [20], a library can adapt to different architectures and users.

1.6 Programming Models

Programming models provide a hardware abstraction layer that enables the programmer to write programs without knowing all the hardware details. The Fortran programming language is widely used in numerical applications. It provides a high level of abstraction and a sequential program execution semantics. However, extracting parallelism from a sequential program is a difficult problem indeed. Parallel programming models let the programmer specify parallelism by exposing parallel loops, independent subtasks, and other typical parallel patterns.

Graphics cards and hybrid architectures have motivated the development of new programming models such as CUDA [14] and OpenCL [36] to name just two. Many programming models have been proposed througout the years, and many more are likely to follow in the coming years due to the focus on multicore architectures. Many models target parallel programming on shared memory architectures [39, 8]. Some efforts specifically target linear algebra libraries, and many of these use some form of dynamic scheduling, see for example [41, 11, 10, 30]. The ultimate goal is to support efficient implementations on hybrid distributed memory architectures.

Chapter 2

Summary of the Papers

2.1 Paper I

Paper I [26] investigates various algorithms for packed storage Cholesky factorization on distributed memory architectures. The Rectangular Full Packed (RFP) format [28] allows reuse of existing ScaLAPACK routines [13, 6] but the performance turns out to be suboptimal for several reasons, which are highlighted in the paper. Two Cholesky factorization algorithms for matrices stored in Square Block Packed (SBP) format are also presented and evaluated. One of the implementations is similar to the right-looking algorithm used by ScaLAPACK insofar as it proceeds in stages of parallel computation and communication that do not overlap. Therefore, this algorithm, while performing better than the RFP algorithm, has similar performance problems as the ScaLAPACK full storage routine. The third algorithm (also designed for SBP) addresses this issue by using one level of lookahead to overlap communication with computation. The computational experiments indicate that the algorithm for SBP with lookahead outperforms the other considered algorithms. The RFP algorithm did not perform as well as the others but its simple Cholesky factorization implementation based on code reuse makes it attractive.

2.2 Paper II

In the second paper [27], we elaborate on the two SBP Cholesky factorization algorithms first presented in Paper I and focus on the variant with lookahead. We go into more detail on how to perform communication and computation concurrently, using asynchronous communication functions available in the Message Passing Interface (MPI) [35]. The paper emphasizes the need to use both asynchronous sends and receives in order to hide communication overhead on both ends of the communication. The global schedule was also improved by rearranging the node computations to interleave the panel factorization with the update phase. A few types of local scheduling inefficiencies were discovered by trace analysis. These inefficiencies appeared to be difficult to avoid in any static schedule and so we considered a limited dynamic scheduling. However, although the local inefficiencies could now be avoided, the global schedule was not substantially improved. This led to the development of simulations that ignored communication overhead and therefore just considered the schedule. These simulations indicate that the static schedule is near optimal in the sense that one processor is busy during virtually the entire computation while the 2D block cyclic data distribution ensures an acceptable load balance. Profiling of the implementation showed that this carries over into practice. The primary consequence of the near-optimality is that further substantial improvements to the schedule must include a different distribution of work to the nodes of the cluster. The effectiveness of the static schedule also explains why the dynamic scheduling did not give an improved global schedule.

2.3 Paper III

Paper III [29] takes a closer look at block storage formats (of which SBP is a special case), blocked in-place transposition, and its application to in-place conversion between storage formats. Although the paper does not contribute much to the theory of neither in-place transposition nor conversion, it does provide algorithm evaluations and comparisons. Matrix transposition and permutations in general can be expressed in many different ways. The paper connects some of them, such as in-place permutation, mixed radix digit permutation, and Kronecker product factorizations of the permutation matrix. The paper argues that a mixed radix number representation is an effective way to reason about blocked in-place transposition algorithms and it has direct practical applications as a useful abstraction in implementations.

2.4 Paper IV

The fourth paper [30] describes a framework for node-scheduling of distributed memory algorithms. The paper considers a model algorithm inspired by the bulge-chasing part of blocked Hessenberg QR algorithms for the nonsymmetric eigenvalue problem. The algorithms perform two-sided transformations which result in both limited parallelism and complex dependencies. We show that a straightforward implementation scales poorly and argue that this is due to a poor schedule rather than an inherent problem. The paper describes a dual wavefront algorithm that is capable of activating all processors during most of the computation, thus resulting in a more scalable algorithm. However, a key point is that implementing the dual wavefront algorithm manually as a set of nested loops with sycnhronized communication is practically infeasible. By using dynamic priority-based node-scheduling of both computation and communication on each node (which in the paper consists of a single uni-processor) the non-scalable straightforward implementation is transformed by relatively small modifications to a scalable dynamic implementation (i.e., the dual wave-front algorithm). Not addressed by the paper is the specific issues related to hybrid architectures. The paper argues that dynamic scheduling is suitable for multicore architectures, as evidenced by for example [8, 39, 11].

Chapter 3

Future Work

There are several interesting research topics related to Papers III and IV that we plan to investigate further. We will implement parallel versions of some algorithms discussed in Paper III [29] on both traditional and specialized shared memory architectures. There is also a large literature on parallel transposition algorithms and some of the proposed algorithms might be incorporated in our evaluation. The ultimate goal is a high performance portable library for conversion between different (block) matrix data layouts. Based on our results in [29], it is likely that we need to use automatic tuning and dynamic algorithm selection to reach this goal.

Paper IV is the first step in our on-going investigation of the usefulness and limits of dynamic node-scheduling on hybrid architectures. The ability to obtain radically different schedules from a sequential program is valuable, as shown in Paper IV [30] where the scalability of an algorithm is significantly enhanced by altering the schedule on each node. There are many details to work out before we are in a position to implement a high performance framework that offers enough functionality to cover a majority of the algorithmic patterns encounterd in dense linear algebra algorithms. The impact on performance of locality of reference, load balance, dependency analysis, and communication can not be ignored. The problem sizes are potentially so large that it is infeasible to do an *a priori* construction and/or analysis of the task graph. Furthermore, since some algorithms are nondeterministic, such a construction might even be impossible. The overhead associated with the dynamic scheduling is critical; if it costs too much to dynamically schedule the operations, then a pure MPI approach or MPI coupled with OpenMP or threaded BLAS would be superior. We plan to address the various issues and implement a minimalistic framework. This will allow us to estimate the inherent overheads associated with dynamic node-scheduling. There are many past and present efforts that aim to provide high level parallel programming models. Our research is orthogonal to these as it addresses the specific problems related to the implementation of a high performance runtime for dynamic node-scheduling.

Bibliography

[1] B. Adlerborn, B. Kågström, and D. Kressner. Parallel Variants of the Multishift QZ Algorithm with Advanced Deflation Techniques. In *Applied Parallel Computing: State of the Art in Scientific Computing*, Lecture Notes in Computer Science, LNCS 4699, pages 117–126. Springer, 2007.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D Sorensen. *LAPACK User's Guide (3rd ed.).* Society for Industrial and Applied Mathematics, 1999.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[4] M. Bader and C. Zenger. Cache Oblivious Matrix Multiplication Using an Element Ordering Based on a Peano Curve. *Linear Algebra and its Applications*, 417(2–3):301–313, 2006.

[5] J. Bilmes, K. Asanovic, C. W. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: A Portable High-Performance ANSI C Methodology. In *Proceedings of the International Conference on Supercomputing*, pages 340–347, Vienna, 1997.

[6] 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 User's Guide.* Society for Industrial and Applied Mathematics, 1997.

[7] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Software*, 28(2):135–151, 2002.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[9] K. Braman, R. Byers, and R. Mathias. The Multishift QR Algorithm. Part I: Maintaining Well-Focused Shifts and Level 3 Performance. *SIAM J. Matrix Anal. Appl.*, 23:929–947, 2001.

[10] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. Technical Report UT-CS-07-600, Innovative Computing Laboratory, University of Tennessee, September 2007.

[11] E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. SuperMatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures. In *SPAA '07: Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 2007.

[12] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive Array Layouts and Fast Parallel Matrix Multiplication. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231. ACM, 1999.

[13] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming*, 5(3):173–184, 1996.

[14] CUDA Zone – The Resource for CUDA Developers. http://www.nvidia.com/object/cuda_home.html. (January, 2009).

[15] J. Demmel and V. Volkov. LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[16] I. S. Dhillon, B. N. Parlett, and C. Vömel. The Design and Implementation of the MRRR Algorithm. *ACM Trans. Math. Software*, 32(4):533–560, 2006.

[17] Z. Drmac and K. Veselic. New Fast and Accurate Jacobi SVD Algorithm: I. *SIAM J. Matrix Anal. Appl.*, 29(4):1322–1342, 2007.

[18] Z. Drmac and K. Veselic. New Fast and Accurate Jacobi SVD Algorithm: II. *SIAM J. Matrix Anal. Appl.*, 29(4):1343–1362, 2007.

[19] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. *SIAM Review*, 46(1):3–45, 2004.

[20] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):216–231, 2005.

[21] G. H. Golub and C. F. van Loan. *Matrix Computations (3rd ed.).* Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[22] R. Granat, I. Jonsson, and B. Kågström. RECSY and SCASY Library Software: Recursive Blocked and Parallel Algorithms for Sylvester-Type Matrix Equations with Some Applications. In R. Ciegis et al., editor, *Parallel Scientific Computing–Advances and Applications*, volume 27, pages 3–24. Springer Optimization and Its Applications, 2009.

[23] R. Granat, B. Kågström, and D. Kressner. Parallel Eigenvalue Reordering in Real Schur Forms. *Concurrency and Computation: Practice and Experience* (to appear), 2009.

[24] F. G. Gustavson. Recursion Leads to Automatic Variable Blocking for Dense Linear Algebra Algorithms. *IBM Journal of Research and Development*, 41, 1997.

[25] F. G. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Recursive Blocked Data Formats and BLAS's for Dense Linear Algebra Algorithms. In *Applied Parallel Computing, PARA'98*, Lecture Notes in Computer Science, LNCS 1541, pages 195–206. Springer, 1998.

[26] F. G. Gustavson, L. Karlsson, and B. Kågström. Three Algorithms for Cholesky Factorization on Distributed Memory Using Packed Storage. In *Applied Parallel Computing: State of the Art in Scientific Computing*, Lecture Notes in Computer Science, LNCS 4699, pages 550–559. Springer, 2007.

[27] F. G. Gustavson, L. Karlsson, and B. Kågström. Distributed SBP Cholesky Factorization Algorithms with Near-Optimal Scheduling. *ACM Trans. Math. Software*, 36(2), 2009.

[28] F. G. Gustavson and J. Wasniewski. Rectangular Full Packed Format for LAPACK Algorithms Timings on Several Computers. In *Applied Parallel Computing: State of the Art in Scientific Computing*, Lecture Notes in Computer Science, LNCS 4699, pages 570–579. Springer, 2007.

[29] L. Karlsson. Blocked In-Place Transposition with Application to Storage Format Conversion. Technical Report UMINF 09.01, Department of Computing Science, Umeå University, SE-901 87 Umeå, Sweden, January 2009.

[30] L. Karlsson and B. Kågström. A Framework for Dynamic Node-Scheduling of Two-Sided Blocked Matrix Computations. In *Proceedings of PARA '08*, Lecture Notes in Computer Science. Springer, accepted 2008.

[31] B. Kågström and D. Kressner. Multishift Variants of the QZ Algorithm with Aggressive Early Deflation. *SIAM J. Matrix Anal. Appl.*, 29(1):199–227, 2006.

[32] B. Kågström, P. Ling, and C. Van Loan. Algorithm 784: GEMM-Based Level 3 BLAS: Portability and Optimization Issues. *ACM Trans. Math. Software*, 24(3):303–316, 1998.

[33] B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS: High Performance Model Implementations and Performance Evaluation Benchmark. *ACM Trans. Math. Software*, 24:268–302, 1998.

[34] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. *SIGARCH Computer Architecture News*, 19(2):63–74, 1991.

[35] Message Passing Interface (MPI) Forum. http://www.mpi-forum.org/. (January, 2009).

[36] OpenCL. http://www.khronos.org/opencl/. (January, 2009).

[37] OpenMP.org. http://openmp.org/wp/. (January, 2009).

[38] N. Park, B. Hong, and V. K. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, 2003.

[39] J. M. Perez, R. M. Badia, and J. Labarta. A Flexible and Portable Programming Model for SMP and Multi-cores. Technical Report 03/2007, Barcelona Supercomputing Center, 2007.

[40] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, T. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[41] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. van de Geijn. Solving Dense Linear Algebra Problems on Platforms with Multiple Hardware Accelerators. In *Proceedings of the 2009 ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Raleigh, North Carolina, February 2009. To appear.

[42] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001.

I

Paper I

Three Algorithms for Cholesky Factorization on Distributed Memory using Packed Storage[*]

Fred G. Gustavson[2], Lars Karlsson[1], and Bo Kågström[1]

[1]*Department of Computing Science and HPC2N, Umeå University*
SE-901 87 Umeå, Sweden
{larsk, bokg}@cs.umu.se

[2]*IBM T. J. Watson Research Center,*
Yorktown Heights, NY 10598, USA.

Abstract: We present three algorithms for Cholesky factorization using minimum block storage for a distributed memory (DM) environment. One of the distributed square block packed (SBP) format algorithms performs similar to ScaLAPACK's full storage routine `PDPOTRF`, and our algorithm with iteration overlapping typically outperforms it by 15–50% for small and medium sized matrices. By storing the blocks contiguously, we get better performing BLAS operations. Our DM algorithms are not sensitive to cache conflicts and thus give smooth and predictable performance. We also investigate the intricacies of using rectangular full packed (RFP) format with ScaLAPACK routines and point out some advantages and drawbacks.

Key words: Cholesky factorization, distributed memory, packed storage, square block packed, rectangular full packed.

[*] Copyright 2007 of *Springer-Verlag*. Reprinted with permission. All rights reserved.

Three Algorithms for Cholesky Factorization on Distributed Memory using Packed Storage

Fred G. Gustavson[1,2], Lars Karlsson[2], and Bo Kågström[2]

[1] IBM's T. J. Watson Research Center,
Yorktown Heights, NY 10598, USA, fg2@us.ibm.com
[2] Department of Computing Science and HPC2N, Umeå University,
S-901 87 Umeå, Sweden, {larsk, bokg}@cs.umu.se

Abstract. We present three algorithms for Cholesky factorization using minimum block storage for a distributed memory (DM) environment. One of the distributed square block packed (SBP) format algorithms performs similar to ScaLAPACK `PDPOTRF`, and our algorithm with iteration overlapping typically outperforms it by 15–50% for small and medium sized matrices. By storing the blocks contiguously, we get better performing BLAS operations. Our DM algorithms are not sensitive to cache conflicts and thus give smooth and predictable performance. We also investigate the intricacies of using rectangular full packed (RFP) format with ScaLAPACK routines and point out some advantages and drawbacks.

1 Introduction

Dense linear algebra routines that are implemented in a distributed memory environment typically use a 2D block cyclic layout (BCL), with ScaLAPACK being one example of a library that uses BCL for all routines [3]. A BCL can provide effective load balance for many algorithms. The mapping of matrix elements to processors does not prescribe how they are later stored on each processor. The approach taken by the ScaLAPACK library is to store each elementary block as a submatrix of a column major 2D array (standard Fortran array) [3]. Another approach is to store each elementary block contiguously, for example as a column major block 2D array.

Storing elementary blocks contiguously has at least three advantages. They will map very well into L1 cache and level 3 operations involving such blocks will therefore tend to achieve high performance and minimize memory traffic. Another benefit is that moving a block can be done by one contiguous memory transfer. In this contribution we use *square elementary blocks* (called a square block, or SB) to store the local matrix. Furthermore, we store only the triangular part of the block matrix to achieve minimum block storage for symmetric matrices. We call this *square block packed* (SBP) format.

We identify an inefficiency in straightforward data parallel implementations, e.g., the implementation of the Cholesky factorization in ScaLAPACK (routine `PDPOTRF`) and develop an iteration overlapping data parallel implementation which removes much of the idling and thus decreases execution time.

2 Near Minimal Storage in a Serial Environment

A recently proposed format for storing triangular or symmetric matrices is called *rectangular full packed* (RFP) (see [8] for details). This format takes many slightly different forms. Figure 1 illustrates a lower triangular matrix. The matrix is

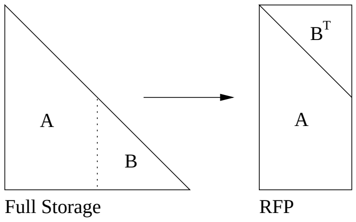


Fig. 1. Illustration of rectangular full packed format

partitioned into two submatrices A and B. The triangular matrix B^T is merged along the diagonal with A. As can be seen, this new matrix can be stored as a standard full format rectangular array with no waste of memory.

Another format for near minimal storage is a generalization of a standard column major format. The matrix is divided into square blocks and the format is based on storing each such block in a contiguous memory area. The blocks can then be stored for example in either a row or column major ordering. Figure 2

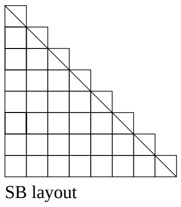


Fig. 2. Illustration of square block packed format

illustrates the square blocks. The elements above the diagonal of the diagonal blocks are wasted storage. By picking the block size to one, we see that we get either the standard row or column major format. For details on this format see [7].

3 Minimum Block Storage in a Distributed Environment

In this section we describe how RFP and SBP can be used in a distributed memory environment. We show how both approaches give a nearly minimum block storage.

3.1 A Distributed SBP Algorithm for Cholesky Factorization

In this contribution we consider the blocked algorithmic variant of Cholesky factorization described in Algorithm 1. We note that this algorithmic variant is used in ScaLAPACK [4]. In a distributed environment with a 2D block cyclic

Algorithm 1 Standard blocked Cholesky factorization

```
1: for each panel left to right do
2:     Partition $A = \begin{bmatrix} A_{11} & \\ A_{21} & A_{22} \end{bmatrix}$, where $A_{11}$ is NB×NB
3:     Factorize $A_{11} = LL^T$ using unblocked algorithm
4:     Update panel $A_{21} := A_{21}L^{-T}$ using triangular solver
5:     Update trailing matrix $A_{22} := A_{22} - A_{21}A_{21}^T$ using symmetric rank-$k$ update
6:     Continue with $A = A_{22}$
7: end for
```

layout with block size NB×NB, block A_{11} resides on one processor, block A_{21} on one processor column, and A_{22} generally resides on all processors. By using parallel triangular solve and symmetric rank-k update routines Algorithm 1 will achieve scalable performance due to good load balance and because most of the computation is in step 5 which is easy to parallelize. However, steps 3 and 4 do not utilize all processors effectively. One variant of this algorithm is to start the next iteration before the current iteration has finished step 5 (see [7] for more details). This is possible by noting that the first updated column panel of the new pivot from step 5 will be used as the only input for step 3 and 4 of the next iteration.

A major problem with a straightforward parallel implementation of Algorithm 1 is the idle time introduced when processors implicitly synchronize after each iteration. This idle time is caused both by slight load imbalances and the work in steps 3 and 4 that are not performed on all processors. By using the iteration overlapping algorithm this idle time will be eliminated if the communication of data between steps 3 and 4 can be carried out while still doing useful work in updates.

The data dependencies in Algorithm 1 are simple. Output from step 3 is input for step 4 whose output in turn is input for step 5. As for the first dependency a column broadcast is all that is needed. The second dependency requires a somewhat more complicated communication pattern and is now described briefly. All subblocks of A_{21} are broadcasted along the processor rows. Once a subblock of

A_{21} reaches the processor holding the diagonal block of that row it is broadcasted along its processor column. One can show that after this, each processor holds the blocks of A_{21} and A_{21}^T that it needs for step 5. In our implementation these blocks are stored in two block buffer vectors W and S, where W (for West border vector) holds blocks of A_{21} and S (for South border vector) holds blocks of A_{21}^T.

We have studied how overlapping two successive pivot steps can affect the performance of our parallel implementation. Our implementation is described in Algorithm 2. The overlapping in Algorithm 2 happens during the execution

Algorithm 2 Cholesky with iteration overlap

```
 1: for each panel left to right do
 2:   Partition global $A = \begin{bmatrix} A_{11} & \\ A_{21} & A_{22} \end{bmatrix}$, where $A_{11}$ is NB×NB
 3:   if process holds $A_{11}$ then
 4:     Factorize $A_{11} = LL^T$ using serial algorithm
 5:     Column broadcast the block $L$
 6:   end if
 7:   if process column holds $A_{21}$ then
 8:     Receive the block $L$
 9:     Partition $S_1 = \begin{bmatrix} S_A & S_B \end{bmatrix}$, where $S_A$ is NB×NB
10:     Update $A_{21} := A_{21} - W_1 S_A$
11:     Scale $A_{21} := A_{21} L^{-T}$
12:     Start communication of $A_{21}$ using buffers $W_2$ and $S_2$ (sender)
13:     Update $A_{22} := A_{22} - W_1 S_B$
14:   else {all other process columns}
15:     Start communication of $A_{21}$ using buffers $W_2$ and $S_2$ (receiver)
16:     Update $A := A - W_1 S_1$
17:   end if
18:   Move (symbolically) $W_1 := W_2$ and $S_1 := S_2$ {there is no data movement}
19: end for
```

of steps 10 to 13. Taken together, steps 10 and 13 perform a complete update. The execution order of the straightforward algorithm would put steps 10 and 13 together, and step 11 after both. Executing step 11 before 13 allows the communication needed for the subsequent update to take place during the update in step 13 (and while the other processors execute step 16).

In Figure 3, we illustrate by example how our local matrices are stored in practice. The blocks are stored columnwise in a one-dimensional block vector indexed by a column pointer (CP) array. The entries in CP are pointers to the first block of each block column.

Figure 4 shows how the two sets of buffers are used in Algorithm 2. The light shaded blocks are those used for the update of iteration i. The darker shaded blocks are those computed during iteration i for use in iteration $i + 1$. After the panel factorization the communication algorithm is started and it will broadcast

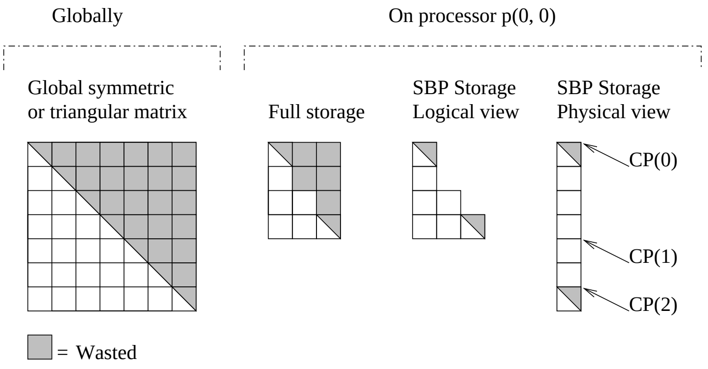


Fig. 3. Illustration of how a 7×7 block global matrix is laid out on a 2×3 mesh in SBP format and addressed with its column pointer (CP) array. The full size of the global matrix is `7NB`×`7NB`.

