

QR Factorization with Morton-Ordered Quadtree Matrices for Memory Re-use and Parallelism*

Jeremy D. Frens[†]
Calvin College
Grand Rapids, MI 49546-4388
jdfrens@calvin.edu

David S. Wise[‡]
Indiana University
Bloomington, IN 47405-7104
dswise@cs.indiana.edu

ABSTRACT

Quadtree matrices using Morton-order storage provide natural blocking on every level of a memory hierarchy. Writing the natural recursive algorithms to take advantage of this blocking results in code that honors the memory hierarchy without the need for transforming the code. Furthermore, the divide-and-conquer algorithm breaks problems down into independent computations. These independent computations can be dispatched in parallel for straightforward parallel processing.

Proof-of-concept is given by an algorithm for *QR* factorization based on Givens rotations for quadtree matrices in Morton-order storage. The algorithms deliver positive results, competing with and even beating the LAPACK equivalent.

Categories and subject descriptors:

G.1.3 [Numerical Analysis]: Numerical Linear Algebra—linear systems; E.1 [Data Structures]: Arrays; D.4.2 [Operating Systems]: Storage Management—segmentation, swapping, virtual memory; B.3.2 [Memory Structures]: Design Styles—primary memory; D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming; F.2.1 [Analysis of Algorithms and Problem Complexity]: Numerical Algorithms and Problems—computations on matrices; G.4 [Mathematical Software]: Algorithm analysis.

*Supported, in part, by the National Science Foundation under a grant numbered CDA93-03189.

[†]Supported, in part, by the U.S. Department of Education under a grant numbered P200A50237.

[‡]Supported, in part, by the National Science Foundation under grants numbered CCR-0073491, ACI-0219884, EIA-0202048.

General terms: Performance; Algorithms.

Additional key words and phrases: storage management, indexing, quadtrees, swapping, cache misses, paging.

1. INTRODUCTION

Earlier work [13, 8, 12] has explored matrix-matrix multiplication using quadtree matrices stored in Morton order. These results suggest that a Morton-order indexing [25, p. 776] of a quadtree matrix effectively blocks the matrix scalars to fall into the transfer blocks of the various levels of the memory hierarchy. Divide-and-conquer algorithms, which honor the re-use of the quadrants of a quadtree matrix, also honor the memory hierarchy of a computer without knowledge about its existence, let alone the particulars of each level of the hierarchy. In general (not restricted to divide-and-conquer algorithms), this phenomenon is called **cache-oblivious** [14]: the algorithm needs neither tuning nor specification of the memory system on which it is running.

In contrast, tiling iterative algorithms for row-major matrices [20, Section 20.4.3] requires advanced knowledge of the sizes of the levels of the memory hierarchy, although there is software that can account for their impact from experiments [4, 23].

Furthermore, divide-and-conquer algorithms have been advocated for parallelism [27]. The independent computations in such an algorithm can be executed in parallel without inter-process communication [9, Section 1.3.1][3].

The time for both accessing the memory and communicating across processes must be reduced for efficient high-performance computing [1, 10, 17].

While the results are encouraging, matrix multiplication is relatively simple and straightforward; other problems are not as well patterned which could significantly affect the performance of the quadtree matrix with these algorithms. One such problem is the *QR* factorization of a matrix [15, Section 5.2]. It is a common problem with algorithms implemented for row- and column-major matrices in the LAPACK library [2]. This paper tackles *QR* factorization for quadtree matrices stored with Morton-order indexing [12].

This paper is organized in seven sections, the first being this introductory section. The second section defines some terms and concepts for quadtree matrices. The third section describes *QR* factorization. The fourth section describes the quadtree matrix functions used for *QR* factorization, including the parallel dispatch of the functions. The fifth section describes some of the issues involved in coding up

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'03, June 11–13, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-588-2/03/0006 ...\$5.00.

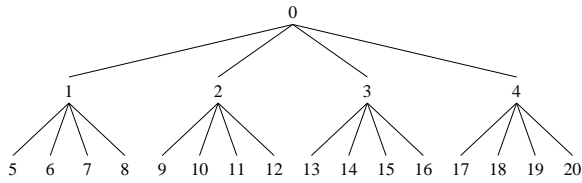


Figure 1: Level-order indexing of quaternary tree

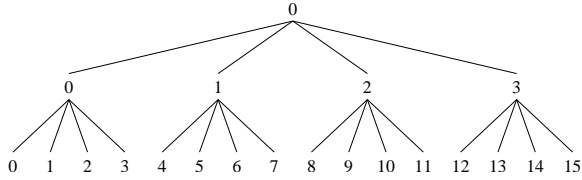


Figure 2: Morton-order indexing of quaternary tree

the functions defined in the fourth section. The sixth section gives uniprocessor and multiprocessor results, and the seventh concludes.

2. QUADTREE MATRICES

DEFINITION 2.1. A **quadtrees matrix** is either zero, a non-zero scalar, or a quadruple of sub-matrices (northwest, northeast, southwest, southeast) of equal size where at least one sub-matrix is non-zero. [24, p. 33]

The four quadrants of a quadtree matrix are selected with the down-arrow operator \downarrow (e.g., $M \downarrow \text{nw}$ for the northwest quadrant of M , $M \downarrow \text{se} \downarrow \text{nw}$ for the northwest of the southeast of M).

2.1 Indexing a Quadtree

By numbering the nodes of the quadtree, the elements of the quadtree can be mapped into an array. Two indexings are of primary interest:

DEFINITION 2.2. The **level ordering of a quaternary tree** is an indexing of the nodes of the quaternary tree such that the root has index 0 and for a node in the tree with index i , its children are indexed $4i + 1$, $4i + 2$, $4i + 3$, and $4i + 4$.¹ [25, p. 776]

DEFINITION 2.3. The **Morton-order indexing of a quaternary tree** is an indexing of the nodes of the quaternary tree such that the root has index 0 and for a node in the tree with index i , its children are indexed $4i + 0$, $4i + 1$, $4i + 2$, and $4i + 3$. [25, p. 776]

Figures 1 and 2 illustrate these two indexings.

The main difference between these indexings is that level order gives a unique index to every non-terminal node while Morton order indexes each level, starting with index zero. The level-order indices on one level of the tree differ from

¹Traditionally, the root of a binary tree has index 1 in level-order indexing [18, p. 401]; however, in trees with a higher degree (such as a quadtree), this leaves gaps in the indexing from one level to the next. A 0-indexed root works well for all trees without indexing gaps.

the corresponding Morton-order indices by only a constant $\sum_{i=0}^{l-1} 4^i = (4^l - 1)/3$ where l is the number of the level (zero-based) [25]. This conversion makes these indexings easily interchangeable.