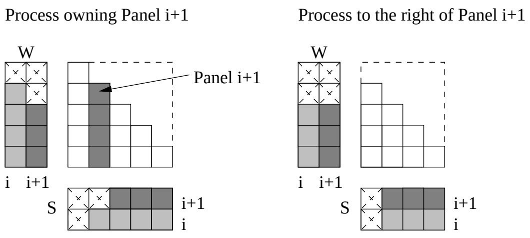


Fig. 4. Data layout for the SBP with double sets of W and S border vectors.

the panel and its transpose to all processors with this data stored in the second set of buffers. While this communication takes place, the first set of buffers is used to finish the update of iteration i.

3.2 A Distributed RFP Algorithm for Cholesky Factorization

Because of the good performance achievable with RFP format in a serial environment (see [9]) we investigated its extension to parallel environments via using ScaLAPACK and PBLAS. Algorithm 3 gives the details of the RFP Cholesky algorithm. The limitations of PBLAS and ScaLAPACK do not generally allow matrices to begin inside an elementary block; each submatrix must be block aligned. Therefore, we use the RFP format on the block level, introducing some wasted storage and thus achieve minimum block storage while still being able to use RFP with existing routines.

The RFP format could be used with an algorithm similar to the one we used with SBP. Such an RFP algorithm would probably achieve similar performance to the SBP algorithm so we did not develop any implementation of it.

Algorithm 3 RFP Cholesky with ScaLAPACK/PBLAS routines

```
1: Matrix $A$ is in RFP format: $A = \begin{bmatrix} A_{11} \backslash A_{22}^T \\ A_{21} \end{bmatrix}$
2: Factor $A_{11} = LL^T$ using ScaLAPACK routine PDPOTRF
3: Update panel $A_{21} := A_{21}L^{-T}$ using PBLAS routine PDTRSM
4: Update trailing matrix $A_{22} := A_{22} - A_{21}A_{21}^T$ using PBLAS routine PDSYRK
5: Factor $A_{22} = LL^T$ using ScaLAPACK routine PDPOTRF
```

4 Related Work on DM Cholesky Factorization

We briefly discuss other packed storage schemes for DM environments.

D'Azevedo and Dongarra suggested in 1997 a storage scheme where the elementary blocks are mapped to the same processor as in the full storage case, but only the non-redundant blocks are stored [6]. Each block column is stored as a submatrix the same way as it would in full storage. The result is that each block column is a regular ScaLAPACK matrix and can be used as such. Note that the blocks will be mapped to the same processors as the SBP format, but the local processor storage layout is different. Benefits include routine reuse via PBLAS and ScaLAPACK routines. However, some new PBLAS routines seem to be required to handle the packed storage [6]. Furthermore, their results indicate that the performance varies wildly with input, making performance extrapolation difficult.

Recently, Marc Baboulin et al. presented a storage scheme which uses relatively large square blocks consisting of at least LCM(p, q)[3] elementary blocks [2]. This format also supports code reuse via PBLAS and ScaLAPACK. The granularity is limited to the distributed block size, which means less possibility to save memory. For the Cholesky factorization routines, the chosen block sizes for performance measurements were between 1024 and 10240. This resulted in a departure from their minimum storage by as much as 7–13%. Using their minimum allowed distributed block size would bring this percentage down to about 1–3% but at the cost of longer execution times.

5 Performance Results and Comparison

In this section we give some performance related results. We compare RFP, SBP and ScaLAPACK routines and analyze the differences that we observed.

All tests were performed on the *Sarek* cluster at HPC2N. It consists of 190 HP DL145 nodes, with dual AMD Opteron 248 (2.2GHz) processor and 8 GB memory per node. The AMD Opteron 248 processor has a 64 kB instruction and 64 kB data L1 Cache (2-way associative) and a 1024 kB unified L2 Cache (16-way associative). The cluster's operating system is Debian GNU/Linux 3.1 and we used Goto BLAS 0.94 throughout.

[3] The least common multiple of the integers a and b (written LCM(a, b)) is the smallest integer that is a multiple of both a and b.

Table 1. Execution times for PDPOTRF and the SBP algorithm with iteration overlap for various square grid sizes. The block size NB is set to 100.

N	**2x2**	**3x3**	**4x4**	**5x5**	**6x6**	**7x7**
4000	2.13/0.86	1.48/0.63	1.04/0.66	0.79/0.68	0.63/0.64	0.57/0.65
8000	14.80/0.92	8.29/0.80	5.33/0.79	3.97/0.77	3.15/0.71	2.64/0.73
12000		25.20/0.83	16.30/0.80	10.90/0.84	8.27/0.80	7.11/0.78
16000		57.30/0.84	34.50/0.85	24.00/0.85	18.30/0.80	13.90/0.85
20000			65.00/0.85	43.90/0.86	33.00/0.81	25.90/0.84
24000					53.90/0.84	42.30/0.85

Table 1 shows selected times for both PDPOTRF and the SBP algorithm with iteration overlap. Each cell has the form X/y, where X is the time (in seconds) of the PDPOTRF routine and y = Y/X, where Y is the time for the SBP algorithm. The same block size was used for both implementations. We identify two trends. First of all, the relative gain by overlapping increases with the number of processors since the idle time is introduced on the entire mesh. The bigger the mesh the more idle time we can remove by overlapping. Second, the relative gain decreases with increasing problem sizes. This is expected because the dominant operation is the trailing matrix update (with $\mathcal{O}\left(N^3\right)$ flops) whereas the operations causing idle time (the panel factorization) make up for only $\mathcal{O}\left(N^2\right)$ flops.

Table 2. Execution time for PDPOTRF and the RFP algorithm using ScaLAPACK routines for various grid sizes.

N	**2x2**	**3x3**	**4x4**	**5x5**	**6x6**	**7x7**
4000	2.13/1.26	1.48/1.16	1.04/1.18	0.79/1.44	0.63/1.42	0.58/1.34
8000	14.80/*1.48*	8.29/1.25	5.33/1.23	3.97/1.37	3.15/1.33	2.64/1.33
12000		25.20/*1.41*	16.30/1.14	10.90/1.34	8.27/1.33	7.11/1.29
16000		57.30/1.16	34.50/*1.34*	24.00/1.28	18.30/1.20	13.90/1.34
20000			65.00/1.13	43.90/*1.40*	33.00/1.22	25.90/1.25

Table 2 is similar to Table 1 but shows selected times for PDPOTRF and our RFP algorithm which uses four calls to ScaLAPACK/PBLAS routines. Each cell has the form X/y, where X is the time (in seconds) of the PDPOTRF routine and y = Y/X, where Y is the time for the RFP algorithm. As can be seen from this table the RFP algorithm has typically a 10–30% longer execution time. By tracing the execution of the algorithm we found two substantial causes for this overhead. The performance of the BLAS operations issued by the RFP algorithm was less efficient than was typical for the other algorithms we tested. Moreover, there are more synchronization points in the RFP algorithm due to the two ScaLAPACK and two PBLAS calls on problems half the size. This amplifies the communication overhead and load imbalance. Taken together, this would probably explain most of the time differences we observed. One interesting detail to note in Table 2 is that when the local matrix dimension is 4000 the RFP algorithm experienced a dramatic loss in performance (emphasized by italics in Table 2). This is caused by a cache effect because the leading dimension is actually 4100 which is close to $2^{12} = 4096$; also the L1 cache on Sarek is only 2-way set associative.

The block size mainly affects performance of the BLAS operations and the load balance. Larger blocks tend to give good BLAS performance but less load balance. For the SBP algorithm the block size is intimately related to BLAS performance because then all `GEMM` calls are on matrices of order `NB`. The ScaLAPACK algorithm is less dependent on the block size because of the fewer and larger PBLAS operations. Table 3 gives an idea of how the block size relates to

Table 3. Impact of block size on performance (measured in Gflops/s per processor) for ScaLAPACK `PDPOTRF` and our overlapping SBP algorithm.

NB	PDPOTRF	Overlapping
25	2.08	1.91
50	2.12	2.57
75	2.09	2.75
100	2.15	3.04
125	2.24	3.06
150	2.13	3.04

performance for both of these algorithms. The processor mesh was 2×3 and the order of the matrix was `N=6000`. On Sarek we see that when we approach a block size of 100 we get close to optimal performance, whereas the block size does not matter much for the ScaLAPACK routine. The gap in performance between the two routines is mainly due to less idling in the overlapping routine.

Finally, we note that our overlapping SBP algorithm could be modified so that it updates first and factorizes the next panel afterwards. This makes the algorithm essentially equal to the straightforward implementation but with a different data format. We implemented this variant too and found that as expected it gave performance nearly identical to the ScaLAPACK algorithm.

6 Future Work

We outline some future directions of development. Our overlapping algorithm relies on the idea that the task of trailing matrix update can be divided into two tasks: the first panel on the column of processors holding the pivot and the rest of the panels on all processors. This allows us to have two iterations on the same processor, but three is not possible. A solution is to further divide the tasks. The trailing matrix update could for example be divided into one task for each block column. Instead of waiting for data it now becomes attractive to do smaller tasks instead. The order of the tasks thus becomes non-deterministic because it would depend on processor interactions. To get a clean implementation it might be necessary to use a style reminiscent of a work pool.

The overlapping algorithm relies heavily on the interleaving of communication and updates. One consequence of the overlapping is that more workspace is needed. In general each ongoing iteration will require its own W and S buffer. It is preferable to have many iterations ongoing because in that way more work is kept at each processor and chances for idling will get reduced. The concept of lookahead in factorization algorithms has been addressed several times (cf. [1, 5, 7]) and recently in [10]. The emphasis of the latter contribution is that a dynamic lookahead is most appropriate. A large lookahead is not feasible in a DM environment because of the large workspace required. Setting a fixed cap (or dynamic relative to a fixed workspace) on the number of iterations may be a feasible solution.

Our work provides an argument for the inclusion of *nonblocking collective* communication routines in communication libraries. The de-facto industry standard MPI has substantial support for nonblocking point-to-point communication but collectives are all blocking. Our implementation emulates nonblocking collectives by repeatedly testing for individual completion of nonblocking point-to-point operations. This complicates the code and probably comes at a higher cost than would have been the case if nonblocking collectives existed as part of the library.

7 Conclusion

We have implemented and compared three algorithms and data formats for minimum block storage in distributed memory environments using a 2D block cyclic data layout.

In a serial environment, the RFP format is an attractive choice [9]. However, the straightforward generalization of serial RFP algorithms has some weaknesses.

The SBP format was implemented and tested with two algorithm variants. One resembles ScaLAPACK's `PDPOTRF` but makes no use of PBLAS or ScaLAPACK routines, and one overlaps iterations. We have demonstrated that performance at least as good as the ScaLAPACK algorithm is attainable, and for the overlapping variant far better performance, especially for small and medium sized matrices, was achieved.

The ideas that we explored in this work can be applied to many other algorithms as well. Two examples very similar to the Cholesky factorization are the LU and QR factorizations.

Acknowledgements. This research was conducted using the resources of the High Performance Computing Center North (HPC2N). Financial support has been provided by the *Swedish Research Council* under grant VR 621-2001-3284 and by the *Swedish Foundation for Strategic Research* under grant A3 02:128.

References

1. R. C. Agarwal and F. G. Gustavson. A parallel implementation of matrix multiplication and LU factorization on the IBM 3090. In M. Wright, editor, *Aspects of Computation on Asynchronous and Parallel Processors*, pages 217–221. IFIP, North-Holland, Amsterdam, 1989.
2. M. Baboulin, L. Giraud, S. Gratton, and J. Langou. A distributed packed storage for large parallel calculations. Technical Report TR/PA/05/30, CERFACS, Toulouse, France, 2005.
3. L. S. Blackford et al. *ScaLAPACK user's guide.* SIAM Publications, 1997.
4. J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5(3):173–184, Fall 1996.
5. K. Dackland, E. Elmroth, and B. Kågström. A ring–oriented approach for block matrix factorizations on shared and distributed memory architectures. In R.F. Sincovec et al, editor, *SIAM Conference on Parallel Processing for Scientific Computing*, pages 330–338. SIAM Publications, 1993.
6. E. D'Azevedo and J. Dongarra. Packed storage extension for ScaLAPACK. Technical Report UT-CS-98-385, 1998.
7. F. Gustavson. Algorithm compiler architecture interaction relative to dense linear algebra. Technical Report RC 23715, IBM Thomas J. Watson Research Center, September 2005.
8. F. Gustavson. New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms. In J. Dongarra, K. Madsen, and J. Wasniewski, editors, *PARA 2004, Applied Parallel Computing, State of the Art in Scientific Computing*, volume 3752 of Lecture Notes in Computer Science (LNCS), pages 11–20. Springer, 2006.
9. F. Gustavson and J. Wasniewski. LAPACK Cholesky routines in rectangular full packed format. In *PARA 2006, Workshop on State-of-the-Art in Scientific and Parallel Computing*, *Lecture Notes in Computer Science (LNCS).* Springer, 2006. To Appear.
10. J. Kurzak and J. J. Dongarra. Pipelined shared memory implementation of linear algebra routines with arbitrary lookahead – LU, Cholesky, QR. In *PARA 2006, Workshop on State-of-the-Art in Scientific and Parallel Computing*, *Lecture Notes in Computer Science (LNCS).* Springer, 2006. To Appear.

II

Paper II

Distributed SBP Cholesky Factorization Algorithms with Near-Optimal Scheduling*

Fred G. Gustavson[2], Lars Karlsson[1], and Bo Kågström[1]

[1] *Department of Computing Science and HPC2N, Umeå University*
SE-901 87 Umeå, Sweden
{larsk, bokg}@cs.umu.se

[2] *IBM T. J. Watson Research Center,*
Yorktown Heights, NY 10598, USA.

Abstract: The minimal block storage Distributed Square Block Packed (DSBP) format for distributed memory computing on symmetric and triangular matrices is presented. Three algorithm variants (Basic, Static, and Dynamic) of the blocked right-looking Cholesky factorization are designed for the DSBP format, implemented, and evaluated. On our target machine, all variants outperform standard full storage implementations while saving almost half the storage. Communication overhead is shown to be virtually eliminated by the Static and Dynamic variants, both of which take advantage of hardware parallelism to hide communication costs. The Basic variant is shown to yield comparable or slightly better performance than full storage ScaLAPACK routine `PDPOTRF` while clearly outperformed by both Static and Dynamic. Models of execution assuming zero communication costs and overhead are developed. For medium and larger sized problems the Static schedule is near-optimal on our target machine based on comparisons with these models and measurements of synchronization overhead.

Key words: Cholesky factorization, distributed memory, packed storage, square block packed.

* Copyright 2009 of *ACM, Inc*. Reprinted with permission. All rights reserved.

Distributed SBP Cholesky Factorization Algorithms with Near-Optimal Scheduling

FRED G. GUSTAVSON
IBM T.J. Watson Research Center and Umeå University
and
LARS KARLSSON and BO KÅGSTRÖM
Umeå University

The minimal block storage Distributed Square Block Packed (DSBP) format for distributed memory computing on symmetric and triangular matrices is presented. Three algorithm variants (Basic, Static, and Dynamic) of the blocked right-looking Cholesky factorization are designed for the DSBP format, implemented, and evaluated. On our target machine, all variants outperform standard full-storage implementations while saving almost half the storage. Communication overhead is shown to be virtually eliminated by the Static and Dynamic variants, both of which take advantage of hardware parallelism to hide communication costs. The Basic variant is shown to yield comparable or slightly better performance than the full-storage ScaLAPACK routine PDPOTRF while clearly outperformed by both Static and Dynamic. Models of execution assuming zero communication costs and overhead are developed. For medium- and larger-sized problems, the Static schedule is near optimal on our target machine based on comparisons with these models and measurements of synchronization overhead.

Categories and Subject Descriptors: F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithm and Problems—*Computations on matrices*; G1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Linear Systems (direct and iterative methods)*; G.4 [**Mathematical Software**]: Algorithm design and analysis, reliability and robustness

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Real symmetric matrices, positive definite matrices, Cholesky factorization, distributed square block format, packed storage, parallel computing, parallel algorithms

This research was conducted using the resources of the High Performance Computing Center North (HPC2N). Financial support was provided by the *Swedish Research Council* under grant VR 621-2001-3284 and by the *Swedish Foundation for Strategic Research* under grant A3 02:128.
Authors' addresses: F. G. Gustavson, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA; email: fg2935@hotmail.com; L. Karlsson and B. Kågström, Department of Computing Science and HPC2N, Umeå University, SE-901 87, UMEÅ; email: {larsk,bokg}@cs.umu.se.

DOI 10.1145/1499096.1499100 http://doi.acm.org/ 10.1145/1499096.1499100

1. INTRODUCTION

Cholesky factorization is a special case of Gaussian elimination for symmetric positive definite matrices. Algorithms for Cholesky factorization can save roughly half of the floating point operations, use half of the memory, and since A is positive definite there is no need for pivoting [Golub and van Loan 1996]. In the literature there has been a great deal of interest in sparse parallel Cholesky algorithms but here we only consider dense matrices. Although not as ubiquitous as large sparse Cholesky factorization, there are applications of large dense Cholesky factorization, for example when solving linear least squares problems with the method of normal equations [Baboulin et al. 2005a].

A number of different parallel dense Cholesky factorization algorithms have been designed and implemented on a wide range of computer architectures. Three of the earliest ones are on use of systolic arrays [Brent and Luk 1982], data-flow [O'Leary and Stewart 1985] and distributed memory [Geist and Heath 1985]. In regard to scheduling, [Gerasoulis and Nelken 1989] considers parallel Cholesky factorization on MIMD architectures. Today, distributed memory (DM) computing with message passing through the MPI interface is the de-facto standard for parallel large-scale computations. Scalable, portable routines for Cholesky factorization adapted to DM architectures can be found in ScaLAPACK [Choi et al. 1996] and PLAPACK [van de Geijn 1997] to name but two.

However, few attempts have been made at storing the symmetric matrix in a packed format. Previous approaches have often looked at packed routines as special cases and have not been able to deliver the same level of performance as full storage routines. Here, we present the Distributed Square Block Packed (DSBP) format which generalizes both standard packed and full storage [Gustavson et al. 2007a]. This format allows a unification of packed and full storage routines into a single high performance implementation.

In [Gustavson et al. 2007b], we first examined the feasibility of the DSBP format. We demonstrated that performance better than the ScaLAPACK full storage Cholesky factorization routine `PDPOTRF` ([Choi et al. 1996]) is achievable for a packed storage routine. In this contribution, three DSBP algorithm variants of parallel packed Cholesky factorization are developed. The first variant (Basic) is a right-looking Cholesky factorization algorithm and it is comparable with `PDPOTRF` but operates on a matrix in DSBP format. The second variant (Static) takes advantage of hardware parallelism to overlap communication with computation via use of look-ahead and non-blocking communication primitives. The third variant (Dynamic) uses a more flexible scheduling on DM nodes and is important on hybrid systems with SMP or multi-core nodes. Our goal is to reach an optimal task schedule on the nodes and perfect overlap between communication and computation (see [Agarwal et al. 1994] for earlier such results for parallel matrix multiplication). We do this by a technique called algorithmic look-ahead [Agarwal and Gustavson 1988; 1989; Dackland et al. 1992; Dackland et al. 1993; Strazdins 1998] which reorders the outer loop to perform the next panel factorization before the current trailing matrix update is complete.

Earlier contributions on packed distributed storage of symmetric matrices include work by [D'Azevedo and Dongarra 1998] where a packed lower triangular matrix is represented as a collection of block columns, each using standard column major storage format. Their main focus is to encapsulate the packed storage within the abstraction framework of the ScaLAPACK building blocks such as the PBLAS. Performance figures are presented which show respectable but slightly worse performance compared with full storage ScaLAPACK routines.

In [Baboulin et al. 2005b], a secondary blocking level is introduced. An *elementary block* corresponds to a distribution block of a two-dimensional square block-cyclic distribution. A *grid block* is a $P_r \times P_c$ block matrix of elementary blocks. A *distributed block* is a square matrix of elementary blocks. Each dimension of a distributed block consists of an integral multiple of $\mathrm{lcm}(P_r, P_c)$ elementary blocks. Thus, a distributed block is made up of a set of complete grid blocks and as a result the elementary blocks are perfectly distributed and all distributed blocks have the same distribution. The packed Cholesky implementation reuses ScaLAPACK and PBLAS on the level of distributed blocks. Performance is for the most part slightly worse than for the full storage ScaLAPACK routine, possibly due to extra communication, library overhead, and a reduced degree of concurrency. We remark that the storage requirement is larger than that of the DSBP format.

2. ORGANIZATION AND NOTATION

The rest of the paper is organized as follows. Section 3 describes the DSBP format for memory efficient storage and high-performance DM implementations. In Section 4, the three Cholesky algorithm variants are described along with a brief discussion of the details of the communication algorithms. The MPI interface provides the possibility to express overlap of communication with computation at the application level, but to what extent overlap is actually exploited is highly machine and software specific. In Section 5, we therefore present an evaluation of the target machine's overlap capabilities. This is crucial for interpreting the performance results given in Section 6. Scalability is examined in Section 7. Finally, we conclude with a summary of our major findings and outline future work in Section 8.

Processors are arranged in a logical $P_r \times P_c$ mesh with each processor having 2-dimensional coordinates (p, q) with $p \in \{0, \ldots, P_r - 1\}$ and $q \in \{0, \ldots, P_c - 1\}$. The matrix A being factored is of size $N \times N$. It is partitioned into

$$N_b = \left\lceil \frac{N}{n_b} \right\rceil$$

submatrices of order n_b with padding of the possibly incomplete last block row and column. Padding simplifies the DSBP addressing scheme; see Section 3 below.

The following LAPACK/BLAS names are used in the text:

—`POTRF`: computes the Cholesky factorization $A = LL^T$.

—`TRSM`: solves a triangular system of equations with multiple right-hand sides.

—`GEMM`: performs a matrix multiply and add update.

—`SYRK`: performs a symmetric rank-k update.

It is important to distinguish the different notions of blocking in DM computing. A block cyclic layout (BCL) has a *distribution block size*, which, in our case, is square. A blocked algorithm usually has one, and sometimes more, *algorithmic* *block sizes.* For the combination of a BCL and a blocked algorithm it is common to use the same block size for both the distribution and the algorithm (typified by ScaLAPACK) and referred to as *distribution blocking.* However, a more general approach decouples the two block sizes and this is often called *algorithmic blocking.* This allows for better computational load balance since the distribution block size can be reduced. Processor interactions are often more frequent when using algorithmic blocking. For example, the diagonal block factorization (see Section 4) is a local operation when using distribution blocking while it is a collective operation when using algorithmic blocking. We do not consider algorithmic blocking in conjunction with DSBP since they may have contradicting goals (improved load balance versus reduced data movement).

3. DISTRIBUTED SQUARE BLOCK PACKED FORMAT

In this section, we give a self-contained discussion on the Square Block Packed (SBP) and Distributed Square Block Packed (DSBP) storage formats. Much of the discussion on SBP have appeared in earlier publications but is summarized here for completeness.

The motivation for looking at other packed storage formats than the standard stacked column format used in BLAS and LAPACK is that there is no efficient way to use level-3 BLAS in combination with the latter format. High performance implementations of the BLAS at IBM and elsewhere have for a long time internally transformed the input into architecture-aware formats, for example the Square Block (SB) format. A block in SB format is stored contiguously and therefore it maps optimally into all levels of the memory hierarchy. The SBP format for packed storage is a convention on how to store a symmetric or triangular matrix as a set of contiguous square blocks.

Researchers at IBM have demonstrated that implementing Cholesky factorization on SBP input on a sequential architecture may not only be faster than standard packed Cholesky but also faster than standard full storage Cholesky. The difference between Cholesky on full storage and on SBP is claimed to be due to a better utilization of the memory hierarchy brought about by the contiguous block storage in SBP. Another benefit of SBP is that the Cholesky code can directly call BLAS kernels that do less data copying and internal transformations since the data is already in a suitable format. Further improvements can be made by storing the blocks themselves in some architecture-aware format, e.g., to better match memory streams or contiguous SIMD register loads. A block stored in a non-canonical format is referred to as a *non-simple block* [Gustavson et al. 2007a].

DSBP is a DM generalization of SBP that maps the blocks to processors by a 2D block-cyclic distribution. Implementations using the DSBP format may also reduce memory traffic when sending and receiving messages by avoiding message packing.

3.1 Description of the DSBP Format

An $N_b \times N_b$ block matrix (with square blocks of order n_b) is distributed according to a two-dimensional BCL with the first block stored on processor $(0, 0)$. The last local block column index on processor column q is

$$j_{\text{last}}(q) = \left\lfloor \frac{(N_b - 1) - q}{P_c} \right\rfloor .$$

Similarly for the last local block row index on processor row p:

$$i_{\text{last}}(p) = \left\lfloor \frac{(N_b - 1) - p}{P_r} \right\rfloor .$$

On each processor we construct an integer array cp (short for Column Pointer) with one component per local block column to speed up address calculation to a constant-time operation. The jth component of cp is defined by

$$\text{cp}[j] = \sum_{k=0}^{j} \left(i_{\text{last}}(p) - \left\lceil \frac{(q + kP_c) - p}{P_r} \right\rceil + 1 \right) - 1.$$

The expression in the sum calculates the number of local blocks on local block column k.

To each block we associate a block offset to index the block vector which stores the blocks. The block offset of local block (i, j) is

$$\text{blockoffset}(i, j) = \text{cp}[j] - (i_{\text{last}}(p) - i).$$

Figure 1 shows a detailed example of a block matrix in DSBP format.

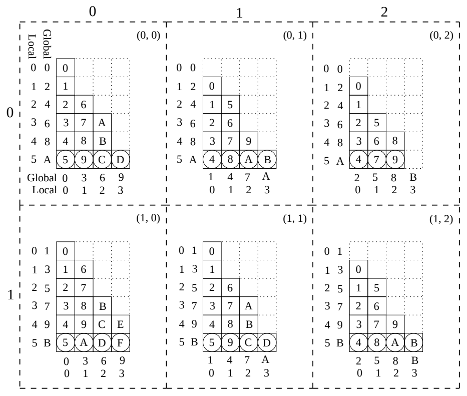


Fig. 1. Detailed example of a 12 × 12 block matrix distributed on a 2 × 3 mesh from the processors' viewpoint. Hexadecimal numbers indicate the local block offsets and circled block offsets correspond to the values in the column pointer array. Dotted blocks emphasize the typical full storage requirements.

Notice how the addressing scheme is based on the last block of a column and a negative offset depending on the local block row index. Thus, the addressing scheme is not dependent on whether the blocks are stored in a block packed or block full storage matrix. For full storage the column pointer array on all processors in Figure 1 contain $(5, 11, 17, 23)$ and the same addressing scheme and Cholesky algorithms can be used. In this case the column pointer array reduces to

$$\mathrm{cp}[j] = \sum_{k=0}^{j} (i_{\mathrm{last}}(p) + 1) - 1 = (j+1) i_{\mathrm{last}}(p) + j.$$

There is a strong connection between the local storage format of DSBP and the Block Compressed Column Storage (BCCS) used for sparse blocked matrices. The column pointer array plays a similar role as the column pointer array in BCCS (hence the same name). Because there is a regular pattern when the matrix is dense the row index array in BCCS does not need to be stored explicitly and the block offset is instead calculated directly from the column pointer array and the row index.

3.2 Properties of the DSBP Format

3.2.1 *Reduced Memory Requirements.* The storage required by the DSBP format is roughly half that of full storage. In case there are incomplete blocks, the last block row and column are padded. The storage requirement of the DSBP format in number of words is thus

$$\frac{N_b n_b^2 (N_b + 1)}{2} = \frac{\left\lceil \frac{N}{n_b} \right\rceil n_b^2 \left(\left\lceil \frac{N}{n_b} \right\rceil + 1 \right)}{2}.$$

3.2.2 *Less Data Movement in BLAS Operations.* Memory streams, vector registers, and other hardware features typically require register blocking. In order to effectively utilize such advanced hardware features, state-of-the-art kernels for BLAS routines such as `GEMM` usually reformat all or some of their operands [Goto and van de Geijn 2007; Gustavson et al. 2007a]. In [Gustavson et al. 2007a] it is shown that the amount of data copying performed in dense matrix factorization is $\mathcal{O}\left(N^3\right)$ but could potentially be reduced to $\mathcal{O}\left(N^2\right)$ by using SBP with non-simple storage formats for the blocks together with special kernel routines that operate on the non-simple blocks.

The four kernels in our Cholesky variants (`POTRF`, `TRSM`, `GEMM`, and `SYRK`) take as input one (`POTRF`), two (`TRSM`, `SYRK`) or three (`GEMM`) blocks, all of which are contiguous on account of the DSBP format. All operands will thus map into all levels of the memory hierarchy without conflict misses (assuming the cache capacity is sufficient to hold the operands and that the caches are at least three-way set associative). Combined with non-simple formats and special kernels this would provide an ideal situation for optimal kernel performance.

3.2.3 *Less Data Movement in Communication.* The message passing library (in this case an implementation of MPI) must pack and unpack messages when sending and receiving. This is more expensive for non-contiguous messages. An $m \times n$ submatrix in column major format generally consists of n contiguous vectors of length m, each separated by LDA $\geq m$ elements. On the other hand, our Cholesky variants send and receive contiguous square blocks.

4. DSBP ALGORITHM VARIANTS OF THE CHOLESKY FACTORIZATION

There are various ways to implement Cholesky factorization, such as left/right-looking and a symmetric version of Crout's method. We have chosen the variant commonly called blocked right-looking Cholesky factorization [Dongarra et al. 1998]. Algorithm 1 describes this variant on a high level.

Algorithm 1 High-Level Right-Looking Cholesky

```
1: while $N$, order of $A \neq 0$ do
2:     Choose block size $b = \min(n_b, N)$.
3:     Partition $A = \begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix}$ where $A_{11}$ has size $b \times b$.
4:     Compute the Cholesky factorization $A_{11} = L_{11}L_{11}^T$ in-place.
5:     Scale $A_{21} \leftarrow A_{21}L_{11}^{-T}$.
6:     Update the trailing matrix $A_{22} \leftarrow A_{22} - A_{21}A_{21}^T$.
7:     Continue with $A = A_{22}$.
8: end while
```

4.1 Buffers

We begin by describing the use of buffers in our parallel implementations. Because some data is remote we use explicit communication into local buffers. In the algorithms that we present there are three types of buffers:

—R (short for Reciprocal) contains a factored diagonal block used as input for scaling (TRSM) operations. Each R is generated by the pivot process and communicated to a separate replicated R residing on processor column.

—W (short for West) stores the scaled blocks of a panel used as input for updates (SYRK and GEMM). The West buffer is distributed along mesh columns and replicated along mesh rows.

—S (short for South) stores the block transpose of W (i.e., it is a row block vector and each block remains untransposed) used as input for GEMM updates. The South buffer is distributed along mesh rows and communicated along mesh columns to each processor.

In Basic, we only need one set of local buffers so $W(j)$ and $S(j)$ refer to these local West and South buffers, respectively. In Static and Dynamic, where two iterations are active at the same time, we use two sets of local buffers to reduce data dependencies and enhance performance. In these algorithms, $W(j)$ and $S(j)$ are instead to be read as the local West buffer $j \bmod 2$ and the local South buffer $j \bmod 2$, respectively. Subscripts i and k in $W_i(j-1)$ and $S_k(j-1)$, see Algorithm 3, refer to blocks i and k, respectively.

4.2 Basic Variant

The high-level blocked right-looking Cholesky factorization is adapted to DSBP in Algorithm 2. The distribution of computation follows the owner-computes rule, which states that the owner of an affected block is also the process that performs the computation. The details of the communication are left until Section 4.5 since

Algorithm 2 Basic

```
 1: for $j = 0, N_b - 1$ do
 2:    % Panel factorization
 3:    Cholesky($A_{jj}$), (POTRF)
 4:    $R \leftarrow A_{jj}$
 5:    *Start to replicate* $R$ (Algorithm 5)
 6:    *Wait for* $R$
 7:    for $i = j + 1, N_b - 1$ do
 8:       $A_{ij} \leftarrow A_{ij} R^{-T}$, (TRSM)
 9:       $W_i(j) \leftarrow A_{ij}$
10:       *Start to replicate* $W_i(j)$ *and* $S_i(j)$ (Algorithm 4)
11:    end for
12:    % Trailing matrix update
13:    *Wait for all* $W_*(j)$ *and* $S_*(j)$
14:    for $k = j + 1, N_b - 1$ do
15:       $A_{kk} \leftarrow A_{kk} - W_k(j) W_k(j)^T$, (SYRK)
16:       for $i = k + 1, N_b - 1$ do
17:          $A_{ik} \leftarrow A_{ik} - W_i(j) S_k(j)^T$, (GEMM)
18:       end for
19:    end for
20: end for
```

they are common to all three variants. Also, communication is intermixed with the computation in both Static and Dynamic variants.

4.3 Static Variant

Basic (Algorithm 2) is divided into distinct communication and computation phases and is therefore unable to effectively overlap communication with computation. By algorithmic look-ahead we mean that a processor begins to factor a panel before all of its preceeding trailing matrix updates have completed on that processor. The number of extra iterations that can be active in this way on any processor is the *depth* of the look-ahead. With this definition, Basic has look-ahead depth zero (no look-ahead), Static and Dynamic both have look-ahead depth one.

Static (Algorithm 3) is our look-ahead depth one variant with aggressively early scheduling of panel factorization suboperations. This scheduling is derived by following the critical path of the Cholesky factorization algorithm.

Figure 2 is a pictorial representation of the non-start-up part of Algorithm 3 (lines 12–37). The left part illustrates the algorithmic look-ahead for fused panel update and factorization (lines 14–18) and fused panel update and scaling (lines 20–28). The right half explains the trailing matrix update (lines 30–36).

4.4 Dynamic Variant

A careful examination of Algorithm 3 using a Gantt-chart of its execution reveals two situations where a processor becomes idle although it still has work to do. We call these situations *spurious synchronizations* since they could have been avoided, at least in the short term, by scheduling another available operation. These are:

(1) *Updates become available in random order* (see Figure 3) due to the non-deterministic order in which messages arrive. The static schedule enforces a

Algorithm 3 Static

```
 1: % First panel factorization
 2: Cholesky($A_{00}$), (POTRF)
 3: $R \leftarrow A_{00}$
 4: *Start to replicate* $R$ (Algorithm 5)
 5: *Wait for* $R$
 6: for $i = 1, N_b - 1$ do
 7:     $A_{i0} \leftarrow A_{i0} R^{-T}$, (TRSM)
 8:     $W_i(0) \leftarrow A_{i0}$
 9:     *Start to replicate* $W_i(0)$ *and* $S_i(0)$ (Algorithm 4)
10: end for
11: % Loop over remaining panels
12: for $j = 1, N_b - 1$ do
13:     % Update and factor diagonal block
14:     *Wait for* $W_j(j-1)$
15:     $A_{jj} \leftarrow A_{jj} - W_j(j-1) W_j(j-1)^T$, (SYRK)
16:     Cholesky($A_{jj}$), (POTRF)
17:     $R \leftarrow A_{jj}$
18:     *Start to replicate* $R$ (Algorithm 5)
19:     % Update and scale panel
20:     *Wait for* $S_j(j-1)$
21:     *Wait for* $R$
22:     for $i = j + 1, N_b - 1$ do
23:         *Wait for* $W_i(j-1)$
24:         $A_{ij} \leftarrow A_{ij} - W_i(j-1) S_j(j-1)^T$, (GEMM)
25:         $A_{ij} \leftarrow A_{ij} R^{-T}$, (TRSM)
26:         $W_i(j) \leftarrow A_{ij}$
27:         *Start to replicate* $W_i(j)$ *and* $S_i(j)$ (Algorithm 4)
28:     end for
29:     % Update non-panel part of trailing matrix
30:     for $k = j + 1, N_b - 1$ do
31:         *Wait for* $S_k(j-1)$
32:         $A_{kk} \leftarrow A_{kk} - W_k(j-1) W_k(j-1)^T$, (SYRK)
33:         for $i = k + 1, N_b - 1$ do
34:             $A_{ik} \leftarrow A_{ik} - W_i(j-1) S_k(j-1)^T$, (GEMM)
35:         end for
36:     end for
37: end for
```

strict order on the independent kernel operations that form a trailing matrix update.

(2) *Scaling is scheduled early* (line 25) in order to maximize the overlap possibilities. However, the updates on lines 24 and 30–36 might be possible to partially perform prior to the scaling. The static schedule waits for R (line 21) before any of the updates are performed and thus enforces a strict order on the operations.

In both cases, the static schedule causes spurious synchronizations to occur and it should be clear that any static schedule would have similar problems.

The first type of synchronization is described in Figure 3. It depicts a 12×12 block matrix distributed on a 2×3 mesh with details for processor $(0, 1)$. To the

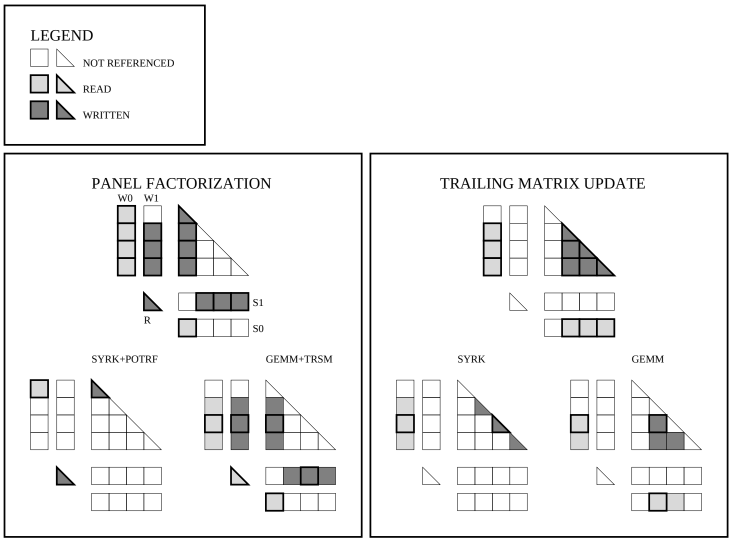


Fig. 2. The two tasks of fused panel factorization and trailing matrix update broken down into series of kernel operations with information on how the buffers are used.

left is a West buffer and below is a South buffer. Remote blocks and buffers have a dotted outline. For a block update to be available the corresponding West and South buffer blocks must both have received their data. The order in which buffer blocks become available is random and therefore it is impossible to optimally match by any static schedule.

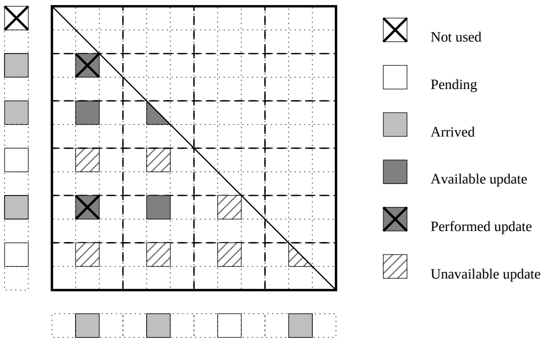


Fig. 3. Availability of updates is determined by the arrival of blocks in both West and South buffers.

An example of the second type of synchronization (obtained from an execution of Static) is illustrated in Figure 4. Note that the figure provides only partial timelines for two of the four processors and that time flows from left to right. The updates on processor $(0, 1)$ labeled A depend only on operations prior to the ones labeled

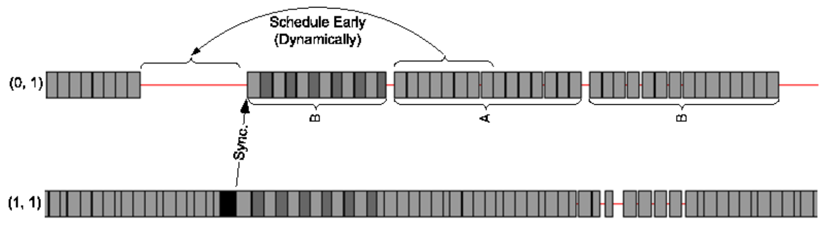


Fig. 4. A partial view of the timelines for the second processor column of a 2×2 mesh executing the Static variant on a 16×16 block matrix. Black boxes are diagonal block factorizations (`POTRF`), dark gray boxes are `TRSM` operations, and the light gray boxes are the `SYRK` and `GEMM` operations. The Static schedule introduces a gap on the $(0, 1)$ processor although any of the updates labeled A (rotated) could be scheduled there.

B and can therefore be executed before B. The second group of updates labeled B are the updates following the panel factorization partially performed in the first group labeled B. The gap before B could be filled by operations in A although the optimal number and selection of operations will depend on many factors, including the iteration number. It is impractical, if not impossible, to construct a static schedule which would fill such gaps.

A dynamic scheduling of tasks on the nodes would be more flexible and allow local avoidance of spurious synchronizations. In order to investigate if removing these spurious synchronizations reduces overall execution time, we designed and implemented a scheduling mechanism which addresses these issues. We give a brief description of the dynamic scheduling mechanism which we refer to as the Dynamic variant. At the start of each iteration a list is created with information about all the kernel operations (the tasks) that will execute in that iteration. The tasks are ordered in the list in exactly the same order as they would be executed by the Static variant. A pointer to the first unexecuted task is kept. The list is scanned sequentially for the first ready task, starting from the first unexecuted task. Figure 5 visualizes an example task list (acronyms used in the figure: **T**RSM, **S**YRK, and **G**EMM). One sees that the next task to execute would be the first ready `GEMM` task

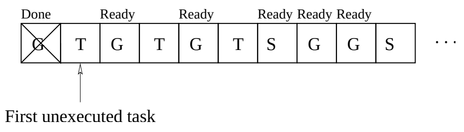


Fig. 5. Data structure for efficient dynamic scheduling.

since the data for the first `TRSM` task is unavailable.

4.5 Node Communication

Since all blocks are square they can be communicated as atomic units. In the algorithms described below, all messages have size n_b^2 and are n_b-by-n_b submatrices. Algorithm 4 shows how a single panel block (scaled block A_{ij}) is communicated to all its replicas in the remote West and South buffers. Communication starts at the

Algorithm 4 Communicate $W_i(j)$ and $S_i(j)$ from iteration j

```
 1: Let processor $(p_i, q_j)$ be the processor that holds block $A_{ij}$
 2: **if** I am on row $p_i$ and need $W_i(j)$ for some update **then**
 3:     **if** I am processor $(p_i, q_j)$ **then**
 4:         $W_i(j) \leftarrow A_{ij}$
 5:     **else**
 6:         receive($W_i(j)$, WEST)
 7:     **end if**
 8:     send($W_i(j)$, EAST) if needed for some update on EAST neighbour
 9: **end if**
10: Let processor $(p_i, q_i)$ be the processor that holds block $A_{ii}$
11: **if** I am on column $q_i$ and need $S_i(j)$ for some update **then**
12:     **if** I am processor $(p_i, q_i)$ **then**
13:         $S_i(j) \leftarrow W_i(j)$
14:     **else**
15:         receive($S_i(j)$, NORTH)
16:     **end if**
17:     send($S_i(j)$, SOUTH) if needed for some update on SOUTH neighbour
18: **end if**
```

root process (p_i, q_j) that owns the scaled panel block A_{ij}. The block is passed on from west to east until it reaches process (p_i, q_i) that owns the diagonal block A_{ii}. This process then splits the communication into a north to south transfer to fill the replicas of the South buffer. Both communications continue in parallel. At the end of this procedure, processors $(p_i, *)$ have their copy of $W_i(j)$ and processors $(*, q_i)$ have their copy of $S_i(j)$.

All communication is performed using non-blocking MPI routines. The algorithm is executed for all the blocks of W and S concurrently and asynchronously. MPI polling is used to discover transfer completion and to resume the corresponding algorithm instance.

Algorithm 5 Communicate R from iteration j

```
1: Let processor $(p_j, q_j)$ be the processor that holds block $A_{jj}$
2: **if** I am on column $q_j$ and need $R$ for some scaling operation **then**
3:     **if** I am processor $(p_j, q_j)$ **then**
4:         $R \leftarrow A_{jj}$
5:     **else**
6:         receive($R$, NORTH)
7:     **end if**
8:     send($R$, SOUTH) if needed for some scaling operation on SOUTH neighbour
9: **end if**
```

Communication of the R buffer is similar (Algorithm 5). The root process (p_j, q_j) that owns the factored diagonal block A_{jj} starts a north to south transfer to build the replicas of R on processors $(*, q_j)$.

4.6 Considerations for Hybrid Systems

If the nodes of the DM system are shared memory (e.g. SMP or multi-core), an additional level of scheduling is required if message passing within a node is to be avoided. We use the term *node-level scheduling* to refer to the scheduling of the tasks of an MPI process onto the processor cores of a DM node. One technique to solve the node-level scheduling problem is to use multi-threaded BLAS on each node and map only one MPI process to each node. This is one way the node-level scheduling problem can be addressed in LAPACK and ScaLAPACK.

For the variants presented here it is probably not efficient to parallelize the kernels, especially if n_b is small. Dynamic scheduling of atomic kernel operations is one way to expose parallelism to many threads. This strategy has been successfully tested recently, by both the PLASMA and FLAME projects [Buttari et al. 2007; Chan et al. 2007] in pure shared memory environments.

Even though several threads are computing there need not be more than one thread that calls MPI routines. In fact, for the Cell BE it might be advisable to let the PPE handle all MPI calls and delegate all computations to the SPEs. Therefore, it is not necessary to have a thread-safe implementation of MPI in order to use algorithms and methods discussed here.

5. COMPUTATION-COMMUNICATION OVERLAP EVALUATION

In what follows, we assume the reader is familiar with concepts such as blocking/non-blocking, send/receive requests, and other basic MPI terminology. For definitions see the MPI standard documents ([MPI Forum 1995]).

On many systems there is separate hardware for communication and computation that execute concurrently. The extent to which this parallelism can be exploited is highly dependent on the machine and system software. We therefore present an evaluation of the overlap capabilities of our target machine in this section.