2.2 Padding and Decorations

The recursive, two-dimensional bifurcation of a quadtree matrix suggests that the order of the matrix should be a power of two. For matrices where this does not apply, the matrices can be padded to the south and east, usually with zeros.

To describe the padding in a matrix, several definitions are useful:

DEFINITION 2.4. A **stripe** is a set of adjacent rows in a matrix. A **colonnade** is a set of adjacent columns. [24, p. 33]

DEFINITION 2.5. **Majority padding** is padding that extends into the west colonnade or the north stripe. **Minority padding** is padding that is found only in the east colonnade or the south stripe. A **perfect quadtree matrix** is a quadtree matrix that has no padding. [12, p. 21]

The space for the zero elements of a quadtree matrix stored in Morton order must be allocated in the computer's address space. However, a second array using level-order indexing (corresponding to the non-terminal nodes of the quadtree) can be used to store **decorations**. These decorations can be used by the quadtree-matrix algorithms to avoid the padding quadrants.

Four decorations are useful:

ZERO	The matrix is the zero matrix.
IDENTITY	The matrix is the identity matrix.
PERFECT	The matrix is a perfect quadtree matrix (i.e., without internal zeros or identities, or padding, cf. Definition 2.5).
UNKNOWN	The contents of the matrix is unknown at this level.

With so few decorations, a single decoration takes up less than a byte of data; decorations can be stored very compactly.

Algorithms can honor the algebra of ZERO and IDENTITY decorations as special cases, avoiding the traversal of the lower levels of that matrix, preventing the scalars from those quadrants from ever being accessed and brought into higher levels of a memory hierarchy (e.g., cache). They will consume space only in the lower levels of a memory hierarchy.

The PERFECT decoration allows algorithms to stop testing the decorations since the matrix does not have any padding or other special quadrants. The UNKNOWN decoration indicates that testing the decorations must continue within the matrix; it must consist of a mixture of PERFECT, ZERO, and IDENTITY matrices.

While these decorations, the ZERO decoration in particular, can assist with processing sparse matrices, this paper focuses on dense matrices.

2.3 Row-Major Matrices

Throughout this paper, the term "row-major" should be read as "row- and column-major" unless otherwise noted.

A row-major matrix stores an entire row of a matrix in consecutive memory locations. Poor spatial locality results from traversing a row-major matrix by columns instead of

rows [7, Section 2], yet column traversals are often necessary in the iterative algorithms written for row-major matrices. Better reuse of local memory (within a transfer block) is achieved by dealing with the matrix in terms of blocks, but this must be provided by the algorithm since the data is not naturally blocked. One standard way to block an algorithm is to tile the loops [20, Section 20.4.3].

3. QR FACTORIZATION

The QR factorization of an $n \times n$ matrix A produces an $n \times n$ orthogonal matrix Q (i.e., $QQ^T = I = Q^TQ$) and an $n \times n$ upper-triangular matrix R such that $A = QR$ [15, Section 5.2].

QR factorization applies a series of orthogonal transformations Q_1, Q_2, \dots, Q_m to $A = A_0, A_1, A_2, \dots, A_m$, updating each successive $A_i = Q_i^T A_{i-1}$ until $A_m = R$, an upper-triangular matrix, is produced. The matrix Q is formed by multiplying the individual orthogonal transformations together. Different algorithms for QR factorization arise since different orthogonal transformations can be used for Q_i and the type of transformation used determines m .

3.1 Householder QR Factorization

A **Householder reflection** is an orthogonal transformation that is applied to one column of a matrix to zero out selected components in a column [15, Section 5.1.2]. In Householder QR factorization, each Q_i is a Householder reflection to zero out the portion of column i below the matrix diagonal of A_i [15, Section 5.2.1].

Most QR factorizations for row-major matrices are done using Householder reflections, and the reflections themselves can be stored individually in the eliminated portion of A . Q is not formed explicitly, although it can be computed quite easily.

3.2 Givens QR Factorization

A **Givens rotation** eliminates just one selected element from the matrix using another element [15, Section 5.1.8]. To eliminate $b \neq 0$ using a , the Givens rotation is formed from the cosine c and sine s of a right triangle:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where

$$c = \frac{a}{\sqrt{a^2 + b^2}} \quad \text{and} \quad s = \frac{-b}{\sqrt{a^2 + b^2}}.$$

If $b = 0$, then the identity matrix is used (i.e., $c = 1$ and $s = 0$) since b is already eliminated. This avoids all undefined operations (like division by zero).

An iterative QR -factorization using Givens rotations is presented in Figure 3 [15, Algorithm 5.2.2].

3.3 Block Householder QR Factorization Algorithms

A block representation for Householder transformations can be used in a block algorithm for QR factorization [15, Sections 5.17 and 5.2.2], resulting in a more memory-efficient algorithm and only slightly increasing the number of floating-point operations. Columns are taken in groups, and each group is factored using the column-based algorithm. The updates to the rest of A are saved and applied to each block

```

for (int j = 0; j < n; j++) {
  for (int i = n; i >= j+1; i--) {
    Scalar c, s;
    givens (c, s, R[i-1,j], R[i,j]);
    for (int k = j; k < n; k++) {
      Scalar top = c * R[i-1,k] - s * R[i,k];
      Scalar bot = s * R[i-1,k] + c * R[i,k];
      R[i-1,k] = top;
      R[i,k] = bot;
    }
    for (int k = 0; k < n; k++) {
      Scalar top = c * Q[i-1,k] - s * Q[i,k];
      Scalar bot = s * Q[i-1,k] + c * Q[i,k];
      Q[i-1,k] = top;
      Q[i,k] = bot;
    }
  }
}

```

Figure 3: Iterative QR factorization using Givens rotations

of columns in turn just before they are factored. These updates use matrix-matrix multiplications instead of less efficient matrix-vector multiplications. The `dgeqrf()` function in LAPACK for QR factorization uses this approach.

The block size in that algorithm (i.e., the number of columns in a group) must be tuned for particular machines just as block sizes in tiling must also be tuned. Either programmers or optimizing compilers must know or obtain this information, although it can be collected automatically (e.g., by PHiPAC [4] or ATLAS [23]).

Elmroth and Gustavson [11] take a recursive approach to QR factorization on row-major matrices using Householder transformations with the same blocking effect, saving a block of column updates so that matrix-matrix multiplication is used for the updating. Their recursive algorithm incurs significantly more flops than a purely iterative algorithm, so their ultimate algorithm is actually an iterative/recursive hybrid. The top-level algorithm is iterative over blocks of the matrix, calling their recursive algorithm to factor each block. They tune the hybrid algorithm on the number of columns factored by the recursive algorithm to balance the benefit of the recursive algorithm with drawback of the increased number of flops.

While Elmroth and Gustavson's recursive algorithm is similar to the function f presented in the next section, their approach is still in terms of the columns of the matrix and column-major storage.

3.4 Elmroth and Gustavson

The hybrid algorithm for QR factorization from Elmroth and Gustavson outperforms `dgeqrf()` on their IBM machines (15–20% better for larger orders); their best performance on an IBM POWER2 node is 90% of the maximum flops rate for the machine. The speed-ups for their parallel version are close to the ideals, improving steadily as the order of the matrix increases.

Their algorithm does have a tuning parameter: the number of columns in a block sent to the recursive algorithm from the iterative algorithm. Using only their recursive algorithm significantly increases the number of floating-point

operations done overall; the hybrid allows them to keep the number of flops lower while gaining the benefits of the recursive algorithm. They do not do any tuning for the memory hierarchy, so they too are cache-oblivious.

4. QUADTREE QR FACTORIZATION

In the QR factorization functions, Z represents the zero matrix, and I represents the identity matrix. In the program, these are implemented as decorations (cf. Section 2.2).

The QR factorization for quadtree matrices consists of two mutually recursive functions, f and e . The function f factors a matrix with a QR factorization; the function e eliminates one triangular matrix using another triangular matrix.

4.1 Quadtree QR Factorize

The factorization function is $f: A \mapsto \langle Q, R \rangle$ where A , Q , and R are $n \times n$ matrices; Q is orthogonal; R is upper-triangular; and $A = QR$. This is the top-level function to factor a matrix. The function is presented in Figure 4. The quadrants in the west are recursively factored, and the southwest result is eliminated using e . Then the southeast is factored using f . After each call to f and e , A is updated appropriately.

4.2 Quadtree QR Eliminate

The elimination function $e: \langle N, S \rangle \mapsto \langle Q, \tilde{N} \rangle$ assists f in factoring matrices. It eliminates an upper-triangular matrix S using another upper-triangular matrix N ; N is updated to \tilde{N} which is also upper-triangular. (N and S get their names from “north” and “south”, respectively, indicating the relative positions of the two blocks in the matrix being factored; e.g., Step 3 of f .) The N , S , and \tilde{N} are all $n \times n$ matrices while Q has order $2n \times 2n$. Q is orthogonal, and \tilde{N} is upper-triangular. The computational postcondition for e is

$$\begin{bmatrix} N \\ S \end{bmatrix} = Q \begin{bmatrix} \tilde{N} \\ Z \end{bmatrix}. \quad (1)$$

The algorithm is presented in Figure 5. Similar to f on A , elimination works west to east through N and S .

The elimination function presents some specialized multiplications that require fewer than $2n^3$ flops. Since N and S are both triangular, Q is returned from e with triangular patterns in it. The ZERO and IDENTITY decorations direct the multiplication algorithms to take advantage of these patterns. A programmer might also tailor special multiplication routines to these patterns, and thereby avoid the need for testing the decorations.

4.3 Memory Efficiency Without Tuning

The benefit of Morton-order indexing and quadtree matrices is that the matrix is blocked at every level of the quadtree matrix. Morton order assigns indices based on blocks, and so each quadrant is indexed with its own unique sequence of consecutive indices. Then, for any memory hierarchy, there will be some level of the quadtree matrix that fits nicely into a transfer block of that memory.

Interestingly, the quadtree-matrix algorithm does *naturally* what has to be *added* to the iterative Householder algorithm (i.e., save updates to A , see Section 3.3).

Furthermore, the functions are cache-oblivious. The quadtree-matrix programmer does not have to tune the blocking

Function $f: A \mapsto \langle Q, R \rangle$ where
 A , Q , and R are $n \times n$ matrices,
 Q is orthogonal, R is upper-triangular, and $A = QR$.

Base case when $n = 1$,
Name: $Q = I$,
 $R = A$.

Recursion when $n > 1$,

Step 1: $\langle Q_1, R_1 \rangle = f(A \downarrow_{nw})$.
Step 2: $\langle Q_2, R_2 \rangle = f(A \downarrow_{sw})$.

Name: $Q_{1\&2} = \begin{bmatrix} Q_1 & Z \\ Z & Q_2 \end{bmatrix}$.

Step 3: $\langle Q_3, R_3 \rangle = e(R_1, R_2)$.
Step 4: $Q_4 = Q_{1\&2} Q_3$.

Step 5: $\begin{bmatrix} U_n \\ U_s \end{bmatrix} = Q_4^T \begin{bmatrix} A \downarrow_{ne} \\ A \downarrow_{se} \end{bmatrix}$.

Step 6: $\langle Q_6, R_6 \rangle = f(U_s)$.

Step 7: $Q = Q_4 \begin{bmatrix} I & Z \\ Z & Q_6 \end{bmatrix}$.

Name: $R = \begin{bmatrix} R_3 & U_n \\ Z & R_6 \end{bmatrix}$.

Figure 4: QR factorization function, f

Function $e: \langle N, S \rangle \mapsto \langle Q, \tilde{N} \rangle$ where
 N , S , and \tilde{N} are $n \times n$ matrices, N and S are upper-triangular,
 \tilde{N} is upper-triangular, Q is an $2n \times 2n$ orthogonal matrix, and
Equation 1 holds.

Base cases when $S = Z$, $\left\{ \begin{array}{l} \text{when } n = 1, \\ Q = I. \\ \tilde{N} = N. \end{array} \right. \left[\begin{array}{l} Q = \text{givens}(N, S). \\ \begin{bmatrix} \tilde{N} \\ Z \end{bmatrix} = Q^T \begin{bmatrix} N \\ S \end{bmatrix}. \end{array} \right.$

Recursion when $n > 1$,

Step 1: $\langle Q_1, \tilde{N}_1 \rangle = e(N \downarrow_{nw}, S \downarrow_{nw})$.
Step 2: $\langle Q_2, \tilde{N}_2 \rangle = e(N \downarrow_{se}, S \downarrow_{se})$.

Name: $Q_{1\&2} = \begin{bmatrix} Q_1 \downarrow_{nw} & Z & Q_1 \downarrow_{ne} & Z \\ Z & Q_2 \downarrow_{nw} & Z & Q_2 \downarrow_{ne} \\ Q_1 \downarrow_{sw} & Z & Q_1 \downarrow_{se} & Z \\ Z & Q_2 \downarrow_{sw} & Z & Q_2 \downarrow_{se} \end{bmatrix}$.

Step 3: $\begin{bmatrix} U_n \\ U_s \end{bmatrix} = Q_1^T \begin{bmatrix} N \downarrow_{ne} \\ S \downarrow_{ne} \end{bmatrix}$.

Step 4: $\langle Q_4, R_4 \rangle = f(U_s)$.

Step 5: $Q_5 = Q_{1\&2} \begin{bmatrix} I & Z & Z & Z \\ Z & I & Z & Z \\ Z & Z & Q_4 & Z \\ Z & Z & Z & I \end{bmatrix}$.