With MPI, overlap is enabled by using *non-blocking primitives* for point-to-point communication (the MPI standard interface does not define any non-blocking collective operations). When the network controller detects an incoming message it must know where to store it. If a process has posted a receive the MPI library could instruct the network controller to place the message directly into the buffer supplied by the user. If the process has not yet posted a receive when an incoming message is detected then either the message transfer must be delayed or some temporary buffer must be allocated. These two options result in two different types of message transfer protocols:

(1) The *eager* protocol: allocate a temporary buffer into which the message is received, and copy from the temporary buffer to the receive buffer when the receive is posted. The eager protocol is used to improve latency of small messages at the cost of reduced bandwidth due to the extra memory copy operations.
(2) The *rendezvous* protocol: the sender and receiver handshake to make sure a receive buffer is available. Transfer of data directly into the receive buffer can thus be guaranteed and this protocol is used for large, bandwidth limited messages.

The eager protocol allows for hardware parallelism but costs at least one extra memory copy. The rendezvous protocol allows overlap on the sending side but not on the receiving side unless the receive is non-blocking too. We therefore conclude that the use of non-blocking send and receive is critical as only then is overlap practically possible on both sides of the communication.

An important feature of an MPI implementation is *independent progress* [Brightwell and Underwood 2004], which gives an MPI implementation the capability of performing communication while the user process is not executing an MPI routine. An MPI implementation that supports independent progress can actually overlap communication with computation and such an implementation is thus preferred.

5.1 Micro-Benchmarks

We designed two micro-benchmarks on our target machine to investigate which type of protocol is used in what situation and also to measure the amount of speedup that we can realistically achieve.

In both benchmarks we combine the execution of one or several GEMM updates with the transmission of a message. In all cases we used two processes on separate nodes and the computation of a GEMM update with the chosen parameters took approximately $T_{\text{compute}} = 544\mu s$. We denote the time to send a message of any particular size $T_{\text{communicate}}$.

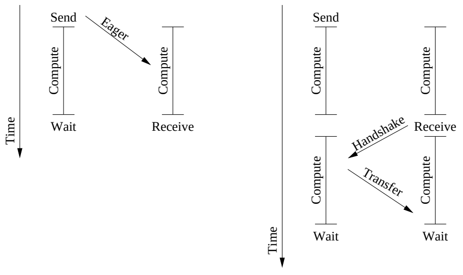


Fig. 6. Two micro-benchmarks to evaluate the MPI library overlap capabilities. Left: a test for the use of an eager protocol. Right: a test for independent progress.

5.1.1 *Benchmark A: Eager versus Rendezvous.* This benchmark is designed to identify which type of protocol is used depending on the message size and it is visually described in Figure 6 (left). Two processes, sender and receiver, each compute one GEMM update. The sender starts a non-blocking send prior to starting a GEMM computation and waits for it to complete after its GEMM computation completes. The receiver starts a blocking receive at the end of its GEMM computation. If an eager protocol is used, we would expect the time for both processes to be approximately

$$\max(T_{\text{compute}}, T_{\text{communicate}})$$

since the communication would be concurrent with the computation. This expected execution time is marked with a dashed curve in Figure 7. In the same figure, the solid curve marks the measured time. Clearly, up to about 32 KB the total time is just above the time to compute, whereas for larger messages it is closer to the

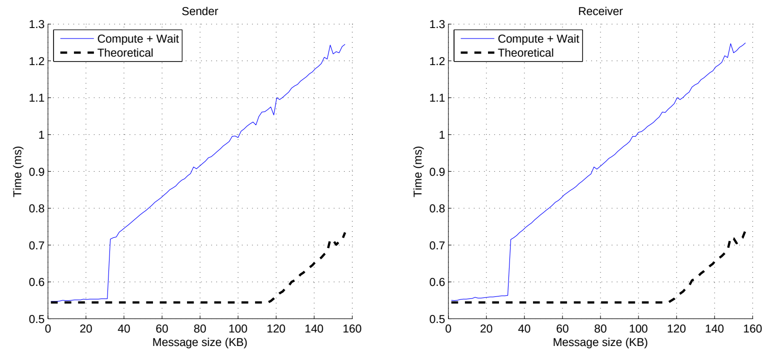


Fig. 7. Results for Benchmark A.

sum of the compute and communicate times. We can be sure, based only on these results, that for messages up to 32 KB the system uses an eager protocol. For larger messages a rendezvous protocol is probably used. However, the increased execution time could also be explained by a lack of independent progress.

5.1.2 *Benchmark B: Independent Progress and Speedup.* This benchmark is designed to discover if the MPI implementation supports independent progress and also allows us to assess the practically achievable speedup of overlapping communication with computation (see Figure 6 (right) for description). Both processes now compute a GEMM twice and the receiver starts the receive between the two computations and waits for it to complete after the second computation. If a rendezvous type protocol is used, then the time for both processes in the presence of independent progress is expected to be

$$T_{\text{compute}} + \max(T_{\text{compute}}, T_{\text{communicate}}).$$

This is marked with a dashed curve in Figure 8. As with the previous benchmark,

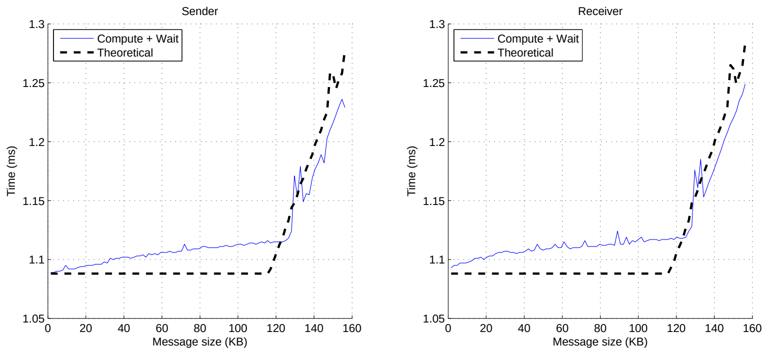


Fig. 8. Results for Benchmark B.

the measured times are marked with a solid curve. There are no major deviations from the expected times and the difference on the expected flat part indicates that overlap slightly affects computation time. This indicates that $90 - 95\%$ of the communication overhead is eliminated on both sides of the communication.

6. PERFORMANCE RESULTS

The performance and scalability of the three algorithm variants were evaluated on the Sarek cluster at the High-Performance Computing Center North (HPC2N) in Umeå, Sweden. The Sarek cluster has 192 nodes with dual AMD Opteron 248 (2.2 GHz) processors. The nodes are connected with a Myrinet 2000 high speed interconnect with the MPI library MPICH-MX version 1.2.7 capable of point-to-point communication with roughly 230 MB/s bandwidth. Each node has 8 GB of memory and the BLAS library we used was GotoBLAS version r1.12. All the tests were performed with a distribution and algorithmic block size of $n_b = 100$. The Dynamic variant executed nearly identically as fast as the Static variant. For this reason, we omit reporting on the performance and scalability of the Dynamic variant.

For large problems the time spent in `GEMM` operations completely dominate the time spent in all other operations as well as the idle time and communication overhead. Therefore, if two algorithms have different `GEMM` performance the one with the highest performance will eventually outperform the other. Hence, we focus our attention on small and medium sized problems to highlight the differences amongst our DSBP algorithms and their relation to `PDPOTRF` in ScaLAPACK. However, large scale problems are also tested to make sure performance does not degrade.

Below, the performance of each kernel is examined. In addition, the communication system's characteristics are determined and discussed. The section ends by comparing our measured performance to our models of execution showing that the Static variant has a near-optimal scheduling of tasks on the nodes for medium and larger sized problems.

6.1 Kernel Performance

Since the operands to each of the four kernels are stored as contiguous blocks of a known size, the execution time of these kernel routines can be accurately approximated by a simple benchmark. There is little variability in the performance of a kernel routine except for where in the memory hierarchy the operands are at the time of the invocation.

To get a fair estimate of the performance of each kernel under a realistic scenario (e.g., the operands are not optimally placed in the memory hierarchy), the time spent in each kernel during the execution of Basic was measured and averaged over the total number of their invocations. The results of these tests are reported in Table I. Note the relatively poor performance of `POTRF`. The reason for this is that LAPACK uses a level 2 kernel factorization routine `POTF2`. Furthermore, the early flattening of the performance is indicative of a mismatch between the blocksize and/or algorithm with the BLAS implementation.

We have not optimized the kernels to take advantage of non-simple block storage formats. In all measurements we used LAPACK and BLAS routines. It is therefore important for us to investigate the performance of each of the these four employed

Kernel	**flops**	**Time (μs)**	**Gflops/s**
`POTRF`	$n_b^3/3$	192	1.74
`TRSM`	n_b^3	350	2.86
`SYRK`	$n_b^3 + n_b^2$	318	3.18
`GEMM`	$2n_b^3$	565	3.54

Table I. Kernel performance figures ($n_b = 100$).

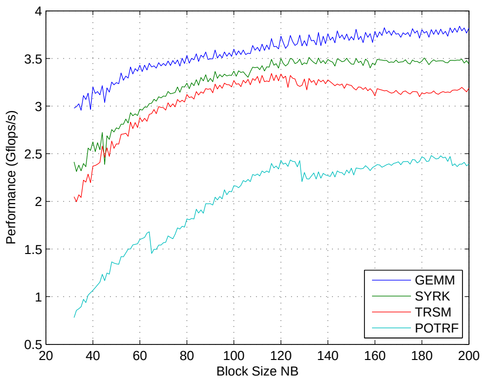


Fig. 9. Performance of kernels with respect to block size.

routines (see Figure 9). It also justifies the choice of $n_b = 100$ for the other tests, because at this block size the performance of the `GEMM` routine is almost flat.

6.2 Communication Performance

An important characteristic of a DM machine is its message passing performance. In our case, it is the MPI implementation MPICH-MX that we benchmark. We use the communication model

$$t_s + t_w m$$

for an m-word message and estimate the parameters t_s and t_w by experimentation. The startup cost (t_s) and inverse bandwidth (t_w) were determined by fitting this model to a ping-pong benchmark (see Table II for results). Note that the latency

t_s	29.6 μs
t_w	34.6 ns

Table II. The communication parameters for the Sarek cluster.

is almost 1000 times higher than the inverse bandwidth. Even so, for the chosen block size of $n_b = 100$ the block transmission time will be approximately 10 times the latency.

6.3 Absolute Performance

We tested Basic and Static as well as the ScaLAPACK full storage `PDPOTRF` routine on various mesh and problem sizes. The largest test was on a 12×12 mesh and a matrix of order 110000. In Figure 10, we report the performance per processor on the 12×12 mesh.

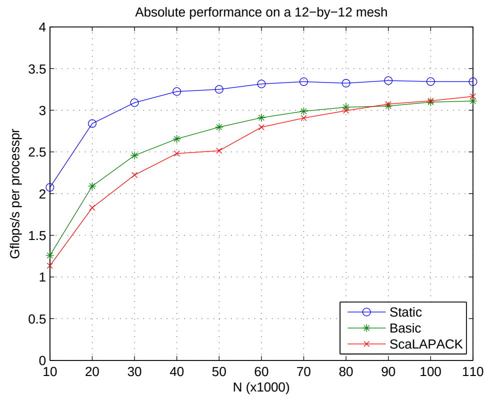


Fig. 10. Absolute performance per processor of Basic, Static, and `PDPOTRF` on 12×12 mesh for various problem sizes.

6.4 Modeled Versus Measured Performance

In this section, we model the execution of Basic and Static to support our claim of the near-optimality of the Static schedule.

6.4.1 *Models of Basic and Static.* Inter-node data dependencies are handled via message passing and intra-node data dependencies by the ordering of the operations. The execution time is determined by the speed of the kernels, the communication overhead, and the inter-node dependencies. Since the kernels are assumed to be optimized, a parallel algorithm should minimize the impact of the communication and other overheads. Therefore, we model the performance of Basic and Static by simulations to see if our algorithms meet this expectation. We do not model communication overhead or other overheads besides computation since we wish to compare our performance with an ideal machine where overhead is not an issue. This is one way to quantify the otherwise so elusive overhead component in parallel algorithms.

6.4.2 *Comparisons.* In Figure 11, a comparison on 36 processors (6×6 mesh) is presented. The measured performance for Static is very close to the simulated performance for medium to large problems. However, Basic is not close to its simulated performance showing that communication overhead is a significantly limiting factor for this less efficient algorithm.

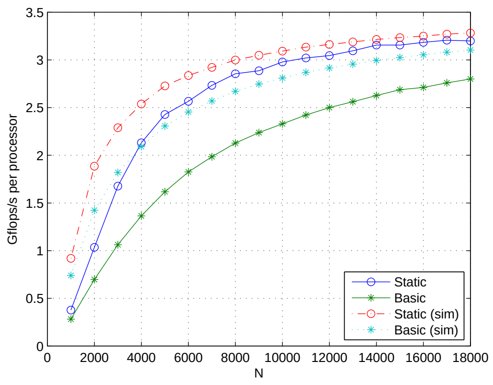


Fig. 11. Comparisons between the simulated performance and the measured performance for Basic and Static on 36 processors arranged in a 6×6 mesh.

Let T denote the parallel execution time (assumed equal for all $P_r \cdot P_c$ processors due to synchronization). The execution time can be partitioned into four components for each processor k $(0 \leq k < P_r \cdot P_c)$:

$$T = T^k_{\text{computation}} + T^k_{\text{communication}} + T^k_{\text{idle}} + T^k_{\text{overhead}}.$$

Typically, the components of T differ for separate processors (hence the superscript k). All three variants (Basic, Static, and Dynamic) use the same kernels and the same distribution of computational work. They are instances of a bigger class of algorithm variants with the same kernels and distribution of work. If $T^k_{\text{communication}} = T^k_{\text{overhead}} = 0$, a lower bound for the execution time is

$$\max_k T^k_{\text{computation}} = T - \min_k T^k_{\text{idle}}.$$

This quantity can be accurately estimated by the Static model since these conditions hold by design.

In Table III, we illustrate the minimum simulated idle times on a 4×4 mesh. Similar negligible idle times for the Static model have been observed for other meshes as well. These negligible simulated idle times show that the Static schedule

	Basic		**Static**	
N	T	$\min_k T^k_{\text{idle}}$	T	$\min_k T^k_{\text{idle}}$
5000	1.0	0.1135	0.9	0.0012
10000	6.8	0.4409	6.4	0.0015
15000	21.9	0.9822	20.9	0.0012
20000	50.7	1.7380	48.9	0.0015
25000	97.5	2.7110	94.8	0.0012
30000	166.9	3.9000	163.0	0.0015

Table III. Simulated execution time and minimum simulated idle time (both in seconds) for the Basic and Static variants.

is near-optimal in theory. We now discuss our empirical evidence.

6.4.3 *Empirical Evidence of Negligible Idle Times.* Simulations of execution strongly indicated that with no communication or other overhead the minimum idle time observed over all processors is close to zero. The comparisons presented in Section 6.4.2 predict that this should be observable in practice. The code was instrumented to accumulate the time spent in synchronizing MPI calls. Table IV shows that at least one processor has a small idle time component, i.e., it is active almost all the time. A further reduction in parallel execution time would either require faster kernel execution, moving work between nodes, or reducing the number of nodes and re-balancing the data layout. Such efficiencies are beyond the scope of this paper.

$P_r \times P_c$	4×4		8×8		12×12	
N	T	$\min T_{\text{wait}}$	T	$\min T_{\text{wait}}$	T	$\min T_{\text{wait}}$
10000	6.554	0.076	1.956	0.116	1.116	0.195
20000	49.719	0.289	13.354	0.092	6.520	0.155
30000	164.880	0.343	43.053	0.107	20.221	0.161
40000	392.612	0.923	100.500	0.109	45.948	0.160
50000	763.795	0.276	194.252	0.311	89.033	0.172
60000	1303.342	0.142	334.360	0.130	150.797	0.211
70000			521.126	0.138	237.592	0.266
80000			783.428	0.147	356.522	1.080
90000			1123.918	0.119	503.045	1.117
100000					692.181	0.400
110000					921.571	0.199

Table IV. Measured time in synchronizing code (e.g., MPI wait routines). The columns labeled $\min T_{\text{wait}}$ contain the smallest measured waiting time over all the processors.

The results in Table IV demonstrate why the Dynamic variant did not outperform the Static variant. Reducing idle time is the primary benefit of a dynamic schedule, but since the Static variant is near optimal in this regard only small improvements, if any, are possible.

7. SCALABILITY

Fixed size scaling (strong scalability) examines how performance degrades when more processors are used to solve a problem of fixed size. The performance per processor should ideally remain constant but in practice it will decrease as a consequence of increased overhead. Based on Figure 12 we conclude that Static is more scalable than Basic. The advantage of overlapping iterations is more apparent when a large number of processors solve a small problem.

Under memory constrained scaling the amount of available memory is assumed to scale linearly with the number of processors and the problem size is scaled so that the consumed memory is kept constant per processor. An algorithm has a good memory constrained scalability if it can maintain a constant performance per processor. For Cholesky factorization, which operates on $\mathcal{O}\left(N^2\right)$ memory, the problem size N must be scaled with a factor $\sqrt{p_2/p_1}$ when the number of processors scales from p_1 to p_2 (see Figure 13). The memory constrained scalability of all tested algorithms was excellent.

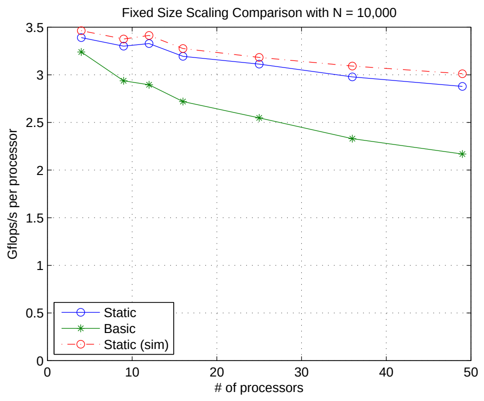


Fig. 12. Performance on various mesh sizes for a fixed problem size.

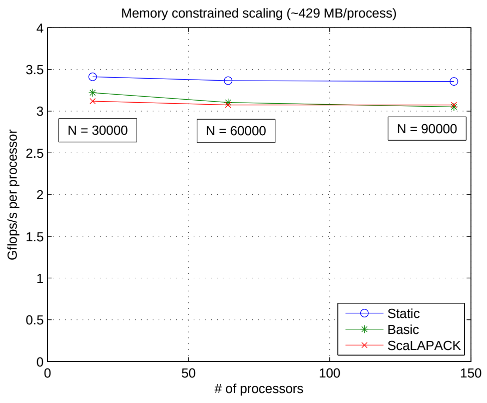


Fig. 13. Memory constrained scaling (with full storage requiring ≈ 429 MB per process).

The isoefficiency function [Grama et al. 1993] maps the number of processors to problem size. It tells how much the problem size must be scaled up to maintain a constant efficiency. In Figure 14, we show curves which are similar to the iso-efficiency curves but instead of comparing parallel execution time with the best serial implementation we measure efficiency as performance per processor. With this metric, we conclude that Static is far more scalable than Basic.

8. CONCLUSIONS AND FUTURE WORK

We presented the Distributed SBP format and showed that it is possible to achieve the same or better performance for packed storage Cholesky factorization compared to full storage. The Static variant has a $5-55\%$ higher performance than the Basic variant for matrices of size N between 5000 and 10000 on 4–49 processors. In addition, the Static variant is significantly more scalable than the Basic algorithm

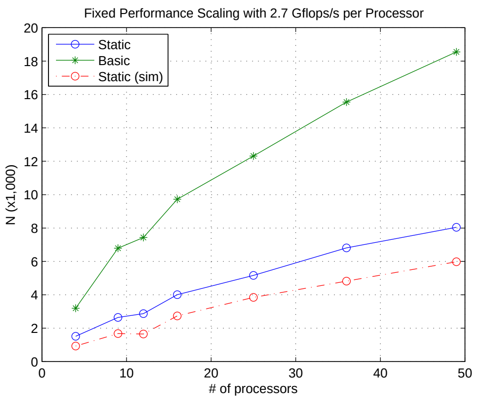


Fig. 14. Problem size required to achieve a fixed performance per processor of 2.7 Gflops/s.

for fixed problem sizes and is also much better to maintain a constant FPU efficiency as the number of processors increase.

Models of execution that assume no parallel overhead except processor idling support that the Static variant is close to the optimum schedule on our target DM machine. Based on these models we also conjecture that the Static variant is capable of nearly completely overlapping the communication with computation. Important characteristics of our target machine include its ability to reach near peak performance on small Level 3 BLAS operations and a low overhead MPI implementation capable of asynchronous communication with independent progress. The presented algorithms should perform well on other machines with similar characteristics.

Since the Static and Dynamic variants give similar performance, the simpler Static variant is sufficient. A dynamic scheduling similar to the Dynamic variant could provide efficient scheduling on hybrid systems with SMP or multi-core nodes.

There is frequent polling of the MPI layer in order for the communication algorithm to detect the completion of requests. This overhead would be avoided if the MPI interface and various MPI implementations supported callbacks when a request completes.

The scheduling of tasks in the Basic variant is described by an outer control loop and inner loops to traverse the blocks of the matrix. The Static variant is more complicated with a necessary preamble (see lines 1–10 of Algorithm 3) before the main control loop. The Dynamic variant is yet more complicated, with a dynamic rearrangement of the control loop body of the Static variant. An optimal schedule is likely to require a rearrangement across several iterations of the control loop. It is difficult to imagine there is any practical way of coding such an optimal schedule without using dynamic scheduling.

REFERENCES

Agarwal, R. C. and Gustavson, F. G. 1988. A parallel implementation of matrix multiplication and LU factorization on the IBM 3090. In *Aspects of Computation on Asynchronous and Parallel Processors*, M. Wright, Ed. IFIP, North-Holland, Amsterdam, 217–221.

Agarwal, R. C. and Gustavson, F. G. 1989. Vector and Parallel Algorithms for Cholesky

Factorization on IBM 3090. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing.* ACM Press, New York, NY, USA, 225–233.

Agarwal, R. C., Gustavson, F. G., and Zubair, M. 1994. A High Performance Matrix Multiplication Algorithm on a Distributed Memory Parallel Machine Using Overlapped Communication. *IBM Journal of Research and Development 38,* 6 (November), 673–681.

Baboulin, M., Giraud, L., and Gratton, S. 2005a. A Parallel Distributed Solver for Large Dense Symmetric Systems: Applications to Geodesy and Electromagnetism Problems. *Internation Journal of High Performance Computing Applications 19*, 353–363.

Baboulin, M., Giraud, L., Gratton, S., and Langou, J. 2005b. A distributed packed storage for large parallel calculations. Tech. Rep. TR/PA/05/30, CERFACS, Toulouse, France.

Brent, R. P. and Luk, F. T. 1982. Computing the Cholesky Factorization Using a Systolic Architecture. Tech. Rep. TR 82–H521, Department of Computer Science, Cornell University, Upson Hall, Cornell University, Ithaca, New York 14853. September.

Brightwell, R. and Underwood, K. D. 2004. An analysis of the impact of MPI overlap and independent progress. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing.* ACM Press, New York, NY, USA, 298–305.

Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. 2007. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. Tech. Rep. UT-CS-07-600.

Chan, E., Quintana-Ortí, E. S., Quintana-Ortí, G., and van de Geijn, R. 2007. SuperMatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures.* San Diego, CA, USA, 116–125.

Choi, J., Dongarra, J. J., Ostrouchov, S., Petitet, A. P., Walker, D. W., and Whaley, R. C. 1996. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming 5,* 3 (Fall), 173–184.

Dackland, K., Elmroth, E., Kågström, B., and Van Loan, C. 1992. Parallel Block Factorizations on the Shared Memory Multiprocessor IBM 3090 VF/600J. *Int. J. Supercomputer Applications 6.1*, 69–97.

Dackland, K., Elmroth, E., and Kågström, B. 1993. A ring–oriented approach for block matrix factorizations on shared and distributed memory architectures. In *SIAM Conference on Parallel Processing for Scientific Computing*, R. S. et al, Ed. SIAM Publications, 330–338.

D'Azevedo, E. and Dongarra, J. 1998. Packed storage extension for ScaLAPACK. Tech. Rep. UT-CS-98-385.

Dongarra, J. J., Duff, I. S., Sorenson, D. C., and van der Vorst, H. A. 1998. *Numerical Linear Algebra on High-Performance Computers.* SIAM.

Geist, G. A. and Heath, M. T. 1985. Parallel Cholesky factorization on a hypercube multiprocessor. Tech. Rep. ORNL–6190, Oak Ridge National Lab., TN (USA). August.

Gerasoulis, A. and Nelken, I. 1989. Scheduling Linear Algebra Parallel Algorithms on MIMD Architectures. In *Parallel Processing for Scientific Computing.* 68–95.

Golub, G. H. and van Loan, C. F. 1996. *Matrix Computations*, third ed. Johns Hopkins University Press.

Goto, K. and van de Geijn, R. A. 2007. Anatomy of High-Performance Matrix Multiplication. Accepted for publication in ACM Transactions on Mathematical Software.

Grama, A. Y., Gupta, A., and Kumar, V. 1993. Isoefficiency: measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology: Systems and Applications 1,* 3, 12–21.

Gustavson, F. G., Gunnels, J. A., and Sexton, J. C. 2007a. Minimal Data Copy for Dense Linear Algebra Factorization. In *PARA 2006: State of the Art in Scientific and Parallel Computing*, B. Kågström et al., Eds. Lecture Notes in Computer Science, LNCS 4699. Springer, 540–549.

Gustavson, F. G., Karlsson, L., and Kågström, B. 2007b. Three Algorithms for Cholesky Factorization on Distributed Memory Using Packed Storage. In *PARA 2006: State of the Art in Scientific and Parallel Computing*, B. Kågström et al., Eds. Lecture Notes in Computer Science, LNCS 4699. Springer, 550–559. Also as IBM Technical Report RC24137.

MPI Forum 1995. MPI: A Message Passing Interface Standard. `http://www.mpi-forum.org/`.

O'Leary, D. P. and Stewart, G. W. 1985. Data-flow algorithms for parallel matrix computation. *Communications of the ACM 28*, 840–853.

Strazdins, P. 1998. A Comparison of Lookahead and Algorithmic Blocking Techniques for Parallel Matrix Factorization. Tech. Rep. TR-CS-98-07, Canberra 0200 ACT, Australia.

van de Geijn, R. A. 1997. *Using PLAPACK*. MIT Press.

III

Paper III

Blocked In-Place Transposition with Application to Storage Format Conversion

Lars Karlsson[1]

[1] *Department of Computing Science and HPC2N, Umeå University*
SE-901 87 Umeå, Sweden
larsk@cs.umu.se

Abstract: We develop a prototype library for in-place (dense) matrix storage format conversion between the canonical row and column-major formats and the four canonical block data layouts. Many of the fastest linear algebra routines operate on matrices in a block data layout. In-place storage format conversion enables support for input/output of large matrices in the canonical row and column-major formats. The library uses algorithms associated with in-place transposition as building blocks. We investigate previous work on the subject of (in-place) transposition and the most promising algorithms are implemented and evaluated. Our results indicate that the Three-Stage Algorithm which only requires a small constant amount of additional memory performs well and is easy to tune. Murray Dow's V5 algorithm, which is a two-stage semi-in-place algorithm that requires a small amount of additional memory is sometimes a better choice. The write-allocate strategy of most cache-based computer architectures appears to be the cause of an observed performance problem for large matrices.

Key words: In-place transposition, storage format conversion, blocked storage formats.

Blocked In-Place Transposition with Application to Storage Format Conversion[*]

Lars Karlsson[†]
larsk@cs.umu.se

UMINF 09.01

Department of Computing Science
Umeå University and HPC2N
S-901 87 Umeå, Sweden

January 26, 2009

Abstract

We develop a prototype library for in-place (dense) matrix storage format conversion between the canonical row and column-major formats and the four canonical block data layouts. Many of the fastest linear algebra routines operate on matrices in a block data layout. In-place storage format conversion enables support for input/output of large matrices in the canonical row and column-major formats. The library uses algorithms associated with in-place transposition as building blocks. We investigate previous work on the subject of (in-place) transposition and the most promising algorithms are implemented and evaluated. Our results indicate that the Three-Stage Algorithm which only requires a small constant amount of additional memory performs well and is easy to tune. Murray Dow's V5 algorithm, which is a two-stage semi-in-place algorithm that requires a small amount of additional memory is sometimes a better choice. The write-allocate strategy of most cache-based computer architectures appears to be the cause of an observed performance problem for large matrices.

[*]This research was conducted using the resources of High Performance Computing Center North (HPC2N).

[†]Funded in part by the VR grant 70625701. The work is also a part within an IBM Shared University Research (SUR) grant.

Contents

1 Introduction

We develop a library for in-place matrix storage format conversion based on in-place transposition algorithms. In-place transposition is a well-studied problem [1, 15, 5, 4, 3, 2, 6, 14, 11]. Nonetheless, the growing gap between CPU processing speed and memory bandwidth/latency unfortunately means that most of the early algorithms for pure in-place transposition take one or more orders of magnitude longer to execute than an out-of-place algorithm. The main reason is that these in-place algorithms move individual elements that are not contiguous in memory, thereby severely stressing the memory hierarchy. Several algorithms address these issues [1, 5, 6, 14] but some are only semi-in-place and require a relatively small but still non-constant amount of additional memory.

In-place transposition has applications in FFT algorithms [9, 8] and to convert between the Column-Major (CM) and Row-Major (RM) canonical matrix storage formats. However, our main motivation is to provide a software package for fast in-place conversion between the canonical CM and RM formats and various block data layouts (see Sections 2 and 6). Such block formats are often used instead of CM/RM in linear algebra kernels to improve data locality. Conversion is required in order to support the familiar storage formats at the interface level while internally working with block data layouts.

The paper is structured as follows. In Section 2, we recall the canonical storage formats (CM/RM) and four canonical block storage formats. After discussing storage formats, we continue by recalling some of the previously published techniques for in-place transposition in Section 3. We focus on algorithms that require only a few sweeps through the matrix since memory bandwidth is the limiting factor. The connection to matrix-vector multiplication and Kronecker products is reviewed in Section 4, and in Section 5, we illustrate how some types of permutations can be implemented using in-place transposition algorithms. Implementations of these permutations are used as building blocks for our storage format conversion library. Conversion between storage formats is further discussed in Section 6 followed by a detailed description of the so-called Three-Stage Algorithm for in-place transposition in Section 7. The Three-Stage Algorithm has been previously mentioned in the literature [10, 14], but to our knowledge it has not been carefully compared with other algorithms. We introduce the conversion library in Section 8, followed by computational experiments in Section 9 and conclusions in Section 10. Finally, in Appendix A we give details on how to re-formulate some previously published algorithms using the notation developed in this paper.

We consistently use a *zero-origin indexing* convention, meaning that an $m \times n$ matrix has m rows numbered $0, \ldots, m-1$ and n columns numbered $0, \ldots, n-1$. The top left element of the matrix is consequently $A(0,0)$ and the first storage location is 0. We interchangeably use *(storage) format* and *data layout* to denote the scheme by which a matrix is stored in memory.

2 Matrix Storage Formats

2.1 Canonical Formats

The two canonical storage formats typically used by compilers are the Row-Major (RM) and Column-Major (CM) data layouts. Using either of these formats, element $A(i, j)$ of the $m \times n$ matrix A is stored at location

$$\begin{array}{rl} i + jm & \text{(CM), or} \\ in + j & \text{(RM).} \end{array}$$

Figure 1 is an example of a 9×6 matrix in CM format. The elements are

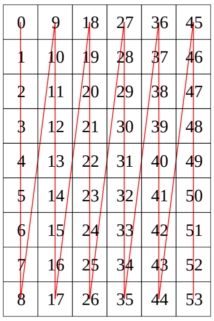


Figure 1: A 9×6 matrix in CM format.

numbered according to their location in memory. We use a polyline to highlight the storage order of the elements. In all examples, the start of the sequence is in the top left corner and the end is in the bottom right corner. In this particular example, the sequence begins with $0, 1, \ldots$ and ends with $\ldots, 52, 53$.

There is a strong connection between the CM/RM formats and matrix transposition. For example, if A is stored in RM format it is indistinguishable from A^T stored in CM format. Hence, transposing A in CM format is the same as converting it into RM format and vice versa.

2.2 Block Formats

It has long been understood that the CM and RM formats are suboptimal for a large class of algorithms, including most of linear algebra, FFT, image analysis, and more. The reason is that spatial data locality is only maintained within columns (CM) or rows (RM), whereas many algorithms need locality in both dimensions. For example, element $A(i, j)$ and $A(i + 1, j)$ are close in CM (stride 1) but $A(i, j)$ and $A(i, j + 1)$ are far apart (stride m). *Block formats* bring elements within certain submatrices (known as blocks) closer together. Accessing a submatrix in a block format typically provides more spatial data locality than accessing the same submatrix in CM or RM format [17].

Among the many proposed hybrid data layouts, the canonical block formats have most of the benefits of hybrid data layouts (e.g., see [7, 17]) while keeping a simple mapping from element to storage location (the so called *storage mapping*). Assume that an $m \times n$ matrix is partitioned into an $M \times N$ block matrix with blocks of size $m_b \times n_b$ and that each block is stored contiguously in memory. Typically, $m = Mm_b$ and $n = Nn_b$. We call such a data layout a *block format* and by choosing CM or RM as the storage format for the blocks and CM or RM as the storage format for the elements inside each block we get the four *canonical* block formats: CCRB, CRRB, RCRB, and RRRB. The suffix RB is an acronym for Rectangular Block, the first letter indicates the storage format used for the blocks and the second letter indicates the storage format for elements inside a block. For example, the RCRB format stores the blocks in RM format while the elements inside each block are stored in CM format.

The element $A_{i_1,j_1}(i_2, j_2)$ of the block matrix

$$A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,N-1} \\ \vdots & \ddots & \vdots \\ A_{M-1,0} & \cdots & A_{M-1,N-1} \end{bmatrix}$$

denotes the (i_2, j_2)-element of the (i_1, j_1)-block. It is the same as $A(i_1 m_b + i_2, j_1 n_b + j_2)$ and is stored at one of the following locations:

$$\begin{array}{rl} (j_1 M + i_1) n_b m_b + (j_2 m_b + i_2) & \text{(CCRB)} \\ (j_1 M + i_1) m_b n_b + (i_2 n_b + j_2) & \text{(CRRB)} \\ (i_1 N + j_1) n_b m_b + (j_2 m_b + i_2) & \text{(RCRB)} \\ (i_1 N + j_1) m_b n_b + (i_2 n_b + j_2) & \text{(RRRB)} \end{array}$$

Figure 2 illustrates a 9×6 matrix (the same matrix as in Figure 1) stored in CCRB format with blocks of size 3×2.

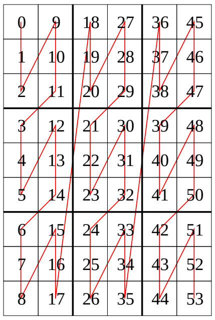


Figure 2: A 9×6 matrix in CCRB format.

3 Basics of In-Place Transposition

3.1 Algorithms for Square Matrices

Square $n \times n$ matrices are easier to transpose in-place than rectangular matrices. We describe some well-known techniques related to square in-place transposition [5].

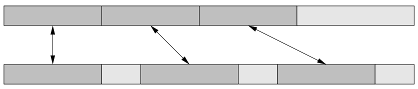
3.1.1 Basic Algorithm

Element $A(i,j)$ of a square $n \times n$ matrix in CM format is moved from location $i+jn$ to location $in+j$. Similarly, element $A(j,i)$ moves from location $in+j$ to location $i+jn$. Therefore, the diagonal elements are not moved at all, whereas the off-diagonal elements $A(i,j)$ and $A(j,i)$ are swapped. Cache blocking must be used to get good performance since in a naive implementation at least one matrix is accessed with a large stride.

3.1.2 Pad Transpose

If a matrix is nearly square, we can pad the matrix so that it becomes square and apply any square in-place transposition algorithm [5]. This is only practical if we can use $|m-n| \cdot \max(m,n)$ storage locations directly following the matrix in memory (something which is impossible in many codes).

```
Algorithm: PACK
for j = 1 to n-1
   for i = 0 to m-1
      [i+j*m] = [i+(j+x)*m]
```



```
Algorithm: UNPACK
for j = n-1 downto 1
   for i = m-1 downto 0
      [i+(j+x)*m] = [i+j*m]
```

Figure 3: Illustration of the PACK and UNPACK algorithms used to implement pad and cut transposes.

The notation $[x]$ in Figure 3 refers to the element at storage location x.

- If $m > n$, pad with $x = m - n$ columns. After transposition, the padded elements make up the x last rows of the matrix and are hence scattered in memory. By applying the `PACK` algorithm (Figure 3) the padded elements are removed and only the transposed original matrix remains.
- If instead $n > m$, pad with $x = n - m$ rows by applying the `UNPACK` algorithm (Figure 3). After transposition, the padded elements make up the x last columns of the matrix and can be safely ignored.

Besides having to transpose a slightly larger matrix, the pad transpose also requires an additional sweep through the matrix.

3.1.3 Cut Transpose

By removing instead of adding elements, the requirement on additional storage directly following the matrix is not mandatory. This technique is called a cut transpose [5].

- If $m > n$, cut away $x = m - n$ rows by applying the `PACK` algorithm (first making sure to save the cut-off elements). After transposition, the cut-off elements make up the x last columns and can be copied back to their correct locations.

- If instead $n > m$, cut away $x = n - m$ columns and save the cut-off elements. After transposition, the cut-off elements make up the x last rows and room for them is created by the `UNPACK` algorithm. After unpacking, the cut-off elements can be copied back to their correct locations.

The primary cost of a cut transpose is the extra sweep through the matrix.

3.2 Algorithms that Follow Cycles

Element $A(i, j)$ is stored at location $k = i + jm$ and after transposition it has moved to location $\bar{k} = in + j$. There is a simple form for the mapping from k to $\bar{k}$:

$$\bar{k} = P(k) = \begin{cases} kn \mod M & \text{if } 0 \le k < M, \\ M & \text{if } k = M, \end{cases} \tag{1}$$

where $M = mn - 1$ [3, 4]. The inverse mapping turns out to be more useful in practice and it can be shown that

$$k = P^{-1}(\bar{k}) = \begin{cases} \bar{k}m \mod M & \text{if } 0 \le \bar{k} < M, \\ M & \text{if } \bar{k} = M. \end{cases} \tag{2}$$

Transposition is a permutation and every permutation can be factored into a product of disjoint cycles. Due to the special structure of P^{-1} a cycle starting at s has a *companion cycle* (or sometimes *dual cycle*) starting at $M - s$ [3]. In some cases, the two cycles coincide and s is said to be *self-dual*. A *cycle leader* is any unique representative of a cycle (e.g., its minimum element). For example, the cycle factorization of P^{-1} for the 5×3 transposition problem is

$$(0)(1\ \ 5\ \ 11\ \ 13\ \ 9\ \ 3)(7)(2\ \ 10\ \ 8\ \ 12\ \ 4\ \ 6)(14).$$

The cycle leaders (the minimum elements) are 0, 1, 2, 7, and 14. There are three singleton cycles: 0, 7, and $14 = M$ and two self-dual cycles. The first cycle has leader $s_1 = 1$ and companion leader $M - s_1 = 13$, while the other cycle has leader $s_2 = 2$ and companion leader $M - s_2 = 12$. Cycle-following algorithms shift the elements of each cycle and previous research has focused on how to reduce the overhead of finding the cycle leaders. There is basically no spatial data locality when shifting a cycle, a fact that can be partly appreciated by observing that the definition of P^{-1} is similar to that of a Park-Miller linear congruential random number generator. Therefore, previously published cycle-following algorithms are of little practical interest on today's computer architectures with deep memory hierarchies.

A single cycle is shifted efficiently by Algorithm 1 which uses the inverse mapping P^{-1}. The notation $[x]_y$ is used to denote the vector of y contiguous elements starting at memory location xy (compare with Figure 3).

It turns out that self-dual cycles always meet in the middle [4, Theorem 7]. In other words, if one starts at s and $M - s$ and simultaneously traverses both cycles, then one will arrive at $M - s$ outgoing from s at the same step that one arrives at s outgoing from $M - s$. If the cycles are not self-dual, then one will complete their cycles at the same step since they have the same length. This symmetry result has been used in [3, 4, 11] to efficiently shift both cycles simultaneously. See Algorithm 2 for an implementation.

Algorithm 1 Cycle Shifting

```
Input: The cycle leader $s$ and the vector length $L$.
1: $a_1 := s$
2: $t := [a_1]_L$
3: $a_2 := P^{-1}(a_1)$
4: while $a_2 \neq s$ do
5:     $[a_1]_L := [a_2]_L$
6:     $a_1 := a_2$
7:     $a_2 := P^{-1}(a_1)$
8: end while
9: $[a_1]_L := t$
```

Algorithm 2 Simultaneous Cycle and Companion Cycle Shifting

```
Input: The cycle leader $s$, $M = mn - 1$, and the vector length $L$.
1: $a_1 := s$
   $\hat{a}_1 := M - s$
2: $t := [a_1]_L$
   $\hat{t} := [\hat{a}_1]_L$
3: $a_2 := P^{-1}(a_1)$
   $\hat{a}_2 := M - a_2$
4: loop
5:     if $a_2 = s$ then
6:         The cycle and its companion are distinct.
7:         $[a_1]_L = t$
           $[\hat{a}_1]_L = \hat{t}$
8:         break
9:     end if
10:    if $\hat{a}_2 = s$ then
11:        The cycle is self-dual.
12:        $[a_1]_L = \hat{t}$
           $[\hat{a}_1]_L = t$
13:        break
14:    end if
15:    $[a_1]_L := [a_2]$
       $[\hat{a}_1]_L := [\hat{a}_2]$
16:    $a_1 := a_2$
       $\hat{a}_1 := \hat{a}_2$
17:    $a_2 := P^{-1}(a_1)$
       $\hat{a}_2 := M - a_2$
18: end loop
```

One approach to finding the cycle leaders is to scan through the elements and use a boolean table with mn entries to record which elements have been moved. The cycle leader test reduces to a table lookup. This approach is memory intensive for floating point matrices unless the table can be embedded into an unused bit in each element.

An approach which does not require any additional memory uses the minimum element as the cycle leader. For each possible cycle leader s in the sequence $0, \dots mn-1$, traverse its cycle until either $t < s$ is encountered (in which case s is rejected) or s is encountered again, completing the cycle and showing that s is the minimum element in its cycle. The computational cost of this approach is significant and makes this approach impractical. However, the idea to traverse the cycle to find its minimum element is useful and is called the *general cycle test* in what follows.

Brenner [3] used number theory results to study the transposition permutation. He showed that all elements in a cycle starting at s are divisible by $d = \gcd(s, M)$ and not divisible by any other larger divisor of M [3, Theorem 1]. We associate all $\phi(M/d)$ such elements with d, where ϕ is Euler's phi function. For each divisor d of M, successively larger multiples of d are considered as possible cycle leaders until all $\phi(M/d)$ elements associated with d have been shifted. Experience has showed that Brenner's algorithm can greatly reduce the overhead of finding cycle leaders. We have adopted Brenner's results as the basis for our implementation.

In practice, cycle-following algorithms are hybrid methods that use a boolean table of limited size, typically with only $(m+n)/2$ entries. The table covers the first few possible cycle leaders. For larger candidates the general test is used. Experience indicates that the transposition often completes before the table is overrun.

ACM Algorithm 302 [2] can also be categorized as a cycle-following algorithm. However, it uses a fundamentally different algorithm for shifting cycles. The algorithm does not appear to be as efficient as either ACM Algorithm 467 or ACM Algorithm 513 [4].

3.3 Variants of Eklundh's Algorithm

Eklundh [6] developed an algorithm for transposing large square $2^n \times 2^n$ matrices out-of-core. His method uses only a small amount of additional in-core memory. The general idea starts with a 2×2 block partitioning:

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

After transposing each block in-place recursively the storage contains the matrix

$$\begin{pmatrix} A_{11}^T & A_{12}^T \\ A_{21}^T & A_{22}^T \end{pmatrix}.$$

To complete the transposition the blocks A_{12}^T and A_{21}^T are swapped. Eklundh pointed out that the entire process can be performed from the simple building block of reading two rows into core memory, swapping some elements, and writing the two rows back in the same place. He combined this simple operation with some intricate index manipulations and arrived at an ingenious non-recursive implementation.

Eklundh's algorithm has since been extended to rectangular matrices and more general composite dimensions in [18]. Variants of Eklundh's algorithm appear to require a larger number of sweeps than cycle-following algorithms and is therefore unlikely to be competitive when the matrix fits in main memory. See [14] for more variants of Eklundh's algorithm.

3.4 Other Algorithms

Murray Dow reviewed several transposition techniques in [5], including the pad and cut transposes. Two block algorithms suitable for vector computers were also presented. The first algorithm (V4), which Dow attributes to Markus Hegland, applies when $m = Mm_b$. The matrix is partitioned into an $M \times n$ block matrix with blocks of size $m_b \times 1$. The blocks are first transposed and then their elements are reordered to complete the transposition. For an example, see [5, Algorithm V4] or Appendix A.

Dow's second block algorithm (V5) partitions both dimensions and applies when $D \equiv \gcd(m, n) > 1$ [5, Algorithm V5]. The dimensions are factored into $m = Dm_b$ and $n = Dn_b$ and partitions the matrix into a square $D \times D$ block matrix with blocks of size $m_b \times n_b$. The first step of the algorithm transposes each block. This is reported to require mn_b additional memory locations. The second step transposes the block matrix using a square in-place transpose algorithm requiring no additional storage.

A three-stage transposition algorithm is presented by Alltop in [1]. It also factors $m = Dm_b$ and $n = Dn_b$ and partitions the matrix into a $D \times D$ block matrix with blocks of size $m_b \times n_b$. The first step transposes the square $D \times D$ block matrix. The second and third steps taken together transpose the individual blocks and are implemented by out-of-place rectangular transpositions using $Dn_bm_b = nm_b$ and $Dn_b = n$ additional elements, respectively.