Step 6: $\langle Q_6, \tilde{N}_6 \rangle = e(\tilde{N}_2, R_4)$.

Step 7: $Q = Q_5 \begin{bmatrix} I & Z & Z & Z \\ Z & Q_6 \downarrow_{nw} & Q_6 \downarrow_{ne} & Z \\ Z & Q_6 \downarrow_{sw} & Q_6 \downarrow_{se} & Z \\ Z & Z & Z & I \end{bmatrix}$.

Name: $\tilde{N} = \begin{bmatrix} \tilde{N}_1 & U_n \\ Z & \tilde{N}_6 \end{bmatrix}$.

Figure 5: QR elimination function, e

of the quadtree-matrix algorithm for the memory hierarchy. The structure itself is blocked, and the blocked nature of quadtree-matrix algorithms leads to blocked algorithms without tuning. Programming for the quadrants of the quadtree matrix is programming for the memory hierarchy.

4.4 Accumulating Q

The iterative algorithms for QR factorizations typically leave Q unfactored; each Q_i is saved in the eliminated portions of A . The quadtree matrix algorithms compute Q explicitly to update A , and so they incur more work (see Section 6.2). (Most applications do not need Q , and if it is, Q can always be formed, regardless of what algorithm was used.)

4.5 Errors

No formal analysis of the accuracy of the quadtree-matrix functions has been done, but the orthogonality of the Givens rotations provides the algorithms with great stability. The error analysis of QR factorization using Givens rotations without pivoting is very favorable [16, Section 1.14.2, Section 18.5].

Explicit pivoting is used with QR factorizations on singular matrices [15, Section 5.4.1] for reasons beyond the scope of this work. Pivoting is unnecessary on nonsingular matrices. Neither the standard algorithm in LAPACK nor the algorithm from Elmroth and Gustavson does any pivoting.

The quadtree-matrix functions only make *explicit* pivoting unnecessary. If $N = Z$ in e , then the Givens rotation generated naturally by e will be a permutation matrix that pivots the matrices so that $\tilde{N} = S$.

4.6 Parallelism

The two functions for QR factorization are evaluated according to the padding in their inputs (similar to the parallelization of matrix multiplication for quadtree matrices [13]). To properly balance the computations in a parallel dispatch, the amount of padding must be considered. First, the padding determines the amount of computation done; second, the padding determines the pattern of memory accesses.

The cases of an algorithm are first determined by the shapes of the matrices: square, stripe, and colonnade (see Definition 2.4). Each case is further analyzed based on the amount of padding: perfect, majority, and minority (see Definition 2.5).

In the parallel dispatch charts [12] (Figures 7 and 9), the steps of the algorithms are listed vertically. Horizontal lines indicate a synchronization. Steps listed side-by-side can be done in parallel. Parallel matrix multiplication routines were called as often as possible. The dispatches in Figures 7 and 9 do a parallel dispatch of the multiplications; for the other dispatch cases, the functions for QR factorization let the multiplication routines handle all of the parallelism.

The function f decomposes into three cases as sketched in Figure 6.

0. **Perfect Square.** When A is perfect, the northwest and southwest quadrants can be factored in parallel (Steps 1 and 2). The multiplications of Steps 4, 5 and 7 are well balanced within each step, so the multiplications of those steps are immediately dispatched in parallel. This dispatch is given in Figure 7.

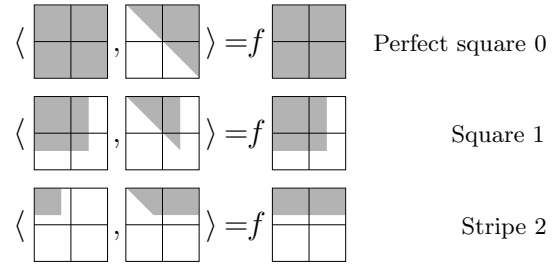


Figure 6: Parallel patterns of f

Step 1: perfect square f	Step 2: perfect square f
Step 3: perfect square e	
Step 4: parallel multiplication	
Step 5: parallel multiplication	
Step 6: perfect square f	
Step 7: parallel multiplication	

Figure 7: Parallel QR factorization: perfect square dispatch

1. **Square.** This is the general case, and it is where top-level dispatching starts. There are no immediate opportunities for parallel dispatch in this case; all parallelism is deferred to the next level in the function-call tree.

MAJORITY-PADDING DISPATCH: This case reduces to one instance of itself on the northwest quadrant of A .

MINORITY-PADDING DISPATCH: All steps are done sequentially.

2. **Stripe.** As with the square case, there are no immediate opportunities for parallel dispatch.

MAJORITY-PADDING DISPATCH: This case reduces to one instance of itself on the northwest quadrant of A plus an update to $A \downarrow ne$ (Step 5).

MINORITY-PADDING DISPATCH: The dispatch for this case is identical to the minority-padding dispatch for the square case.

The function e breaks down into only two cases as depicted in Figure 8. The case and size of the padding is determined by the padding in S since N is always upper-triangular and dense.

0. **Perfect Square.** The eliminations of Steps 1 and 2 can be done in parallel with each other. Parallel dispatch is possible for the multiplications of Steps 3, 5, and 7. This pattern of dispatch is given in Figure 9.

1. **Stripe.** This case is strange because the majority case is non-trivial (unlike most other majority-padding cases) and because there is some parallelism in the minority case.

MAJORITY-PADDING DISPATCH: as noted, this case does not merely reduce to an instance of itself. All if the

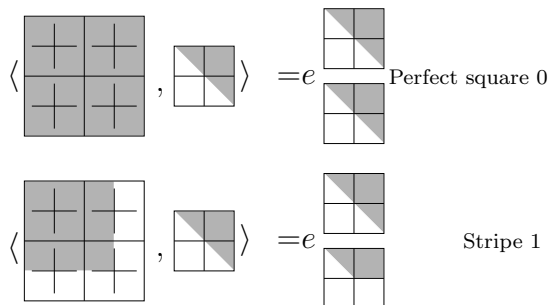


Figure 8: Parallel patterns of e

Step 1: perfect square e	Step 2: perfect square e
Step 3: parallel multiplication	
Step 4: perfect square f	
Step 5: parallel multiplication	
Step 6: perfect square e	
Step 7: parallel multiplication	

Figure 9: Parallel QR elimination: perfect square dispatch

steps of the function must be done sequentially (although some are trivial).

MINORITY-PADDING DISPATCH: all steps must be done sequentially although the multiplications steps can be dispatched immediately in parallel.

5. THE CODE

Matrix-matrix multiplication is implemented using the algorithms by Frens and Wise [13, 26]. The functions f and e are implemented in C as is the matrix multiplication algorithm. Drivers were written in C++.

5.1 Unfolding and Re-rolling the Base Case

Iterative loops are routinely unrolled [20, Section 17.4.2] to take advantage of superscalar architectures and software pipelining. Optimizing compilers do this automatically for programmers.

However, these same optimizing compilers have a poorer understanding of recursion, and they do not unfold the base cases [6, 21] of a recursive function. While this was known in earlier work [13], finding the right unfolding for the compiler to optimize into the best superscalar code has been the real key. Matrix-matrix multiplication is unfolded to an 8×8 base case by hand. To avoid filling the instruction cache, to tolerate cache mapping, and to take advantage of the optimizing compiler’s knowledge of loops, the unfolded base cases are re-rolled into loops that operate on the Morton-order indices [26]. The same technique is used to code the base cases of the QR -factorization functions f and e using the algorithm in Figure 3 with Morton-order indices.

It is assumed that the optimizing compilers unroll the LAPACK and BLAS algorithms. (Everything is compiled with the maximum optimizations.) Unfolding the base cases

of the quadtree algorithms only establishes a level playing field with these other libraries.

5.2 Decorations

The code for functions f and e make use of the decorations in the quadtree matrices. Neither one is coded to recognize the IDENTITY decoration since A does not start with identity quadrants and none of the A_i updates have identity matrices introduced into them. However, both functions place IDENTITY decorations into the Q s that they generate, so the matrix multiplication routines handle identity quadrants.

Two versions of each function are used to deal with the decorations. One recognizes and reacts appropriately to the decorations; the other ignores the decorations. The second version is called from the first when A in f or S in e is a perfect quadtree matrix. (Incidentally, based on zero padding, N must always be a “perfect” triangular matrix; if it were not, then S would be all zero and there would have been no need to call e on it.) The test-free version avoids the overhead of testing the decorations. Arguably, the programmer should write just one version, and the two versions could be generated automatically.

Due to the unfolded 8×8 base case, any 8×8 quadrant with even one non-zero value in it is considered to be a perfect quadtree matrix. The zeros in this matrix will then be used in the base-case computation, but ultimately this is faster than testing for down to the scalar level.

5.3 Multiprocessing Base Cases

Since a parallel dispatch has an overhead, there is a point where a parallel dispatch on a quadrant takes more time than to use the uniprocessor algorithm. General tests indicate that a 32×32 quadrant (or smaller) is best done uniprocessor.

Also, if the padding in a matrix is small enough, it is worthwhile in the parallel dispatch to ignore the padding. Parallel dispatch can then switch over the perfect square case earlier for more parallelism higher in the function-call tree. Again, general tests indicate that padding 32 elements wide or 32 elements tall (or smaller) is best ignored.

6. RESULTS

Three machines were used to obtain timing results. The quadtree matrix algorithm was run against `dgeqrf()`, the QR factorization algorithm from LAPACK using Householder transformations.

6.1 The Machines

Three machines were used to run timing experiments: an SGI Power Challenge, an SGI Octane, and a Sun Enterprise 450 Model 4400, all described in Table 1. All of the tests were run on these machines in shared mode, although care was taken to run the tests when the load on the machines was minimal.

The MIPSpro compiler was used on the SGI machines. The manufacturer provided its own BLAS libraries; LAPACK was compiled locally. On the Sun machine, the Sun Workshop Compiler 5.0 was used with the Sun Performance Library 2.0, which supplies precompiled BLAS and LAPACK libraries. The LAPACK `dgeqrf()` was not tuned on any of the machines. To avoid most problems with stride [15, Section 1.4.4], every array for row-major matrices is allocated with an odd stride.

Property	Power Challenge	Octane	Enterprise 450
Number of processors	10	1	4
Type of processors	R8000	R10000	Ultra Sparc II
Clock speed	75 MHz	195 MHz	400 MHz
Virtual memory	2 GB	39 GB	5.4 GB
RAM (shared)	2 GB	128 MB	2 GB
Secondary cache	4 MB	1 MB	4 MB
Instruction cache	16 KB	32 KB	16 KB
Data cache	n/a	32 KB	16 KB
Maximum mflop/s	300 mflop/s	390 mflop/s	400 mflop/s
Compiler flags	-Ofast=ip21 or -Ofast=ip30, -64, -mips4, -r8000 or -r10000, -SWP:=0N, -OPT:alias=RESTRICT, -IPA		-fast, -xrestrict

- RAM is shared on multiprocessor machines; processor speed, caches, and mflop/s are per processor.
- The floating point unit of an R8000 is connected only to secondary cache [22, Section 2.2].

Table 1: System parameters

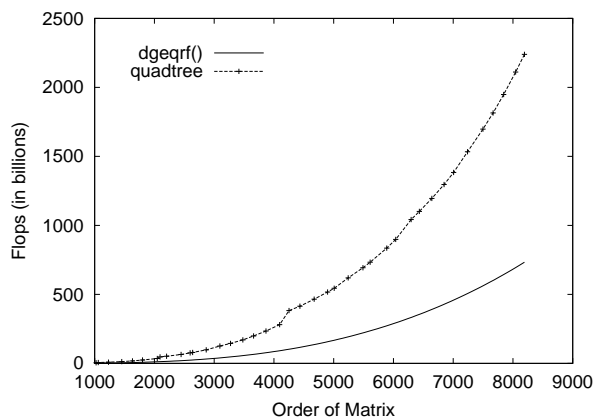


Figure 10: Flop count of QR factorization algorithms

6.2 Flop Counts

The flop count for `dgeqrf()` (approximately $4n^3/3$ [15, Section 5.2.1]) and the flop count for the quadtree algorithm (tallied explicitly in the code) are graphed in Figure 10. (Analytically, the flop count for the quadtree algorithm is $4.17n^3$.) The flop count for the quadtree matrix is over three times higher because it multiplies Q explicitly, while `dgeqrf()` merely stores each Q_i .

6.3 Uniprocessor Results

The running times and mflop/s of `dgeqrf()` and the quadtree algorithm on the three machines are given in Figures 11 through 16. The graphs of mflop/s performance include plots of the maximum flop rates of the various machines.

Since the results reported in earlier work [13, 12], the base case of quadtree matrix-multiplication on the SGIs has been improved significantly; consequently, the quadtree-matrix QR factorization has also improved significantly. The base case has always been unfolded, but more experience and pa-

tient experiments with the SGI MIPS architecture has paid off in better re-rolling of the unfolded code.

It is also very important to inform the compiler that array pointers point to disjoint sections of memory; then the optimizer can optimize array-element computations for superscalar architectures. We have successfully been using the `-OPT:alias=restrict` compiler flag on the SGIs for a long while [13], but only recently have we taken advantage of the `-xrestrict` compiler flag and the `restrict` keyword in C for array declarations. These more than doubled the performance on the Enterprise 450 (contrasted to previous work [12]).