4 Transposition as Matrix-Vector Multiplication

The Kronecker product $A \otimes B$ of the $m \times n$ matrix A and the $p \times q$ matrix B is an $mp \times nq$ matrix with $a_{ij}B$ as its (i, j)-th element. The vec operator forms a vector by stacking the columns of a matrix underneath eachother [12]. Thus with $a_{\bullet j}$ denoting the j-th column of the matrix A

$$\operatorname{vec} A = \begin{bmatrix} a_{\bullet 0} \\ \vdots \\ a_{\bullet n-1} \end{bmatrix}.$$

The order of the elements corresponds to the layout in memory when A is stored in CM format. Transposing A amounts to permuting $\operatorname{vec} A$ into $\operatorname{vec} A^T$ by multiplication with a permutation matrix

$$\operatorname{vec} A^T = L_m^{n \cdot m} \operatorname{vec} A.$$

The so-called *vec-permutation* matrix $L_m^{n \cdot m}$ is $nm \times nm$ and permutes an nm-vector by taking every m-th element of the vector starting with the first, then every m-th element starting with the second, and so on. An alternative definition of $L_m^{n \cdot m}$ is as the permutation matrix which verifies

$$L_m^{n \cdot m}(e_j^n \otimes e_i^m) = e_i^m \otimes e_j^n,$$

where, for example, e_i^n denotes the i-th unit vector (counting from zero) of length m. Yet another definition is constructive:

$$L_m^{n \cdot m} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} e_i^m (e_j^n)^T \otimes e_j^n (e_i^m)^T.$$

The vec-permutation matrix $L_3^{5 \cdot 3}$ is illustrated in Figure 4. For a review of

$$\left[\begin{array}{ccc|ccc|ccc|ccc|ccc}
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\hline
\cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
\hline
\cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1
\end{array}\right]$$

Figure 4: The vec-permutation matrix $L_3^{5 \cdot 3}$. The dot-elements are zeroes.

the history and properties of the Kronecker product and the vec-permutation matrix in particular see [12].

5 Blocked Transposition

With blocked transposition we consider algorithms that primarily read and write contiguous storage locations. In this section, we explain how certain permutations can be implemented with in-place transposition algorithms (adapted to move contiguous vectors) as building blocks.

We follow the approach of Fraser [8] and others and view storage locations in a *mixed-radix number system* and consider digit permutations. We use the notation $[r_1, r_0](d_1, d_0)$ to specify the radices r_1 and r_0 as well as the digits d_1 and d_0 in a mixed-radix number system. Another way to specify mixed-radix numbers is to subscript each digit with its radix. We stick to the former notation since it separates radices from digits.

For numbers with four positions, which is primarily what we are using, the definition of a mixed-radix number is

$$[r_3, r_2, r_1, r_0](d_3, d_2, d_1, d_0) = d_3 r_2 r_1 r_0 + d_2 r_1 r_0 + d_1 r_0 + d_0, \qquad (3)$$

where the radices are greater than one ($r_i > 1$) and $d_i \in \{0, \ldots, r_i - 1\}$. With these conditions, every decimal number $x \in \{0, \ldots, r_3 r_2 r_1 r_0 - 1\}$ has a unique representation. Note that $r_3 = r_2 = r_1 = r_0 = 10$ corresponds to our decimal number system. The example below also illustrates an alternative way of presenting mixed-radix numbers using subscripted radices:

$$[4, 6, 5, 7](2, 5, 3, 4) = 2_4 5_6 3_5 4_7 = 2 \cdot (6 \cdot 5 \cdot 7) + 5 \cdot (5 \cdot 7) + 3 \cdot (7) + 4 = 620.$$

Consider again the block matrix

$$A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,N-1} \\ \vdots & \ddots & \vdots \\ A_{M-1,0} & \cdots & A_{M-1,N-1} \end{bmatrix}$$

where each block is of size $m_b \times n_b$. The element $A_{i_1,j_1}(i_2, j_2)$ is stored (assuming CM format) in the storage location with the mixed-radix number representation

$$[N, n_b, M, m_b](j_1, j_2, i_1, i_2) = (j_1 n_b + j_2) M m_b + (i_1 m_b + i_2). \tag{4}$$

If the same element instead was stored at the location

$$[N, M, n_b, m_b](j_1, i_1, j_2, i_2) = (j_1 M + i_1) n_b m_b + (j_2 m_b + i_2), \tag{5}$$

then the matrix A would have been stored in the CCRB storage format.

There is a connection between mixed-radix numbers and certain vectors built by Kronecker products. The number in (4) gives the position of the only non-zero element in the vector

$$e^N_{j_1} \otimes e^{n_b}_{j_2} \otimes e^M_{i_1} \otimes e^{m_b}_{i_2}. \tag{6}$$

Multiplying (6) with a particular permutation matrix

$$(I_N \otimes L^{n_b \cdot M}_M \otimes I_{m_b})(e^N_{j_1} \otimes e^{n_b}_{j_2} \otimes e^M_{i_1} \otimes e^{m_b}_{i_2}) = e^N_{j_1} \otimes e^M_{i_1} \otimes e^{n_b}_{j_2} \otimes e^{m_b}_{i_2}$$

commutes the two vectors in the middle of the Kronecker product (6) and the position of the only non-zero element of the resulting vector is now given by the number in (5).

Many algorithms can be expressed in terms of factorizations of the vec-permutation $L^{n \cdot m}_m$, including most algorithms discussed in this paper. See [14] (and also [13]) for an extensive study on the subject with many existing and some new algorithms and their relations to factorizations of the vec-permutation.

5.1 In-Place Blocked Transposition

In this section, we investigate three classes of permutations and their implementation with in-place transposition algorithms as building blocks. We show that swapping two adjacent digits can be implemented by a set of independent in-place transpositions that move contiguous vectors. Furthermore, we also show that swapping two non-adjacent digits can be implemented by a set of independent *square* in-place transpositions if the radices of the two digits are equal. Finally, we show how to fuse two adjacent digit swaps together provided they operate on different digits. This allows for an implementation which sweeps through the matrix once instead of two times.

We start by looking at swapping two adjacent digits d_k in (3). To indicate which digits we intend to swap we will make use of a so-called *transposition pattern*. The pattern $(j, i, \cdot, \cdot)$, for example, simply indicates that we intend to swap the third and fourth digits. A pattern for swapping two adjacent digits is called *regular*. Without loss of generality we apply the pattern $(\cdot, j, i, \cdot)$ to storage locations expressed in the mixed-radix number system

$$[r_3, r_2, r_1, r_0](d_3, \mathbf{j}, \mathbf{i}, d_0). \tag{7}$$

All elements addressed by $\mathbf{i} \in \{0, \ldots, r_1 - 1\}$ and $\mathbf{j} \in \{0, \ldots, r_2 - 1\}$ (keeping d_3 and d_0 fixed) are interpreted as an *embedded matrix* of size $r_1 \times r_2$. There are $d_3 d_0$ such embedded matrices in the example above, one for each choice of d_3 and d_0. We use this interpretation to highlight that swapping two digits can be implemented by transposing all of the embedded matrices in-place.

In the example above, we wish to permute the elements so that the element at location (7) is moved to location

$$[r_3, r_1, r_2, r_0](d_3, \mathbf{i}, \mathbf{j}, d_0). \tag{8}$$

Equation (7), which is the storage location *before* permutation, expands to

$$d_3 r_2 r_1 r_0 + d_0 + (\mathbf{j} r_1 + \mathbf{i}) r_0, \tag{9}$$

and (8), the storage location *after* permutation, expands to

$$d_3 r_2 r_1 r_0 + d_0 + (\mathbf{i} r_2 + \mathbf{j}) r_0. \tag{10}$$

The only difference is in the parenthesized expressions. Notice the connection between the parenthesized expressions and the CM storage mapping of the associated $r_1 \times r_2$ embedded matrix (call it B). The expression $(\mathbf{j} r_1 + \mathbf{i})$ in (9) is the location of $B(\mathbf{i}, \mathbf{j})$ prior to transposition and the expression $(\mathbf{i} r_2 + \mathbf{j})$ in (10) is the location of the same element after transposition. The key point is that we can implement an adjacent digit swap by adapting the memory references of any (in-place or out-of-place) transposition algorithm and repeat the algorithm for every choice of the digits d_3 and d_0. Moreover, from (10) we make the following observation. With d_3, $\mathbf{i}$, and $\mathbf{j}$ given, the elements corresponding to all choices of d_0 are contiguous and maintain their relative order after the permutation. Therefore, we can move all r_0 such contiguous elements at once. Figure 5 is an illustration of an embedded matrix associated with the regular pattern $(\cdot, j, i, \cdot)$ when we interpret the matrix as being block-partitioned and in CM format. In summary, swapping two adjacent digits amounts to a set of independent (in-place) transpositions that move contiguous elements.

A pattern associated with swapping two non-adjacent digits is called *separated*. An example of a separated pattern is $(\cdot, j, \cdot, i)$ which swaps the first and the third digits. We use the mixed-radix number representation

$$[r_3, r_2, r_1, r_0](d_3, \mathbf{j}, d_1, \mathbf{i})$$

and expand the storage location before the permutation to

$$d_3 r_2 r_1 r_0 + \underline{d_1 r_0} + (\mathbf{j} r_1 r_0 + \mathbf{i}) \tag{11}$$

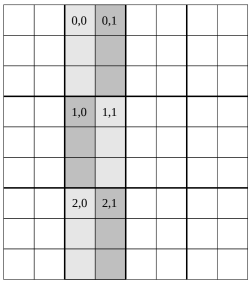


Figure 5: An illustration of the 3×2 embedded matrix (inside a 9×8 block-partitioned matrix) associated with $[3, 2, 3, 3](1, \mathbf{j}, \mathbf{i}, d_0)$, $d_0 = 0$. Each shaded vector of length three groups together the contiguous elements obtained by varying the choice of d_0 between 0 and $r_0 - 1 = 2$.

and the storage location after the permutation to

$$d_3 r_2 r_1 r_0 + \underline{d_1 r_2} + (\mathbf{i} r_2 r_1 + \mathbf{j}). \tag{12}$$

The embedded matrices are $r_0 \times r_2$. Note that the underlined term is slightly changed. This implies that if $r_0 \neq r_2$, then the first storage locations ($\mathbf{i} = \mathbf{j} = 0$) differ before and after the permutation is applied. Therefore, it is impossible to independently tranpose each embedded matrix in such cases. However, if the embedded matrices are indeed square (i.e., $r_0 = r_2$ in the example) then the two underlined terms are equal and it is easy to verify that independent in-place transpositions of each embedded matrix is possible. We remark that swapping two non-adjacent digits can be performed by a sequence of adjacent digit swaps and that a Kronecker product factorization of the permutation matrix can be used to express the same result. In summary, swapping two non-adjacent digits (i.e., applying a separated transposition pattern) is possible to implement using square in-place transposition of the embedded matrices assuming they are square.

The final type of transposition pattern that we have identified is the *fused* pattern (j, i, j, i). The double set of indices indicate that there are two sets of embedded matrices and the fused pattern is indeed just a combination of the two regular patterns $(j, i, \cdot, \cdot)$ and $(\cdot, \cdot, j, i)$. Each contiguous set of elements in the former pattern corresponds to an embedded matrix in the latter pattern. Hence, an implementation of the fused pattern could perform the transpositions associated with the latter pattern $(\cdot, \cdot, j, i)$ while moving elements as a part of the former pattern $(j, i, \cdot, \cdot)$. The cost would essentially be half that of applying the two patterns individually since each element is fetched from memory only once. In Kronecker product notation, the idea of a fused pattern stems from

$$\underbrace{(L_{r_2}^{r_3 \cdot r_2} \otimes I_{r_1 r_0})}_{(j,i,\cdot,\cdot)} \underbrace{(I_{r_3 r_2} \otimes L_{r_0}^{r_1 \cdot r_0})}_{(\cdot,\cdot,j,i)} = \underbrace{L_{r_2}^{r_3 \cdot r_2} \otimes L_{r_0}^{r_1 \cdot r_0}}_{(j,i,j,i)}.$$

The table below summarizes the seven possible transposition patterns.

Pattern	Comment	Name
$(\cdot,\cdot,j,i)$	Pointwise	Regular
$(\cdot,j,i,\cdot)$		Regular
$(j,i,\cdot,\cdot)$		Regular
$(\cdot,j,\cdot,i)$	$r_2 = r_0$	Separated
$(j,\cdot,\cdot,i)$	$r_3 = r_0$	Separated
$(j,\cdot,i,\cdot)$	$r_3 = r_1$	Separated
(j,i,j,i)	$r_1 r_0$ is small	Fused

We remark that the separated pattern $(\cdot,j,\cdot,i)$ can be implemented with limited additional memory as a sequence of out-of-place transpositions even if the embedded matrix is not square. This fact is used in, for instance, Dow's V5 algorithm.

6 Matrix Storage Format Conversions

All six of the storage formats reviewed in Section 2 can be succinctly described by a mixed-radix number system as in Section 5 using a block-partitioned matrix (i.e., $m = Mm_b$ and $n = Nn_b$). The ability to permute the digits is therefore all we need to do in-place conversion between each pair of formats. The six storage formats and some of the permutations that convert between them are shown in Figure 6. The following permutations are referred to in the diagram:

1 regular pattern $(\cdot,j,i,\cdot)$,

2 regular pattern $(j,i,\cdot,\cdot)$,

3 regular pattern $(\cdot,\cdot,j,i)$.

The diagonal arrows in the middle of the diagram represent the fused pattern (j,i,j,i) which is the combination of 2 and 3.

Conversion between two block formats with mismatching block sizes can be implemented by first converting to CM/RM (whichever is closer in the lattice) and from there to the output format. This is the strategy we use in our conversion software. It is an open question whether special relationships between mismatching block sizes (such as the output block size being a multiple of the input block size) can be exploited to improve performance.

7 Three-Stage Algorithm for Transposition

Below we describe an interesting algorithm for in-place transposition, which was mentioned already in [14] and brought to our attention by Fred Gustavson [10]. Bold radices and digits indicate which digits that are about to be swapped.

$$\begin{array}{ll}
[N, \mathbf{n_b}, \mathbf{M}, m_b](j_1, \mathbf{j_2}, \mathbf{i_1}, i_2) & \text{Stage A} \\
[\mathbf{N}, \mathbf{M}, n_b, m_b](\mathbf{j_1}, \mathbf{i_1}, j_2, i_2) & \text{Stage B1} \\
[M, N, \mathbf{n_b}, \mathbf{m_b}](i_1, j_1, \mathbf{j_2}, \mathbf{i_2}) & \text{Stage B2} \\
[M, \mathbf{N}, \mathbf{m_b}, n_b](i_1, \mathbf{j_1}, \mathbf{i_2}, j_2) & \text{Stage C} \\
[M, m_b, N, n_b](i_1, i_2, j_1, j_2) &
\end{array}$$

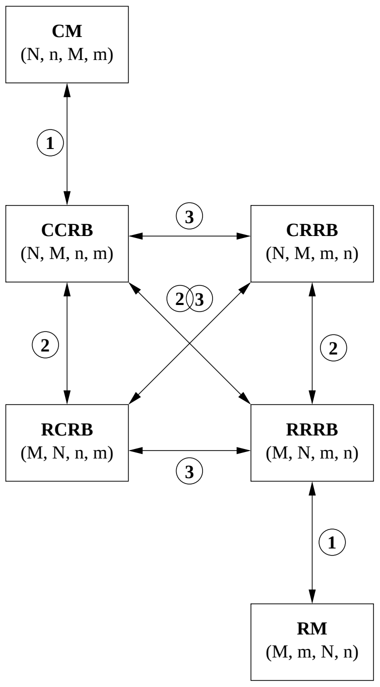


Figure 6: Conversion lattice showing six storage formats, their radices in a mixed-radix number system, and some of the permutations (circled numbers) that convert between them.

Assuming that m_b and n_b are reasonably large, say between 50 and 100 (see Section 9.3.1), then the following item list includes some of the reasons to why this algorithm can be expected to be efficient.

- Stage A results in N embedded matrices with vector length m_b.
- Stage B1 results in one embedded matrix with vector length $m_b n_b$.
- Stage B2 results in MN embedded matrices that are small enough ($m_b n_b$ elements) to be transposed out-of-place.
- Stage C results in M embedded matrices with vector length n_b (compare with Stage A).
- Stages B1 and B2 can be fused, reducing the number of sweeps through the matrix from four to three.
- Small cuts on the rows and/or columns can be used to ensure that the block sizes m_b and n_b are suitable (neither too small nor too large).
- As few as three sweeps (no cuts) and at most five sweeps (cuts of both rows and columns) through the matrix are required.

Note that all stages result in in-place transposition problems that are substantially smaller than the original $m \times n$ problem.

The algorithm is easily understood with the help of the conversion lattice in Figure 6. Stage A implements a conversion from CM to CCRB format, Stage B converts from CCRB to RRRB, and finally Stage C converts from RRRB to RM format (i.e., the matrix has been transposed). Take a 9×6 matrix with blocks of size 3×2 as an example (Figure 1). Stage A converts to CCRB format (Figure 2). Stages B converts from CCRB to RRRB (Figure 7). Finally, Stage C

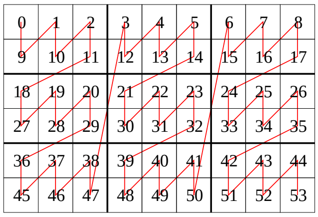


Figure 7: Configuration of a 9×6 matrix after Stage B. The matrix is now transposed but in CCRB format (i.e., the original matrix is in RRRB format).

converts from RRRB to RM (Figure 8).

8 Software

We have developed a software package written in C99 for the purpose of providing fast in-place transposition of large rectangular matrices and in-place conversion between storage formats. Some of the features of the package are listed below.

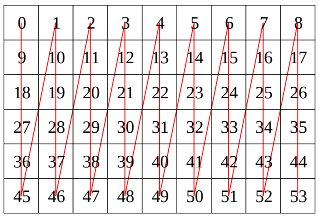


Figure 8: Configuration of a 9×6 matrix after Stage C. The matrix is now transposed and in CM format (i.e., the original matrix is in RM format).

- Portable since it is written in C99.
- Uses the Three-Stage Algorithm for transposition and cycle-following algorithms for in-place permutation.
- Uses the ideas of Brenner [3] to prune the search for cycle leaders.
- Exploits square transposition algorithms when any intermediate matrix is square.
- Automatically selects suitable block sizes m_b and n_b by finding the largest divisor of m such that $b_{\text{low}} \leq m_b \leq b_{\text{high}}$ and similarly for n_b.
- The parameters b_{low} and b_{high} enable tuning to a particular machine.
- Uses small cuts of rows and columns to give improved performance when no appropriate block sizes can be chosen (for example, when m and/or n are prime).
- Driver routines for in-place conversion back and forth between all 15 pairs of these storage formats:
  - CM/RM
  - CCRB/CRRB
  - RCRB/RRRB

9 Computational Experiments

Extensive experiments have been carried out on two different architectures. On each machine, all cores of a node were reserved for the application and only one core was actually used during each test. We present performance figures for the Three-Stage Algorithm together with comparisons with Dow's V5 algorithm, Alltop's three-stage algorithm, and out-of-place transposition. We also present a qualitative study of various cycle-following algorithms on a large set of problems.

9.1 Machines

We have performed our experiments using two different machines at the HPC2N facility in Umeå, Sweden. Since the execution of matrix transposition is ultimately memory bound we have benchmarked both systems using two benchmarks that are inspired by the STREAM benchmark [16]:

- **Copy** measures the time it takes to copy a large vector of double precision numbers ($y \leftarrow x$).
- **Scale** measures the in-place scaling of a double precision vector ($x \leftarrow \alpha{\cdot}x$). The multiplication with a scalar is merely to hinder the compiler from optimizing away the entire loop body. There should be plenty of spare clock cycles available to hide the overhead of the multiplication.

It is important to notice that the benchmarks are not intended to measure the peak hardware memory bandwidth but to establish the practical peak bandwidth obtainable by our particular combination of hardware, compiler, and compiler optimizations. Both benchmarks are implemented as one-statement for-loops in C99 and verification of the assembler output shows that the compiler does unrolling and vectorization. In contrast with the STREAM benchmark, our code uses dynamically allocated memory and the vector size is determined at runtime. This makes the benchmark more similar to a typical usage pattern.

We chose these two benchmarks to capture two fundamentally different usage patterns. In an out-of-place transposition, the matrix is copied from one memory area to another and then copied back. If the matrix is large, then the memory written to does not reside in the cache. This usage pattern is captured by the Copy benchmark. A cycle-following in-place transposition, on the other hand, moves data around cycles. The memory that is written to was recently read and hence is likely to be found in the cache. This usage pattern is similar to that in the Scale benchmark.

Some characteristics of the two machines are given in Table 1. The benchmark figures are the time divided by the length of the vectors. The Copy benchmark is roughly 58% slower than the Scale benchmark on Akka and roughly 43% slower on Sarek.

Name	*Akka*	*Sarek*
Processor	Dual Intel Xeon QC L5420	Dual AMD Opteron 248
Frequency	2.5 GHz	2.2 GHz
Memory	16 GB	8 GB
Compiler	PathScale 3.1	PathScale 3.1
Switches	`-O3 -march=auto`	`-O3 -march=auto`
BM: Copy (t_{copy})	4.845 ns (3098 MB/s)	6.731 ns (2265 MB/s)
BM: Scale (t_{scale})	3.067 ns (4925 MB/s)	4.708 ns (3240 MB/s)

Table 1: Characteristics of the HPC2N machines used for the experiments.

Many cache-based systems use a *write-allocate* strategy when writing to memory that is not cached. The write-allocate strategy means that the hardware reads the cache line into cache prior to the write. This has the side-effect that a write to uncached memory requires twice the memory bandwidth compared to a memory read or a cached write. The vector extension SSE (used on our machines) provides special instructions to write directly to memory (so-called *non-temporal* instructions). Nonetheless, the Copy and Scale benchmarks indicate that a dramatic difference can be observed.

9.2 Qualitative Study of Cycle-Following Algorithms

The general cycle test employed by many cycle-following algorithms (recall Section 3) is one of the largest sources of overhead in such algorithms. To get an idea of how significant this overhead may be and how it can be reduced by using a hybrid method with a limited lookup table, we experimented with three different algorithms using three different table sizes on all of the 61752 rectangular matrices $m \times n$ with $m, n \in \{2, \dots, 250\}$. We counted the number of times (α) that P^{-1} was evaluated during a general cycle test. The number α varies considerably between problems, so to get an overview we computed two statistical quantities:

$$\max\left(\frac{\alpha}{mn}\right) \quad \text{and} \quad \operatorname{avg}\left(\frac{\alpha}{mn}\right).$$

The maximum and average are taken across all 61752 problems. The ideal scenario is for both of these to be close to zero. An average of one means that we expect to calculate P^{-1} approximately twice as many times as necessary (roughly half of them during the cycle shifting and the other half during the general tests).