On the Power Challenge (Figures 11 and 12), despite doing considerably *more work* (explicitly accumulating Q), the quadtree algorithm is noticeably *faster* in *raw processor time* than `dgeqrf()`. The flop rate bears this out: the quadtree QR factorization performs at a flop rate almost *four* times better than `dgeqrf()`, and it is nearly as good as the performance of the quadtree matrix algorithm for matrix multiplication.

The `dgeqrf()` function does not even reach a quarter of the mflop/s performance of `dgemm()` (the BLAS3 matrix-matrix algorithm) on the Power Challenge. It may also be slightly unfair comparing the flop rates of the quadtree algorithm for QR factorization to the performance of `dgeqrf()` since the quadtree algorithm benefits from the flop-rate boosting matrix-multiplication algorithm in accumulating Q . But the difference in performance of `dgemm()` and of `dgeqrf()` should not be a factor of four.

On the Power Challenge, `dgeqrf()` were compiled from source code and linked to the manufacturer's BLAS which performs quite well. The performance of `dgeqrf()` would be improved on the Power Challenge if time was spent tuning the algorithm and the compilation; however, without tuning for any block sizes, the quadtree algorithm performs very well.

The Octane (Figures 13 and 14) has very telling results. Just like the Power Challenge, `dgeqrf()` was compiled from source code on the Octane. Despite doing less work, `dgeqrf()`

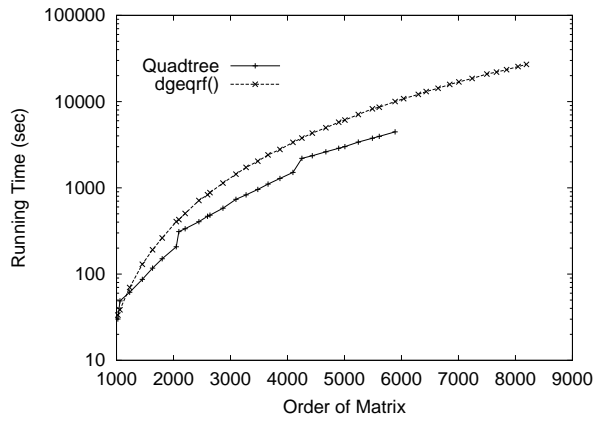


Figure 11: Power Challenge uniprocessor times

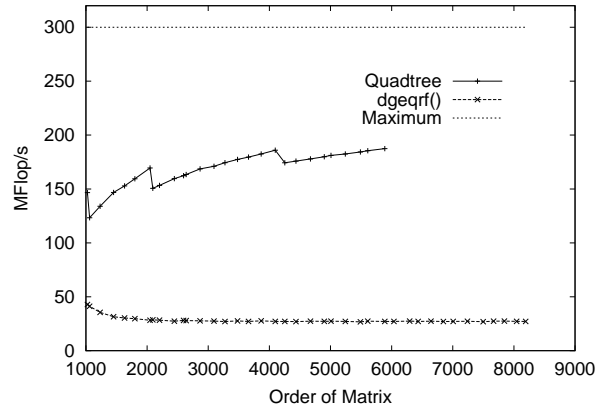


Figure 12: Power Challenge uniprocessor mflop/s

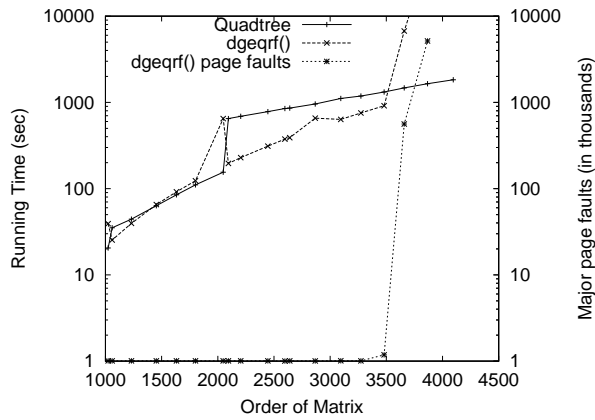


Figure 13: Octane uniprocessor times

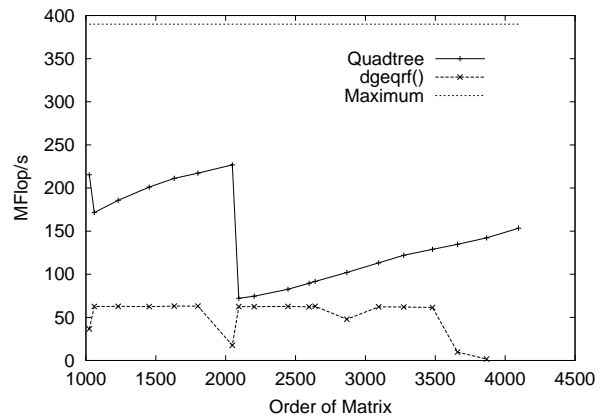


Figure 14: Octane uniprocessor mflop/s

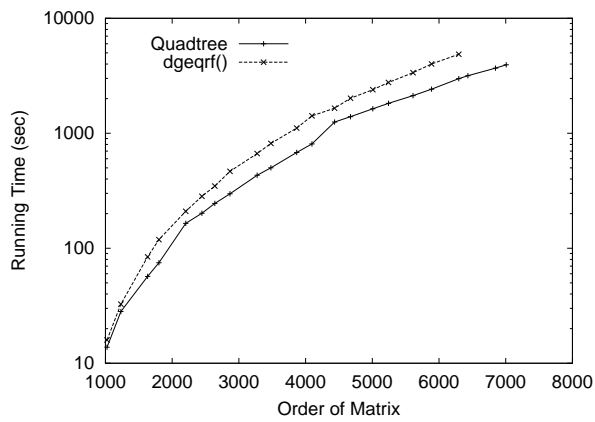


Figure 15: Enterprise 450 uniprocessor times

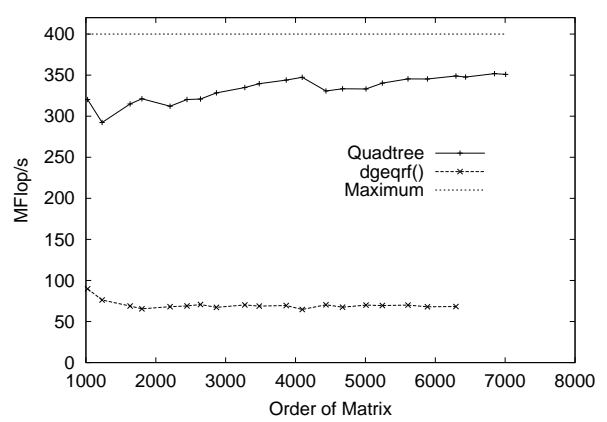


Figure 16: Enterprise 450 uniprocessor mflop/s

takes as much time to compute as does the quadtree matrix algorithm on matrices with orders less than 3500. (The little hiccup at order 2048 for `dgeqrf()` is reproducible and consistent with the results for `dgemm()`; it is most likely a striding issue.)