The algorithms we considered were ACM Algorithm 467 [3] (A467), ACM Algorithm 513 [4] (A513), and a simplified variant of ACM Algorithm 467 that does not take advantage of companion cycles (A467s). For each of these algorithms the table sizes 0, $(m+n)/2$, and 100 were used. The results are reported in Table 2. The worst case for A467 was 2.19 without any table, which is not

Algorithm	A513			A467			A467s		
Table Size	0	$\frac{m+n}{2}$	100	0	$\frac{m+n}{2}$	100	0	$\frac{m+n}{2}$	100
$\max\left(\frac{\alpha}{mn}\right)$	6.89	2.77	3.40	2.19	1.46	1.58	4.28	3.14	3.31
$\operatorname{avg}\left(\frac{\alpha}{mn}\right)$	1.79	0.72	0.78	0.42	0.07	0.07	1.24	0.24	0.27

Table 2: Comparison of the number of times that P^{-1} is evaluated as part of general cycle tests for three hybrid cycle-following in-place transposition algorithms using three different table sizes.

so alarming since evaluating P^{-1} is not so expensive. Looking at the average values, going from no table to a table of size $\frac{m+n}{2}$ does bring a substantial reduction. For A467 the average is reduced by a factor of 6.0, for A467s the factor is 5.17, while for A513 it is only 2.49. The range of problems studied above is relevant in our context since the Three-Stage Algorithm with $m_b = n_b = 50$ requires the solution of transposition problems that are within the considered range for all $m, n \in \{1, \dots, 12500\}$.

We also measured the number of cycles (β) for each of the problems. This is an interesting problem characteristic which also varies considerably across the problem space. We found that

$$\max\left(\frac{\beta}{mn}\right) \approx 0.33 \quad \text{and} \quad \operatorname{avg}\left(\frac{\beta}{mn}\right) \approx 0.01.$$

Thus, the average cycle length is approximately $\frac{mn}{0.01mn} = 100$.

9.3 Evaluation of the Three-Stage Algorithm

The performance of pointwise cycle-following algorithms is very poor due to high overhead per element and a seemingly random memory access pattern. Our experiments indicate that the execution time is typically between 5 to 10 times longer compared to the Three-Stage Algorithm. Therefore, we instead compare with an out-of-place transposition and implementations of Dow's V5 algorithm and Alltop's three-stage algorithm.

We begin by evaluating different aspects of the Three-Stage Algorithm before moving on to algorithm comparisons in Section 9.4.

9.3.1 Block Size

The block size can not be chosen freely since the possible block sizes depend on the problem size. However, cuts can be applied to alter the problem size. It is important to understand how the problem size impacts performance in order to determine when it is economical to pay for the overhead of cutting. We designed an experiment where we modified a fixed problem size so that both dimensions are divisible by some common factor (the block size). By doing this for all block sizes from 1 to 200 we get results for a collection of almost equally large problems for a range of block sizes. By comparing the *time per element* ($\frac{T}{mn}$) instead of actual execution time we reduce the impact of the different problem sizes. The results on Akka displayed in Figure 9 show the total time as well as the time for each of the three stages. The results on Sarek are qualitatively similar to the results on Akka. There is a peak at $m_b = n_b = 128$ in Stage B which is

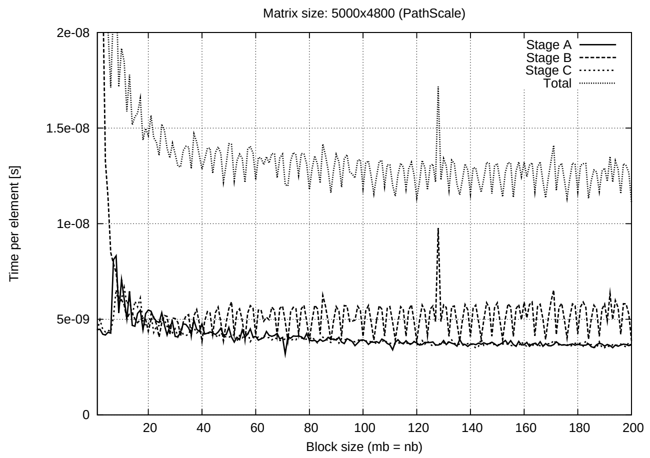


Figure 9: Performance breakdown on Akka of the Three-Stage algorithm on a fixed problem (5000×4800) for all block sizes ($m_b = n_b$) in the range $1, \ldots, 200$.

likely caused by cache thrashing in the small out-of-place transpositions. Other than that, a block size larger than 30 gives good performance whereas smaller block sizes should be avoided, mainly due to the performance of Stage B which eventually turns into a pointwise cycle-following algorithm when $m_b = n_b = 1$.

9.3.2 Cutting and Other Overhead

Cuts require an additional sweep over the matrix and will therefore impact performance. Similar to our experiment with different block sizes we started from a fixed problem size and a fixed block size and modified the problem size to induce cuts of size 1 to 99 in both rows and columns. The performance of the pre- and post-processing steps (the cuts) on Akka are reported in Figure 10 (the results on Sarek are qualitatively similar). Note that the performance is

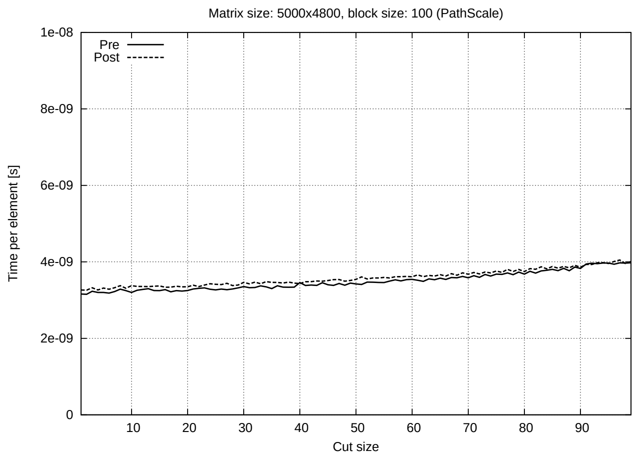


Figure 10: Performance on Akka of the pre- and post-processing steps during a symmetric cut on a fixed problem (5000×4800).

comparable to the three stages as reported in Section 9.3.1 and the introduction of a cut costs alot whereas a large cut does not cost significantly more than a small cut.

An interesting aspect of the implementation is the observed overhead in the cycle-following part of the algorithm. We measured the time spent in cycle shifting in addition to the total time and Figure 11 shows the results on Akka for various block sizes. For larger block sizes the overhead is insignificant whereas for smaller block sizes the implementation might be improved by introducing a lookup table.

9.4 An Evaluation of Transposition Algorithms

There are many parameters that affect the performance of transposition algorithms. Examples include the implementation of an algorithm, choice of compiler and settings, different optimizations and machine characteristics. The problem size and the shape of the matrix also affect performance. Some problem sizes might require cuts or cause severe cache thrashing. Some algorithms have additional tuning parameters such as block sizes and thresholds.

We compare the Three-Stage Algorithm, Murray Dow's V5 algorithm, Alltop's algorithm, and two implementations of out-of-place transposition: one naive implementation and one tuned cache-blocked algorithm. We have per-

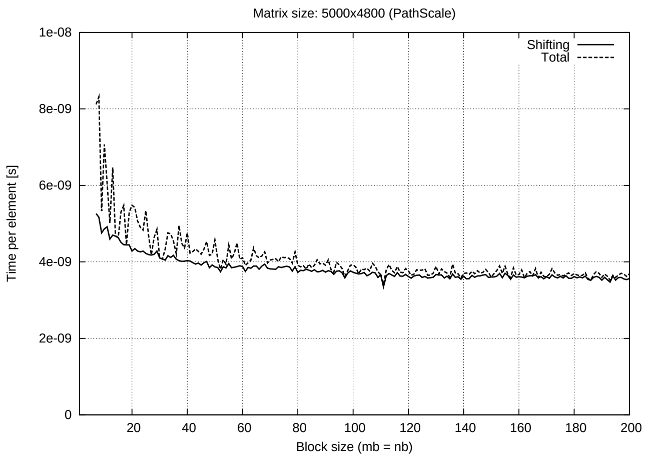


Figure 11: Performance breakdown on Akka of Stage A in the Three-Stage algorithm on a fixed problem (5000×4800). The time to do the actual shifting is shown together with the total time, which includes finding the cycle leaders. For larger block sizes the overhead is negligible but for some small block sizes the overheads are significant.

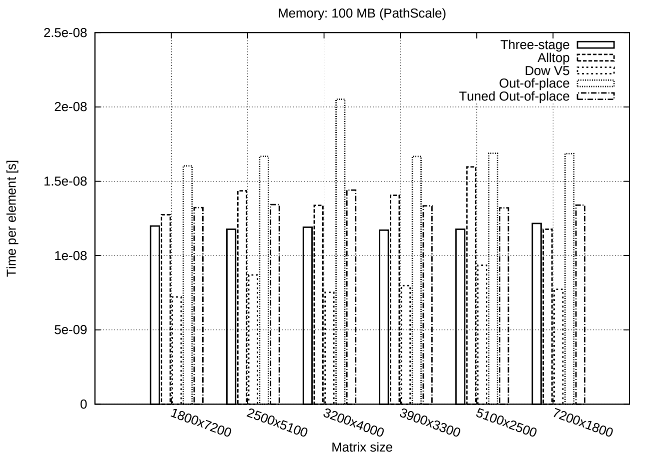


Figure 12: Performance comparison on Akka using matrices of different shapes that all require roughly 100 MB of memory.

formed many experiments on both machines to investigate and capture the behaviour of each algorithm. We have selected representative data that capture most of what we found during experimentation. In the listing below, we summarize some facts about our experiments.

- The block size for the Three-Stage Algorithm was set to $m_b = n_b = 100$ (a suitable block size based on the results in Section 9.3.1) and all problem sizes were multiples of 100. Thus, no cuts were required and a block size of 100 was used for all executions.
- The common divisor D in Alltop's algorithm and Dow's V5 algorithm was chosen optimally for each problem size.
- All experiments were repeated five times and only the best result was kept in order to minimize the noise from system activities.
- The out-of-place algorithm was optimized by the compiler, which is capable of cache-blocking transformations.
- The tuned out-of-place algorithm is an automatically tuned implementation of a blocked transposition where different block traversal schemes and cache block sizes were taken into account. We chose the best performing implementation on a 1200×1800 test problem and that implementation traverses the blocks and elements in row-major order and has a block size of 16×16.

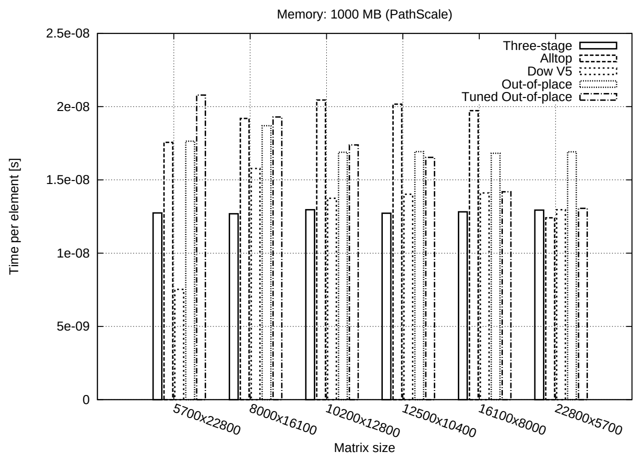


Figure 13: Performance comparison on Akka using matrices of different shapes that each require roughly 1000 MB of memory.

We observed that the memory footprint and the shape of the matrix affected performance significantly. We therefore performed experiments on matrices of different shapes having roughly the same memory footprint. For matrices requiring around 100 MB, the results on Akka are displayed in Figure 12. For

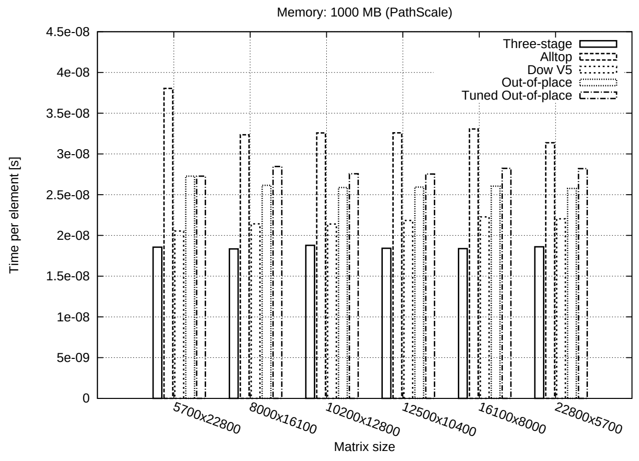


Figure 14: Performance comparison on Sarek using matrices of different shapes that each require roughly 1000 MB of memory.

larger matrices of around 1000 MB, the results on Akka are shown in Figure 13 and the corresponding results on Sarek are in Figure 14.

Dow's V5 algorithm is sometimes considerably faster than any of the other algorithms, especially for smaller matrices. The Three-Stage Algorithm has a relatively predictable performance (partly due to the fixed block size of $m_b = n_b = 100$) and is among the fastest, especially for large matrices. Alltop's algorithm appears to be rather inefficient for large matrices (see Figures 13 and 14).

The performance of Dow's V5 algorithm differs alot between small and large matrices and also for different shapes of large matrices. We think that this is partly due to data locality issues but also due to the worse performance of out-of-place transformations in general. In Table 3, we show two simple models of the execution time for each of the algorithms. The Three-Stage Algorithm

Algorithm	**Optimistic**	**Pessimistic**
Three-Stage	$3mn \cdot t_{\text{scale}}$	—
Dow's V5	$mn \cdot t_{\text{scale}} + mn \cdot t_{\text{scale}}$	$mn \cdot t_{\text{scale}} + 2mn \cdot t_{\text{copy}}$
Alltop	$2mn \cdot t_{\text{scale}} + mn \cdot t_{\text{scale}}$	$2mn \cdot t_{\text{scale}} + 2mn \cdot t_{\text{copy}}$
Out-of-place	$2mn \cdot t_{\text{copy}}$	—

Table 3: Models of the execution time under both an optimistic and a pessimistic scenario.

consists of three stages that are in-place (i.e., similar to Scale). The out-of-place algorithms have two stages that are both out-of-place (i.e., similar to Copy). The remaining algorithms have different models depending on the problem size. Dow's V5 algorithm has a first stage of in-place character and a second stage which is an out-of-place transformation of $Dm_bn_b = mn_b$ elements at a time. If the cache is large enough to retain the buffer until it is copied back, then the usage pattern is similar to the Scale benchmark in that the written memory resides in cache. This is the *optimistic* scenario. On the other hand, if the cache is not large enough, then the second stage consists of two operations similar to the situation in the Copy benchmark. This requires much more bandwidth (more than twice since $t_{\text{copy}} > t_{\text{scale}}$) and is the *pessimistic* scenario. A similar analysis has been done for Alltop's algorithm and all models are summarized in Table 3. Note that a multilevel cache hierarchy means that a transition from an optimistic to a pessimistic scenario will be gradual.

These models match the data pretty well. For example, most cases where Dow's V5 algorithm is considerably faster than the others are of the optimistic type whereas the other cases tend to be of the pessimistic type. Table 4 shows the prediction of each model on both machines. The figures in that table are for comparison with Figures 12, 13, and 14.

	Akka		**Sarek**	
Algorithm	Optimistic	Pessimistic	Optimistic	Pessimistic
Three-Stage	0.920	—	1.412	—
Dow's V5	0.613	1.276	0.942	1.817
Alltop	0.920	1.582	1.412	2.288
Out-of-place	0.969	—	1.346	—

Table 4: Model predictions on both machines (all figures should be multiplied by 10^{-8}). These figures are for comparison with Figures 12, 13, and 14.

10 Conclusions and Future Work

We have demonstrated that it is possible to develop cache-efficient in-place transposition algorithms based on generalized versions of previously known cycle-following algorithms. The performance of such algorithms rivals the best known semi-in-place algorithms by Alltop and Dow as well as out-of-place transposition. The write-allocate strategy of cache-based computer architectures, although in theory partly alleviated with non-temporal instructions, can be a performance problem in practice.

Conversion between the CM, RM, and the four block formats CCRB, CRRB, RCRB, and RRRB can be performed in-place using our software package which is based upon the techniques discussed in this paper. The implementation uses the cut transpose technique to give reasonable performance for cases when m and/or n do not enable a suitable choice of block size.

Evidence suggests that a hybrid implementation of the cycle-following kernel could be used instead of the pure implementation we currently have in order to reduce overhead.

Thread parallelization of independent subproblems and/or cycle shifts might improve performance on multicore architectures. The best algorithm for a particular problem depends on many parameters and there is no single best algorithm.

Acknowledgements

I would like to acknowledge Fred Gustavson for introducing me to the subject of in-place transposition and storage format conversions. He pointed out the merits of the Three-Stage Algorithm and together with Tadeusz Swirszcz he recently published improvements to cycle-following algorithms [11]. I also acknowledge Bo Kågström for his kind support, references to and comments on the subject, and constructive criticism on drafts of this manuscript.

References

[1] W. O. Alltop. A Computer Algorithm for Transposing Nonsquare Matrices. *IEEE Transactions on Computers*, 24(10):1038–1040, 1975.

[2] J. Boothroyd. Algorithm 302: Transpose Vector Stored Array. *Communications of the ACM*, 10(5):292–293, 1967.

[3] N. Brenner. Algorithm 467: Matrix Transposition in Place. *Communications of the ACM*, 16(11):692–694, 1973.

[4] E. G. Cate and D. W. Twigg. Algorithm 513: Analysis of In-Situ Transposition. *ACM Transactions on Mathematical Software*, 3(1):104–110, 1977.

[5] M. Dow. Transposing a Matrix on a Vector Computer. *Parallel Computing*, 21(12):1997–2005, 1995.

[6] J. O. Eklundh. A Fast Computer Method for Matrix Transposing. *IEEE Transactions on Computers*, 21(7):801–803, 1972.

[7] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. *SIAM Review*, 46(1):3–45, 2004.

[8] D. Fraser. Array Permutation by Index-Digit Permutation. *Journal of the ACM*, 23:298–309, 1976.

[9] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[10] F. G. Gustavson. The Relevance of New Data Structure Approaches for Dense Linear Algebra in the New Multicore/Manycore Environments. Technical Report RC24599, IBM Research, 2008. (Also submitted to PARA'08).

[11] F. G. Gustavson and T. Swirszcz. In-Place Transposition of Rectangular Matrices. In B. Kågström et al., editors, *Applied Parallel Computing. State of the Art in Scientific Computing, PARA 2006.* Lecture Notes in Computer Science, Vol. 4699, pages 560–569. Springer, 2007.

[12] H. V. Henderson and S. R. Searle. The vec-Permutation Matrix, the vec Operator and Kronecker Products: a Review. *Linear and Multilinear Algebra*, 9:271–288, 1981.

[13] J. R. Johnson. Matrix Transposition. Department of Mathematics and Computer Science, Drexel University, Philadelphia, PA 19104, 1995. (Manuscript).

[14] S. D. Kaushik, C. H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan. Efficient Transposition Algorithms for Large Matrices. In *Proceedings of Supercomputing '93*, pages 656–665, 1993.

[15] S. Laflin and M. A. Brebner. Algorithm 380: In-situ Transposition of a Rectangular Matrix. *Communications of the ACM*, 13(5):324–326, 1970.

[16] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995.

[17] N. Park, B. Hong, and V. K. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, 2003.

[18] H. K. Ramapriyan. A Generalization of Eklundh's Algorithm for Transposing Large Matrices. *IEEE Transactions on Computers*, 24(12):1221–1226, 1975.

A Reformulated Algorithms

Below is the V4 algorithm by Murray Dow [5, Algorithm V4] expressed in our notation:

$$\begin{array}{l} [\,\mathbf{n}\,,\mathbf{M},m_b](\,\mathbf{j}\,,\mathbf{i_1},i_2) \\ [M,\ \mathbf{n}\,,\mathbf{m_b}](i_1,\ \mathbf{j}\,,\mathbf{i_2}) \\ [M,m_b,\ n\ ](i_1,i_2,\ j\ ) \end{array}$$

The algorithm applies when $m = Mm_b$ with m_b being the block size. The final representation is correct since the memory location is

$$i_1 m_b n + i_2 n + j = (i_1 m_b + i_2)n + j = in + j$$

which agrees with the memory location of the matrix in RM format. Both steps can be performed in-place (regular transposition patterns).

Dow's V5 algorithm [5, Algorithm V5], which applies when $m = Dm_b$ and $n = Dn_b$, can be expressed as:

$$\begin{array}{l} [D,\mathbf{n_b},D,\mathbf{m_b}](j_1,\mathbf{j_2},i_1,\mathbf{i_2}) \\ [\mathbf{D},m_b,\mathbf{D},\ n_b\ ](\mathbf{j_1},i_2,\mathbf{i_1},j_2) \\ [D,m_b,D,\ n_b\ ](i_1,i_2,j_1,j_2) \end{array}$$

The first step can not be performed in-place since n_b and m_b are different (or otherwise the matrix would be square) and hence the embedded matrices are rectangular. The second step can be performed in-place since the embedded matrices are square.

The three-stage algorithm by Alltop [1], which applies when $m = Dm_b$ and $n = Dn_b$ is expressed below:

$$\begin{array}{ll}
[\mathbf{D}, n_b, \mathbf{D}, m_b](\mathbf{j_1}, j_2, \mathbf{i_1}, i_2) & \text{Stage A} \\
[D, n_b, \mathbf{D}, \mathbf{m_b}](i_1, j_2, \mathbf{j_1}, \mathbf{i_2}) & \text{Stage B1} \\
[D, \mathbf{n_b}, \mathbf{m_b}, D](i_1, \mathbf{j_2}, \mathbf{i_2}, j_1) & \text{Stage B2} \\
[D, m_b, \mathbf{n_b}, \mathbf{D}](i_1, i_2, \mathbf{j_2}, \mathbf{j_1}) & \text{Stage C} \\
[D, m_b, D, n_b](i_1, i_2, j_1, j_2) &
\end{array}$$

Stage A has a separated pattern but since it is square it can be performed in-place. Stages B1 and B2 together is actually just an adjacent digit swap in another mixed-radix number system, namely

$$[D, \mathbf{n_b D}, \mathbf{m_b}](i_1, \mathbf{j_2 D} + \mathbf{j_1}, \mathbf{i_2}).$$

Stages B and C are also possible to perform in-place but Alltop suggested using out-of-place transposition without mentioning the possibility of in-place transposition. The additional memory required is $n_b D m_b$ for Stage B and only $n_b D$ for Stage C.

IV

Paper IV

A Framework for Dynamic Node-Scheduling of Two-Sided Blocked Matrix Computations

Lars Karlsson[1] and Bo Kågström[1]

[1] *Department of Computing Science and HPC2N, Umeå University*
SE-901 87 Umeå, Sweden
{*larsk, bokg*}*@cs.umu.se*

Abstract: Blocked matrix algorithms are characterized by a high utilization of floating point units. Memory bandwidth is not a critical issue due to the surface-to-volume effect of level 3 algorithms. Factors limiting the performance of distributed algorithms include communication overhead and spurious synchronizations. Load balance can be achieved by using a 2D Block Cyclic Layout. To reduce communication overhead and synchronizations, a node algorithm is often rearranged into an efficient but more complicated variant. Frameworks for dynamic scheduling of node programs promise to remove much of the complexities while producing performance improvements. We present a design of a minimalistic framework for dynamic scheduling specifically targeting two-sided blocked matrix computations. A model algorithm, nonscalable in its straightforward implementation and with applications in modern algorithms for the nonsymmetric eigenvalue problem is shown to be scalable in practice. The scalability is enabled by the framework, specifically by the priority-based scheduling mechanism.