The most telling result on the Octane comes from matrices with orders above 3500. The page faults for `dgeqrf()` are plotted with the raw processor times in Figure 13, and both increase sharply around order 3500. LAPACK was not tuned for the virtual memory system on this Octane, and so it has trouble with matrices so large. This is perhaps not too surprising since LAPACK was compiled from source; however, this same problem manifests itself in `dgemm()` on the same machine, and the BLAS library *did* come from the manufacturer.

The quadtree algorithm, on the other hand, does much better. The plot of its performance on the Octane appears to be a step function, with steps just after orders that are a power of two. Handling the padding appears to incur a greater cost on the Octane than on the other machines, but the overall performance on the Octane is quite respectable. Most notably, the performance of the quadtree algorithm continues to improve in spite of the fact that another level of the memory hierarchy is being used. (In fact, the quadtree-matrix algorithm triggers the virtual memory system at smaller orders since it explicitly stores Q in another array that consumes extra memory.) The algorithm was *never* tuned for *any* level of the hierarchy, and yet the quadtree matrix handles each level of the memory hierarchy *without extra coding effort and without any knowledge of machine specifics*.

Performance on the Sun (Figures 15 and 16) is also telling, in different ways. It is clear that there are striding problems for `dgeqrf()` on matrices whose orders are a power of two. However, the quadtree algorithm is competitive in the raw processor times despite doing more work. The mflop/s bear this out even better. The relative difference between the quadtree algorithms and `dgeqrf()` is not as close on the Sun Enterprise as it is on the SGIs; however, unlike the SGIs, LAPACK on the Enterprise is part of the Sun Performance Library. One would expect manufacturer-supplied code to be tuned to be as efficient as possible, but these results do not demonstrate this. Yet the quadtree algorithm for QR factorization performs at the same level as matrix-matrix multiplication of quadtree matrices.

Similar tests have been done on the Sun Ultra 5/10 [12]; the results are very similar to Figures 15 and 16.

6.4 Parallel Results

The optimal speed-ups on the graphs for the parallel runs are plotted as horizontal lines.

The Power Challenge does not come with a parallel implementation of `dgeqrf()`. Two solutions were attempted: linking LAPACK to a parallel BLAS and using a Power Challenge version of ScaLAPACK [5]. Both were unsuccessful. Linking to a parallel BLAS did not give enough parallelism; ScaLAPACK is intended for distributed systems which did not work well with the shared-memory on the Power Challenge.

The parallel quadtree algorithm compiled just fine on the Power Challenge. Its speed-ups (Figure 17) are all steadily, asymptotically approaching the ideals. The speed-up is good for two and four processors. The speed-up for eight proces-

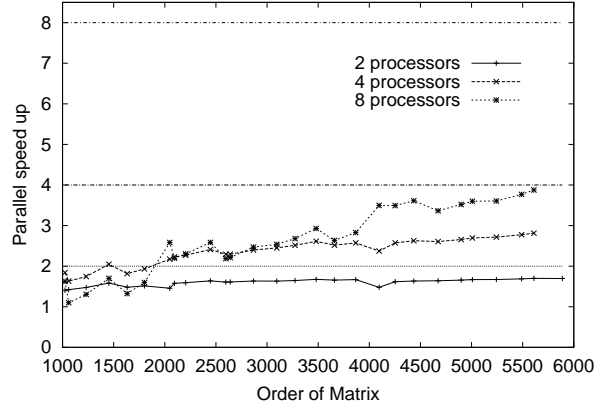


Figure 17: Power Challenge quadtree speed-up

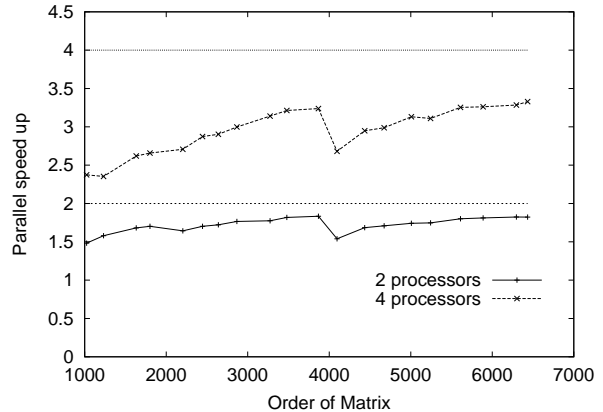


Figure 18: Enterprise 450 quadtree speed-up

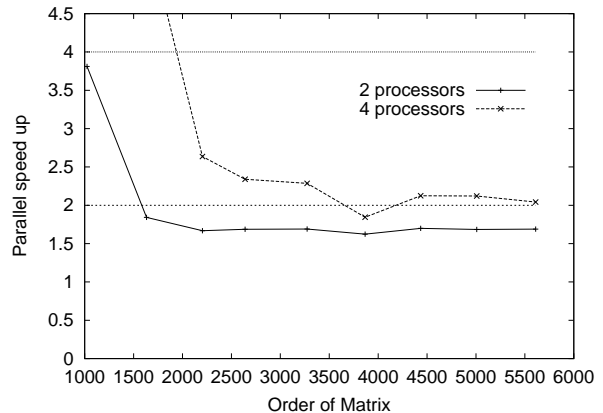


Figure 19: Enterprise 450 `dgeqrf()` speed-up

sors is disappointing, getting just over half the speed-up it should although the increase in the speed-up is clear. Opportunities for parallel dispatch in the quadtree QR factorization are much rarer than they are for, say, matrix multiplication; with eight processors, the opportunities happen much lower in the quadtrees themselves, resulting in less of a payoff in the parallel dispatch.

On the Enterprise 450, the speed-up of `dgeqrf()` was computed using wall-clock time. The Sun Performance Library on the Enterprise 450 uses threads to implement parallelism. The processor timer (from `getrusage()`) measures the total time spent by *all* threads, making the results useless for speed-up calculations. Instead, a wall-clock timer was used to time the uniprocessor and parallel tests, and these wall-clock times were used to calculate *only* the speed-ups on the Enterprise 450 for `dgeqrf()`.

The uniprocessor results on the Enterprise 450 (Figures 15 and 16) call the parallel speed-ups of `dgeqrf()` into question in at least two ways. First, the processor times and flop rate indicate that `dgeqrf()` does not perform as it should. Second, the wall-clock times of some uniprocessor runs were extraordinarily large. Grossly distorted uniprocessor times will make the speed-ups of even very poor parallel runs appear to be good. So the speed-ups for `dgeqrf()` are highly suspect. As far as they can be trusted, the speed-up for two processors seems to be fairly good.

In contrast, the speed-up of the quadtree algorithm (computed using `getrusage()`) increases as the order increases. The performance on two processors starts out close to the ideal, and steadily improves with relatively minor steps at power-of-two orders. The performance of four processors is fairly good and also improves as the order increases.

7. CONCLUSION

The quadtree matrix is automatically tuned for multiple levels of a memory hierarchy; the algorithms written for the quadtree matrix are cache-oblivious by definition. Their performance is consistent even when accessing a new level of the hierarchy as seen quite tellingly on the Octane in Figure 13. The natural blocking of Morton-order indexing and the divide-and-conquer algorithms for quadtree matrices honor the memory hierarchy without tuning or special knowledge.

One of the important considerations in high-performance computing is the innermost computation—the computation that takes advantage of a superscalar architecture. Each tweaking of a base case for matrix multiplication has resulted in significant improvements in performance. More tuning could be done for improved performance of the base cases of both QR factorization functions.

The tuning of the base case does not invalidate the claim that these quadtree-matrix algorithms are cache oblivious. Unfolding a base case (or unrolling an inner loop) depends on the particulars of the *processor*, the *cache mapping*, and perhaps the size of the *instruction cache*; it is *not* a memory-hierarchy issue. An algorithm that did not need to be unfolded or unrolled with explicit tuning could be called *processor oblivious* or *superscalar oblivious*.

As for the parallelism, the divide-and-conquer functions f and e result in relatively easy parallelism that performs quite well (especially on the Enterprise 450, Figure 18).

These results suggest future work for quadtree matrices. Optimizing compilers are needed to unfold and re-roll the

base cases of quadtree matrix algorithms. Compilers are needed to automatically generate the parallel versions of quadtree algorithms from stylish recursive codes. The parallel dispatch of the functions for QR factorization must be generalized to rectangular matrices. Tests need to be run on distributed memory machines. Research at Calvin College and Indiana University is already following these paths.

8. REFERENCES

- [1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures*, pages 95–105. ACM Press, New York, June 1995.
<http://doi.acm.org/10.1145/215399.215427>
- [2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proc. '90 Int. Conf. on Supercomputing*, pages 2–11. SIAM, Philadelphia, Nov. 1990.
<http://www.acm.org/pubs/citations/proceedings/supercomputing/110382/p2-anderson/>
- [3] T. Axford. *The Divide-and-Conquer Paradigm as a Basis for Parallel Language Design*, chapter 2, pages 26–65. In Kronsjö and Shumsheruddin [19], 1992.
- [4] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proc. '97 Int. Conf. on Supercomputing*, pages 340–347. ACM Press, New York, July 1997.
<http://doi.acm.org/10.1145/263580.263662>
- [5] L. S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users’ Guide*. SIAM, Philadelphia, 1997.
- [6] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, Jan. 1977.
<http://doi.acm.org/10.1145/321992.321996>
- [7] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottentodi. Recursive array layouts and fast parallel matrix multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 13(11):1105–1123, Nov. 2002.
<http://www.computer.org/tpds/td2002/11009abs.htm>
- [8] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proc. ACM SIGPLAN '01 Conf. on Program. Language Design and Implementation*, pages 286–297. ACM Press, New York, 2001.
<http://doi.acm.org/10.1145/378795.378859>
- [9] M. Cole. *Parallel Software Designs*, chapter 1, pages 1–25. In Kronsjö and Shumsheruddin [19], 1992.
- [10] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Commun. ACM*, 39(11):78–85, Nov. 1996.
<http://doi.acm.org/10.1145/240455.240477>
- [11] E. Elmroth and F. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. Develop.*, 44(4):605–624, July 2000.
<http://www.research.ibm.com/journal/rd/444/elmroth.html>

- [12] J. D. Frens. *Matrix Factorization Using a Block-Recursive Structure and Block-Recursive Algorithms*. PhD thesis, Indiana University, Computer Science Department, Sept. 2002. Available as Technical Report #568. <http://cs.indiana.edu/Research/techreports/TR568.shtml>
- [13] J. D. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program.*, pages 206–216. ACM Press, New York, June 1997. <http://doi.acm.org/10.1145/263764.263789>
- [14] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Symp. on Foundations of Computer Science*, pages 285–298. IEEE Computer Society, Los Alamitos, CA, Oct. 1999. <http://www.computer.org/proceedings/focs/0409/04090285abs.htm>
- [15] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The John Hopkins University Press, Baltimore, third edition, 1996.
- [16] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 1996.
- [17] F. Ino, N. Fujimoto, and K. Hagihara. LogGPS: A parallel computational model for synchronization analysis. In *Proc. 8th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program.*, pages 133–142. ACM Press, New York, June 2001. <http://doi.acm.org/10.1145/379539.379592>
- [18] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison Wesley Longman, Boston, New York, third edition, 1997.
- [19] L. Kronsjö and D. Shumsheruddin, editors. *Advances in Parallel Algorithms*. John Wiley & Sons, New York, 1992.
- [20] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.
- [21] R. Rugina and M. Rinard. Recursion unrolling for divide and conquer programs. In S. Midkiff, J. Moreira, M. Gupta, S. Chatterjee, J. Ferrante, J. Prins, W. Pugh, and C.-W. Tseng, editors, *Languages and Compilers for Parallel Computing*, volume 2017, pages 34–48. Springer, Berlin, 2001. <http://link.springer.de/link/service/series/0558/bibs/2017/20170034.htm>
- [22] Silicon Graphics, Inc. R8000 microprocessor chip set. Technical report, Silicon Graphics, Inc., 1994.
- [23] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proc. '98 Int. Conf. on Supercomputing*. ACM Press, New York, 1998.
- [24] D. S. Wise. Undulant block elimination and integer-preserving matrix inversion. *Sci. Comp. Program.*, 33(1):29–85, Jan. 1999. <http://www.cs.indiana.edu/ftp/techreports/TR418.html>
- [25] D. S. Wise. Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro-Par 2000 — Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 774–883. Springer, Heidelberg, 2000. <http://link.springer.de/link/service/series/0558/bibs/1900/19000774.htm>
- [26] D. S. Wise, J. D. Frens, Y. Gu, and G. A. Alexander. Language support for Morton-order matrices. In *Proc. 8th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program.*, pages 24–33. ACM Press, New York, June 2001. <http://doi.acm.org/10.1145/379539.379559>
- [27] A. Y. H. Zomaya, editor. *Parallel & Distributed Computing Handbook*. Computer Engineering. McGraw-Hill, New York, 1996.