Key words: Distributed memory, dynamic scheduling, blocked matrix computations, priority-based scheduling, wavefront algorithm.

A Framework for Dynamic Node-Scheduling of Two-Sided Blocked Matrix Computations

Lars Karlsson and Bo Kågström

Department of Computing Science and HPC2N, Umeå University,
S-901 87 Umeå, Sweden, {larsk,bokg}@cs.umu.se

Abstract. Blocked matrix algorithms are characterized by a high utilization of floating point units. Memory bandwidth is not a critical issue due to the surface-to-volume effect of level 3 algorithms. Factors limiting the performance of distributed algorithms include communication overhead and spurious synchronizations. Load balance can be achieved by using a 2D Block Cyclic Layout. To reduce communication overhead and synchronizations, a node algorithm is often rearranged into an efficient but more complicated variant. Frameworks for dynamic scheduling of node programs promise to remove much of the complexities while producing performance improvements. We present a design of a minimalistic framework for dynamic scheduling specifically targeting two-sided blocked matrix computations. A model algorithm, nonscalable in its straightforward implementation and with applications in modern algorithms for the nonsymmetric eigenvalue problem is shown to be scalable in practice. The scalability is enabled by the framework, specifically by the priority-based scheduling mechanism.

1 Introduction

Blocked matrix algorithms on possibly hybrid distributed memory machines (the nodes themselves may be shared address space parallel computers) usually utilize a large fraction of the machine's peak performance [8]. The main limiting factors are communication overhead and spurious synchronizations, both between and within nodes. Communication overhead can often be reduced by rearranging the node algorithm to expose overlap possibilities via nonblocking communication. The overlap is then exploited in hardware via network interfaces with Direct Memory Access (DMA) which ultimately reduces the communication overhead [5]. Spurious synchronizations are handled similarly; by rearranging the node algorithm, spurious synchronizations can be removed.

The added complexity increases the likelihood of programming errors. For some algorithms it is also difficult to find an almost optimal schedule, which typically depends on machine parameters, problem and block sizes.

To remove most of the extra complexity one may use *dynamic scheduling.* In such dynamic implementations the execution order of different portions of the node program (its *tasks*) is *non-deterministic.* Some recent work on dynamic scheduling on shared address space and multicore architectures are [6,7,3,2].

Typically, a new bookkeeping overhead is introduced and some programmer assistance in decomposing the program into tasks as well as guidance on enforcing dependencies is required. A dynamic scheduling approach mainly benefits programmer productivity, maintainability, quality assurance, and portability. Performance for particular problems may often be suboptimal due to unexploited problem-specific optimizations.

In this contribution, we present a design of a minimalistic and efficient dynamic scheduling framework. The design is intended for blocked matrix computations on hybrid distributed memory machines. The tasks are statically distributed to processes and dynamic scheduling is applied only on the nodes.

We introduce a model algorithm which appears as the computationally dominant part in several algorithms related to the nonsymmetric eigenvalue problem [4,10,9,1]. A straightforward parallelization on a 2D Block Cyclic Layout (BCL) is nonscalable. This is not caused by load imbalance but rather by a suboptimal execution order that introduces spurious synchronizations. We take advantage of the priority-based scheduling in our framework to impose more efficient execution orders. The structure of an algorithm expressed in our framework retains its overall sequential appearance.

2 Framework Design

Our framework design consists of four major parts:

1. An API to express a node program in a familiar sequential style while allowing it to be executed by multiple threads.
2. An efficient algorithm and API that at runtime identifies data dependencies from programmer declarations of read and write accesses to matrix blocks.
3. A scheduling and dependency tracking mechanism.
4. An API and scheduling mechanism for asynchronous matrix-based communication built on top of MPI [11].

2.1 Application Programming Interface

The main API of the framework is used to annotate a correct sequential node program so that during execution it constructs a correct Directed Acyclic Graph (DAG) representation of the program. The DAG is then scheduled onto a team of threads consisting of one master thread (the thread that builds the DAG) and $w \geq 0$ worker threads. So far, in all experiments we have used $w = 0$ to demonstrate that the node-schedule greatly affects overall performance. The master thread handles all calls to MPI functions which means that the MPI implementation does not need to be thread-safe.

The framework considers two types of tasks: computation tasks (user-defined) and communication tasks (fully integrated).

Computation tasks are constructed from a sequence of API function calls. First a DAG node and its payload (user-defined task-specific information) are created, then all task dependencies are constructed, and finally the node is committed to the scheduler.

The scheduling and execution of communication tasks is fully integrated within the framework. Communication tasks are constructed using a separate API for matrix-based communication. The structure of task creation is the same as for computation tasks.

Data dependencies for a particular task are identified at runtime by the method described in Section 2.2. During task construction a sequence of API calls is used to specify read and write accesses to one or more matrix blocks.

2.2 Runtime Data Dependence Analysis

Due to the surface-to-volume effect in level 3 blocked matrix computations the number of floating point operations is often far greater than the number of memory references. The difference is even greater between floating point operations and *matrix block* references. This holds for the entire computation as well as for all of its level 3 subcomputations (such as `GEMM` updates, recursive panel factorizations, etc.) and is at the heart of the method described in this section.

Data dependencies may tightly couple otherwise independent parts of a large program. Manual analysis could therefore be prohibitively difficult. *Approximation* of data dependencies at *runtime* using matrix blocks as the unit of memory reference effectively solves the problem of coupling.

A task may request read-only access (read request) or read/write access (write request) to a block. Concurrent reads are allowed but writes serializes. Artificial DAG nodes (so called *join* nodes) are constructed to collect dependen-

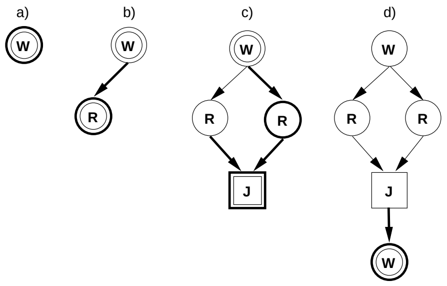


Fig. 1. Example showing how dependencies on a matrix block are efficiently constructed from a sequence of read and write requests (in this case: Write, Read, Read, Write). Double outlines denote nodes which may get new arcs and are hence retained as a part of the algorithm state. Bold arcs are the new dependencies that are added during the current step of the algorithm.

cies from concurrent reads to a single block in anticipation of a write request. This process is illustrated by example in Figure 1 and it allows the amount of memory required by the algorithm to remain constant.

3 Application Example

A model algorithm (Algorithm 1: `Sweep`) with applications in the nonsymmetric eigenvalue problem is described and analyzed in this section. We use our node-scheduling framework to effectively turn the straightforward implementation of `Sweep` into an efficient, scalable implementation which has two antidiagonal wavefronts running through the matrix.

3.1 Model Algorithm: Sweep

Algorithm 1 details the distributed algorithm from a global perspective. We use a special notation to specify which process or group of processes that are involved in each operation. With `P(i,j)` we mean the process that owns element `A(i,j)`. The notation `P(i,*)` refers to the *process row* that owns the *matrix row* `A(i,:)` and similarly `P(*,j)` refers to the *process column* that owns the *matrix column* `A(:,j)`. In each iteration, a diagonal block of size $n_b \times n_b$ is used to compute an $n_b \times n_b$ orthogonal matrix Q and is modified in the process by applying Q^T from the left and Q from the right. The iteration step is half the block size (the distribution block size is n_b) and the algorithm is assumed to start aligned with a block. At every other iteration the diagonal submatrix is local to one process (a *local* iteration, lines 7–15), whereas during the remaining iterations it is distributed onto four processes (a *cross-border* iteration, lines 17–37).

After the orthogonal matrix Q has been computed and applied to the diagonal block (line 8 or 18) the corresponding block row and column must also be updated in order to complete the orthogonal similarity transformation. Figure 2 shows the affected portions of the matrix during two iterations of the loop (one local and one cross-border).

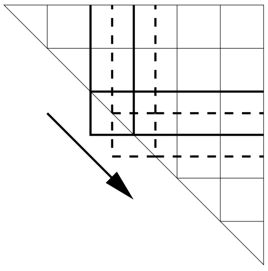


Fig. 2. Illustration of the model algorithm on a sample block matrix with $N = 6n_b$. Solid lines mark the region affected by iteration $i = 1 + 2n_b$. The diagonal block is used to compute Q and the corresponding block row and column are modified by the subsequent updates. Dashed lines show the half-block step taken to the next iteration ($i = 1 + 2.5n_b$) and is also an example of a cross-border iteration.

All of the block row and column operations (copy, update, and send/receive) are partitioned into independent suboperations at block boundaries. Each sub-

Algorithm 1 Sweep: Distributed Memory Model Algorithm. A is $N \times N$ and is paritioned into an $N_b \times N_b$ block matrix with blocks of size $n_b \times n_b$.

```
local = true
for i = 1 to N-nb+1 step nb/2
   a = i
   b = i + nb - 1
   c = i + nb/2 - 1
   if local then
      Copy A(a:b, a:b) to C on P(a, a)
      Compute Q from C on P(a, a)
      Copy C to A(a:b, a:b) on P(a, a)
      Broadcast Q to P(a, *) from P(a, a)
      Broadcast Q to P(*, a) from P(a, a)
      Copy A(1:a-1, a:b  ) to W on P(*, a)
      Copy A(a:b,   b+1:N) to S on P(a, *)
      Update A(1:a-1, a:b  ) = W*Q  on P(*, a)
      Update A(a:b,   b+1:N) = Q'*S on P(a, *)
   else
      Gather A(a:b, a:b) to C on P(a, a)
      Compute Q from C on P(a, a)
      Scatter C to A(a:b, a:b) from P(a, a)
      Partition Q into two column blocks: Q = [Q1, Q2]
      Send Q2 from P(a, a) to P(b, b)
      Broadcast Q1 to P(a, *) and P(*, a) from P(a, a)
      Broadcast Q2 to P(b, *) and P(*, b) from P(b, b)
      Partition W into two equal column blocks: W = [W1, W2]
      Partition S into two equal row    blocks: S = [S1; S2]
      Copy A(1:a-1, a:c  ) to W1 on P(*, a)
      Send A(1:a-1, a:c  ) to W1 on P(*, b) from P(*, a)
      Copy A(1:a-1, c+1:b) to W2 on P(*, b)
      Send A(1:a-1, c+1:b) to W2 on P(*, a) from P(*, b)
      Copy A(a:c,   b+1:N) to S1 on P(a, *)
      Send A(a:c,   b+1:N) to S1 on P(b, *) from P(a, *)
      Copy A(c+1:b, b+1:N) to S2 on P(b, *)
      Send A(c+1:b, b+1:N) to S2 on P(a, *) from P(b, *)
      Update A(1:a-1, a:c  ) = W*Q1  on P(*, a)
      Update A(1:a-1, c+1:b) = W*Q2  on P(*, b)
      Update A(a:c,   b+1:N) = Q1'*S on P(a, *)
      Update A(c+1:b, b+1:N) = Q2'*S on P(b, *)
   end if
   local = not local
end for
```

operation is regarded as an atomic unit of computation (copy and update) or communication (send/receive) and maps to one task.

We illustrate the general programming pattern of specifying a task and dependencies for a left update on the local block `A(il,jl)` (see line 15). We have abstracted from the details of the actual implementation to shorten the example and make its structure more clear.

```
task = AllocateComputationalTask()
ProcessReadRequest(task, Q)
ProcessWriteRequest(task, A(il,jl))
// Fill in task payload...
SetTaskPriority(task, myrow+il*Pr + mycol+jl*Pc)
CommitTask(task)
```

3.2 Analysis

We derive an approximate upper bound on the efficiency of the straightforward implementation. We assume that Q can be computed for free, that all updates perform at a fixed performance of α flops/s. Communication and other sources of overhead are not taken into consideration.

To perform all local block row updates (lines 14–15) over the course of the algorithm *sequentially* requires

$$T_{\text{local}} \approx \frac{N_b n_b^3 (N_b - 1)}{\alpha}$$

seconds. Similarly, to perform all cross-border block row updates (lines 34–37) over the course of the algorithm *sequentially* requires

$$T_{\text{cross}} \approx \frac{N_b n_b^3 (N_b - 2) + n_b^3}{\alpha}$$

seconds. The same timing models hold for block column updates.

When the algorithm executes in parallel on a $P_r \times P_c$ mesh the updates are assumed to be perfectly load balanced across the involved processes. This means that local block row and column updates are parallelized over P_c and P_r processes, respectively. The cross-border block row and column updates are parallelized onto $2P_c$ and $2P_r$ processes, respectively. Since, in every iteration, one process has to participate in both a block row and a block column update these two operations are effectively serialized. The estimate of the parallel execution time of the straightforward implementation is therefore approximated by

$$T_p \approx \frac{T_{\text{local}}}{P_c} + \frac{T_{\text{local}}}{P_r} + \frac{T_{\text{cross}}}{2P_c} + \frac{T_{\text{cross}}}{2P_r} \leq \frac{3T_{\text{local}}}{2} \frac{P_r + P_c}{P_r P_c}.$$

The sequential execution time is $T_s = 2T_{\text{local}} + 2T_{\text{cross}} \leq 4T_{\text{local}}$ which gives the following approximate bound on the efficiency,

$$E_p \leq \frac{8}{3(P_r + P_c)}.$$

3.3 Dual Wavefront Implementation

In order to appreciate the number of allowed schedules we look at a single block column of the matrix and consider a sequence of updates from the left. During the first iteration, `i=1` and the block column is updated on rows `1:nb`. During the second iteration, `i=1+nb/2` and the block column is updated on rows `1+nb/2:3*nb/2`. The affected rows overlap and there is a true data dependence that serializes the updates. The key thing to notice is that, while serialized on a particular block column, all block columns are independent. The same reasoning holds for the updates from the right, replacing block column with block row.

This allows for an algorithm where the number of applied updates differ between each block column/row at any given time. The *dual antidiagonal wavefront* implementation described visually in Figure 3 is an example where different priorities change the behavior and performance of the algorithm. Since all tasks

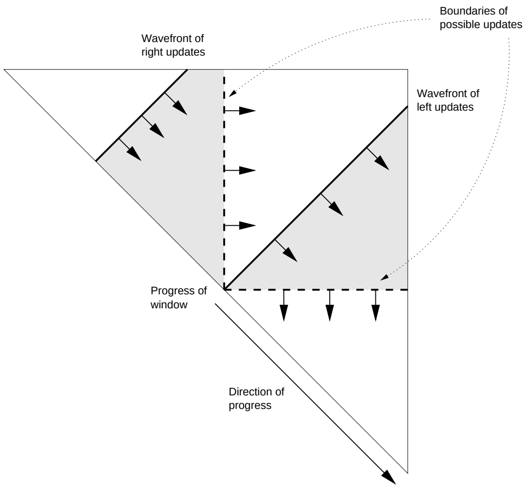

Boundaries of possible updates
Wavefront of right updates
Wavefront of left updates
Progress of window
Direction of progress

Fig. 3. The two antidiagonal wavefronts (the upper one corresponds to updates from the right, the lower one to updates from the left). The grey areas represent the parts of the matrix that have pending updates at this point.

affect a single block in the matrix we use the global upper left coordinates (i, j) of that block as input to the priority calculation. To obtain the dual antidiagonal wavefront behavior we used the following priorities: *computing* Q has priority 0 (highest priority), *an update from the left* has priority $i + j$, and *an update from the right* has priority $i+j+N/2$. Note that $i+j$ is constant along an antidiagonal.

3.4 Some Computational Results

In this section, we present measurements of execution time in the form of parallel efficiency. The Sarek cluster at HPC2N, used for our experiments, consists of 190 nodes with dual AMD Opteron 248 (2.2GHz) processors connected with Myrinet 2000, and 8GB of memory per node. We used MPICH MPI, one MPI process per socket and only one thread per process ($w = 0$) since the processors have a single core. We do not yet have an equivalent sequential implementation of the `Sweep` algorithm but as an estimate of the sequential cost we accumulated the time spent in kernel computations (copy and update) in each process. In Figure 4 and Figure 5, we present results for the straightforward implementation and the dual antidiagonal wavefront implementation, respectively. In these figures, the

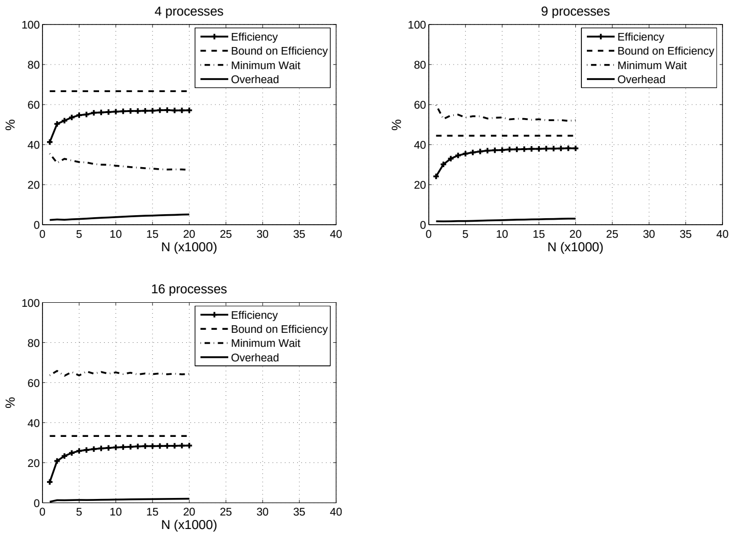


Fig. 4. Efficiency, upper bound on efficiency, overhead, and minimum synchronization overhead (Minimum Wait) for the straightforward implementation of `Sweep` ($n_b = 100$).

following information is displayed.

– *Efficiency.* Serial execution time is estimated by summing the time spent in kernel routines. To be precise, the efficiency is estimated by

$$E_p = \frac{\sum_{i=1}^{P_r P_c} T_{\text{kernel}}^{(i)}}{P_r P_c T_p}.$$

– *Bound on efficiency.* The bound on E_p from Section 3.2,

$$\frac{8}{3(P_r + P_c)},$$

- *Minimum wait.* On process i, the time spent in blocking MPI functions is $T_{\text{wait}}^{(i)}$. The *minimum wait* is

$$\frac{\min_i T_{\text{wait}}^{(i)}}{T_p},$$

which approximates the impact of synchronization overhead.
- *Overhead.* This estimates the overhead due to the dynamic scheduling, such as constructing the DAGs, MPI polling, etc. The overhead curve shows the overhead as a percentage of the processor-time product $(P_r P_c T_p)$.

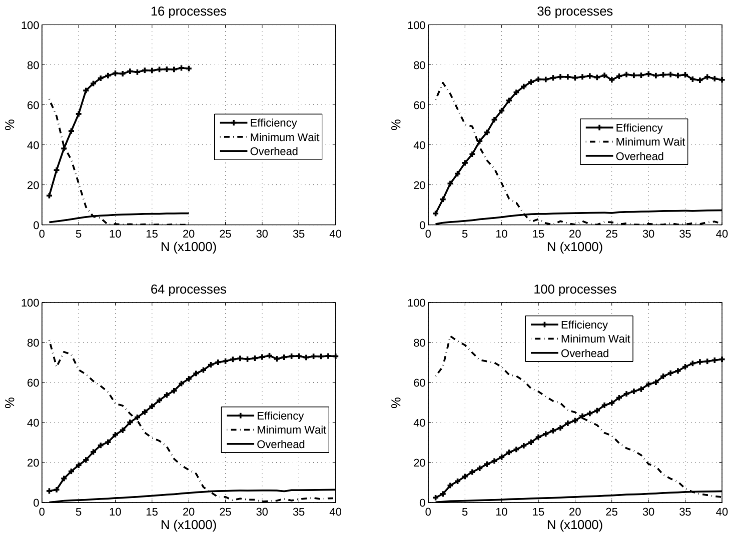


Fig. 5. Efficiency, overhead, and minimum synchronization overhead (Minimum Wait) for the more efficient dual wavefront implementation of `Sweep` $(n_b = 100)$.

In Figure 4 (straightforward implementation), the relative efficiency comes close to the theoretical upper bound. The overhead is a few percent and the minimum wait is almost constant but still very significant. The straightforward implementation is not scalable and the reason is synchronization overhead. In Figure 5 (dual wavefront implementation), the relative efficiency peaks at around 70–80%. A strong correlation between the minimum wait and the efficiency shows that synchronization overhead is virtually eliminated by the dual wavefront algorithm for large enough problems.

4 Conclusions

We have presented a design of a framework for dynamic node-scheduling of blocked matrix computations on hybrid distributed memory machines. The framework uses an efficient runtime data dependence analysis method. Priority-based scheduling has been shown to extract much more of the available parallelism than standard FIFO scheduling. For example, comparing the timings in Figures 4 and 5 for 16 processes and $N = 20000$ (the largest problem solved on 16 processes), we get a speedup of 2.75 in favor for the wavefront implementation of `Sweep`. Going from 16 to 64 processes in Figure 5, we get another speedup of 3.32. We also see that $N = 20000$ is not large enough to reach practical peak on 64 processes.

References

1. B. Adlerborn, B. Kågström, and D. Kressner. Parallel Variants of the Multishift QZ Algorithm with Advanced Deflation Techniques . In B. Kågström et al., editors, *Applied Parallel Computing: State of the Art in Scientific Computing, PARA 2006*, Lecture Notes in Computer Science, LNCS 4699, pages 117–126. Springer, 2007.
2. P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006. ACM.
3. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.
4. K. Braman, R. Byers, and R. Mathias. The Multishift QR Algorithm. Part I: Maintaining Well-Focused Shifts and Level 3 Performance. *SIAM J. Matrix Anal. Applics.*, 23:929–947, 2001.
5. R. Brightwell and K. D. Underwood. An analysis of the impact of MPI overlap and independent progress. In *ICS '04: Proc. of the 18th annual international conference on Supercomputing*, pages 298–305, New York, NY, USA, 2004. ACM Press.
6. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. Technical Report UT-CS-07-600, University of Tennessee at Knoxville, 2007. Also as LAPACK Working Note 191.
7. E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. Super-Matrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 9-11 2007.
8. G. H. Golub and C. F. Van Loan. *Matrix Computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.
9. R. Granat, D. Kressner, and B. Kågström. Parallel Eigenvalue Reordering in Real Schur Forms. *Concurrency and Computation: Practice and Experience*, accepted 2008. Also as LAPACK Working Note 192.
10. B. Kågström and D. Kressner. Multishift Variants of the QZ Algorithm with Aggressive Early Deflation. *SIAM J. Matrix Anal. Applics.*, 29:199–227, 2006.
11. MPI: A Message Passing Interface Standard. `http://www.mpi-forum.org/`, 1995.